DeviceMate Software

User's Manual

019-0105 • 020212 - B

DeviceMate Software User's Manual

Part Number 019-0105 • 020212–B • Printed in U.S.A.

Notice to Users

Z-WORLD PRODUCTS ARE NOT AUTHORIZED FOR USE AS CRITICAL COMPONENTS IN LIFE-SUPPORT DEVICES OR SYSTEMS UNLESS A SPECIFIC WRITTEN AGREEMENT REGARDING SUCH INTENDED USE IS ENTERED INTO BETWEEN THE CUSTOMER AND Z-WORLD PRIOR TO USE. Life-support devices or systems are devices or systems intended for surgical implantation into the body or to sustain life, and whose failure to perform, when properly used in accordance with instructions for use provided in the labeling and user's manual, can be reasonably expected to result in significant injury.

No complex software or hardware system is perfect. Bugs are always present in a system of any size. In order to prevent danger to life or property, it is the responsibility of the system designer to incorporate redundant protective mechanisms appropriate to the risk involved.

Trademarks

Dynamic C® is a registered trademark of Z-World Inc.

Windows® is a registered trademark of Microsoft Corporation

Z-World, Inc.

2900 Spafford Street
Davis, California 95616-6800
USA

Telephone: (530) 757-3737
Fax: (530) 757-3792
www.zworld.com

Table of Contents

Chapter 1. Introduction

This manual is intended for embedded systems designers and support professionals who are using Z-World's DeviceMate software to web-enable an embedded device. The device must have a free asynchronous serial port.

1.1 Assumptions

C programming experience is assumed. A working knowledge of TCP/IP is also assumed. For further information on these topics refer to:

- *The C Programming Language* by Kernighan and Ritchie
- *Dynamic C User's Manual* by Z-World
- *An Introduction to TCP/IP* by Z-World

1.2 Definition of Terms

DeviceMate - Any Ethernet-enabled Z-World controller board can be used as a DeviceMate unit (DMU). In the DeviceMate Development Kit this unit is the RCM2200.

DeviceMate Feature Set - A group of programs that extend the ability to control and monitor an embedded target that is connected to the DMU.

Peer - If the distinction does not matter, the term peer is used to mean either the DeviceMate unit or the target.

Subsystem - Specific features in the feature set. For example, the watchdog subsystem implements the watchdog feature. As with all of the subsystems that compose the feature set, the watchdog subsystem has code running on both the DeviceMate and the target that is connected to the DeviceMate.

Target - An embedded controller board that is connected to the DeviceMate via a serial port. In the DeviceMate Development Kit this is the RCM2300; however, the target does not have to be a Rabbit-based board.

User - A programmer using the DeviceMate subsystems to develop application programs, as opposed to the end user of said applications.

XTC - Extended Target Communications is a subsystem that provides other subsystems with a reliable transport service.

1.3 Documentation Road Map

This section gives a brief description of the rest of this manual.

Chapter 2. Overview

The various components that make up the DeviceMate software environment are presented, along with a brief description of the software capabilities and instructions on using the sample programs.

Chapter 3. DeviceMate Feature Set

This chapter describes the software capabilities in detail, including the configuration information needed for applications running on the target. Sample programs for Rabbit-based targets (i.e., Dynamic C code) illustrate the subsystems and introduce their APIs. These sample programs (and others) are available in the **samples\dmtarget** directory. Sample programs for nonRabbit-based targets are not reproduced in this manual, but are available in **samples\dmtarget\Arch**. All sample programs are in source code format.

Chapter 4. Applications Running on the DeviceMate Unit

This chapter describes the requirements of applications running on the DeviceMate unit. The code for the sample program **samples\dmunit\devmate.c** is presented. This program runs several of the subsystems and an HTTP server.

Information is given about setting up the Dynamic C file system, FS2. FS2 may be used by the file system subsystem and/or the message logging subsystem.

Information on the use of the serial console available on the DeviceMate unit is also presented in this chapter.

Chapter 5. Function Reference for Target Applications

This chapter gives function descriptions for each subsystem API available on the target.

Chapter 6. Porting Guidelines for NonRabbit-Based Targets

This chapter is about the use of a nonRabbit-based target. Steps for porting code are summarized, and then specifics are given dealing with data handling, memory access, operating system interaction, and the use of preememptive multitasking environments.

Appendix A. Guidelines for Writing Custom Subsystems

Additional subsystems may be added to the DeviceMate software. This appendix gives detailed information on how to do it.

Appendix B. Using XTC

This appendix is additional support for those writing their own subsystem: it describes how a subsystem must interface with the eXtended Target Communications subsystem (XTC) to use its reliable transport service.

Chapter 2. Overview

The DeviceMate hardware and software provide a way to network-enable an embedded device. Connecting any device with RS232 to a DeviceMate unit using a serial connection (CMOS-level signals can be used if the DeviceMate unit is a Rabbit core module), gives the device a gateway to local networks and the Internet.

With DeviceMate hardware and software, your embedded device can effectively send e-mail and serve web pages. It can also upload files to a file system that was previously set up on the DeviceMate and then update variables referenced in the files. This makes it possible to remotely monitor I/O on the device with a web browser.

The DeviceMate can also act as an external watchdog for the embedded device (aka "target"). Rabbit-based embedded targets can be reprogrammed remotely via the DeviceMate.

DeviceMate software includes library support for programming both the target embedded controller and the DeviceMate controller. Since there are two controllers in a DeviceMate system, there are two sets of software components: one on the DeviceMate unit and one on the target. Application programs run on both the DeviceMate unit and the target.

Sample programs for the DeviceMate unit are in **samples\dmunit**. Sample programs for a Rabbit-based target are in **samples\dmtarget**; for nonRabbit targets they are in **samples\dmtarget\Arch**. See "Sample Programs" on page 5 for more information.

There are several sample programs provided for the DeviceMate unit. As with all sample programs, they are in source code form. They all can be used with any of the many sample programs provided for the target. For most features, you simply set some macros and, if necessary, set up the network addresses, then you compile and load the application with Dynamic C to the DeviceMate. Some features require additional programming on the DeviceMate unit.

You program your embedded target to talk to the DeviceMate via a serial port using the DeviceMate subsystem APIs provided.

2.1 Hardware Connections

For details on hardware connections, please refer to the *DeviceMate Getting Started Manual*. The following figure is a quick look at the hardware connections.

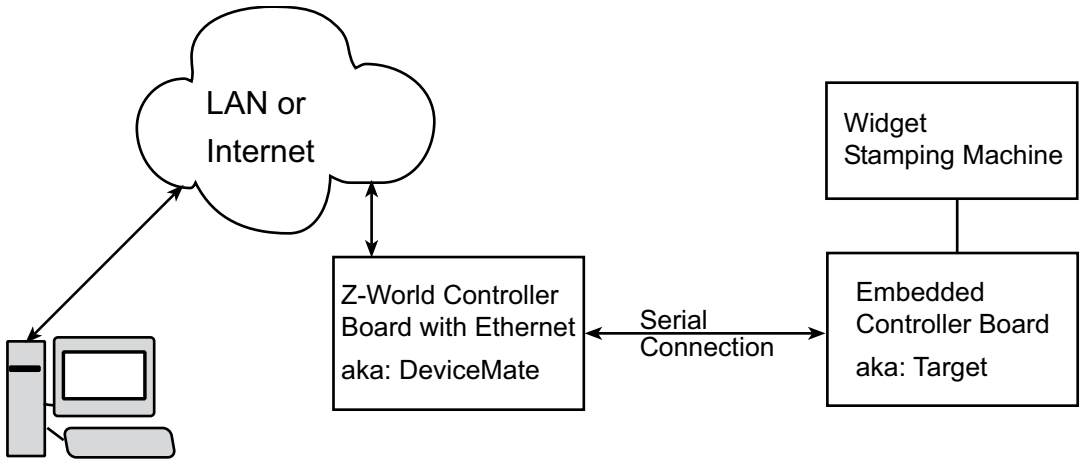


Figure 1. DeviceMate Connected to an Embedded System

2.2 Software Components

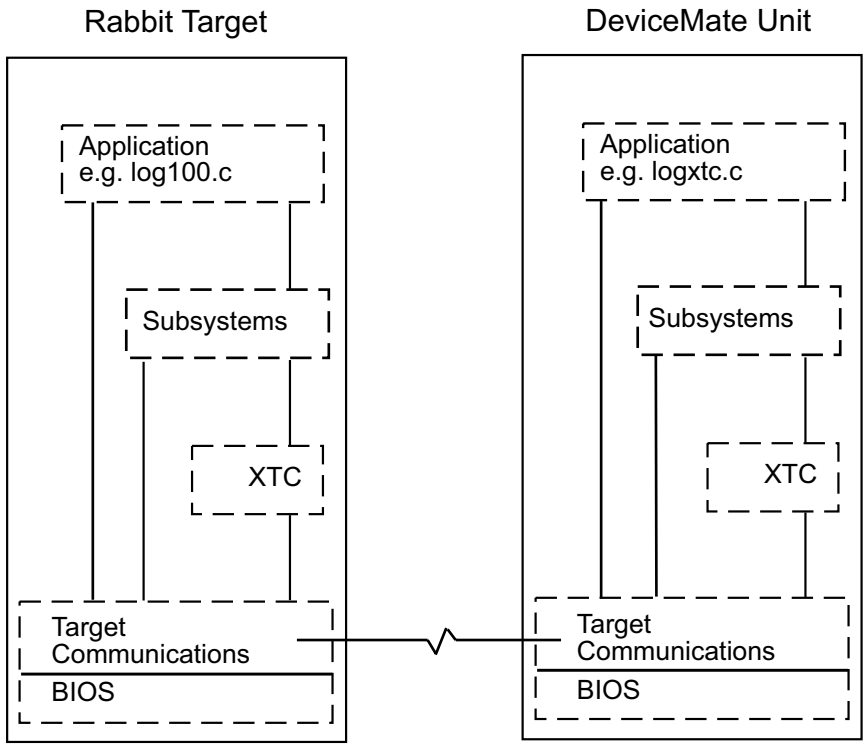
The DeviceMate feature set comprises:

- E-Mail
- File System
- Message Logging
- Remote Monitoring of Target I/O
- Remote Program Download (Rabbit-based targets only)
- Serve Web Pages
- TCP and UDP Sockets
- Update Variables on Web Pages
- Watchdogs

Several software subsystems implement the DeviceMate feature set. All of the subsystems used by the target are described in Chapter 3 starting on page 9. All of the subsystems used by the DeviceMate unit are described in Chapter 4 starting on page 29.

2.2.1 Layers of Communication

Many of the subsystems are clients of eXtended Target Communications (XTC). XTC provides a reliable transport layer. But, as the flowchart below indicates, subsystems can call Target Communications (TC) routines directly, as can applications. If XTC is not used, then packet delivery to the peer is not guaranteed.



2.2.2 Sample Programs

The sample programs run in pairs (except for **loader.c**).

Table 1. Application Pairs on the DeviceMate Unit and the Target

Application on DeviceMate	Applications on Target
devmate.c	**act_open.c**, **dns_look.c**, **log100.c**, **pas_open.c**, **smtp.c**, **tcp_time.c**, **udp_echo.c**, **var.c**, **wd.c**
devmate_2flash.c	**act_open.c**, **all.c**, **dns_look.c**, **fs.c**, **fs_block.c**, **fs_gen.c**, **fs_tiny.c**, **log100.c**, **pas_open.c**, **smtp.c**, **tcp_time.c**, **udp_echo.c**, **var.c**, **wd.c**
devmate_fs.c	**act_open.c**, **all.c**, **dns_look.c**, **fs.c**, **fs_block.c**, **fs_gen.c**, **fs_tiny.c**, **log100.c**, **pas_open.c**, **smtp.c**, **tcp_time.c**, **udp_echo.c**, **var.c**, **wd.c**
devmate_loader.c	**act_open.c**, **all.c**, **dns_look.c**, **fs.c**, **fs_block.c**, **fs_gen.c**, **fs_tiny.c**, **log100.c**, **pas_open.c**, **smtp.c**, **tcp_time.c**, **udp_echo.c**, **var.c**, **wd.c**
devmate_zconsole.c	**act_open.c**, **all.c**, **dns_look.c**, **fs.c**, **fs_block.c**, **fs_gen.c**, **fs_tiny.c**, **log100.c**, **pas_open.c**, **smtp.c**, **tcp_time.c**, **udp_echo.c**, **var.c**, **wd.c**
fs.c	**fs.c**, **fs_block.c**, **fs_gen.c**, **fs_tiny.c**
fs_flash.c	**fs.c**, **fs_block.c**, **fs_gen.c**, **fs_tiny.c**
loader.c	Not applicable.
logxtc.c	**log100.c**
tcp.c	**act_open.c**, **dns_look.c**, **pas_open.c**, **smtp.c**, **tcp_time.c**, **udp_echo.c**
var.c	**var.c**
wd.c	**wd.c**

To run the sample programs, follow this sequence of steps:

1. Connect the programming cable to your PC and the DeviceMate.
2. Connect power to the DeviceMate unit.
3. While running Dynamic C v 7.10 or later, configure and compile one of the sample programs for the DeviceMate. See “Applications Running on the DeviceMate Unit” on page 29 for configuration information for these programs.

   Note: In the documentation for Dynamic C the term “target” is used to refer to the controller board that is connected to Dynamic C via the programming cable. When “target” is used by Dynamic C it is limited to that definition. When using Dynamic C, any menus, dialog boxes, or other messages that use the term target are (almost) NEVER referring to a device that is serially connected to a DeviceMate unit. Dynamic C is (almost) ALWAYS referring to the controller board that it is connected to it via the programming cable. The exception is when using the remote program download feature. After the Ethernet loader is running on the DeviceMate unit, Dynamic C will recognize the device serially connected to the DeviceMate as the target.

2.2.2.1 For Rabbit-based Targets:

1. Disconnect power to the DeviceMate.
2. Disconnect the programming cable.
3. Connect the programming cable to the target.
4. Connect power to the DeviceMate and target.
5. While running Dynamic C (version 7.10 or later), compile the application to the target and run it. For example, to check out the functionality of the watchdog subsystem, open the file **samples\dmtarget\wd.c**. Hit **F9** to compile and run the watchdog application.

2.2.2.2 For nonRabbit-based Targets:

The initial setup for all sample programs is the following:

1. Create a base directory on the target device (or development machine). The following steps assumes this directory is called **dm**.
2. Copy the target-specific **.zip** or **.tar** file to **dm**.

   For example:

   ```
   CD dm
   COPY e:\dc710\samples\dmtarget\arch\i386\tc_bc.zip
   ```

   or, for Unix-based systems:

   ```
   cd dm
   cp /cdrom/dc710/samples/dmtarget/arch/sparc_sol8/tc_solaris.tar
   ```

3. Unpack the archive.

   For example:

   ```
   PKUNZIP tc_bc.zip
   ```

   or, for Unix-based systems:

   ```
   tar xvf tc_solaris.tar
   ```

4. Examine the README files for detailed instructions.
5. Configure the library. This means editing **dm/tc_conf.h** to accurately reflect the target machine architecture. It is recommended that the debugging flags be defined, at least initially.
6. If the target is not one that is supported "out of the box," you will possibly need to write some adapter code. See “Porting Guidelines for NonRabbit-Based Targets” on page 101 for directions on how to do this.
7. Compile the library and samples using the "make" command.
8. Connect the serial port of the target processor to that of the DeviceMate.

To run a particular sample:

1. Ensure that DeviceMate and target configurations match, especially serial port speed.
2. Run the appropriate DeviceMate sample program (e.g., **devmate.c**), with or without Dynamic C debugging (stdio window etc.). We recommended running with the Dynamic C programming cable connected and the debugging flags turned on in **tc_conf.lib**.
3. Execute the appropriate sample program on the target.

2.3 Further Programming

The sample programs can be used as templates for further code development. Depending on the application running on the embedded system, this could be very little or quite a bit of design and coding work.

Users can also develop customized subsystems. Information needed for doing that can be found under “Guidelines for Writing Custom Subsystems” on page 113.

2.3.1 Faster Debug/Development

With a Rabbit-based target, using two programming cables and Dynamic C project files allows for a faster debug/development cycle. The default project file, **default.dcp**, uses COM1 for serial communication. Use the **File | Project | Save As . . .** menu option to create a project file with a descriptive name, such as **dmUnit.dcp**. Then go to **Options | Communications** and choose COM2. Again, use the **File | Project | Save As . . .** menu option to create a new project file with a descriptive name, this time something like **dmTarget.dcp**. Now you can run two copies of Dynamic C side by side, each using a different project file with the correct comm port for the controller it is talking to.

Chapter 3. DeviceMate Feature Set

The DeviceMate feature set is implemented by certain software subsystems.

The subsystems are:

- E-Mail
- File System
- Message Logging
- Remote Program Download (Rabbit-based targets only)
- Watchdogs
- TCP and UDP Sockets
- Web Page Variables

This chapter is about the subsystems running on the target: their configuration and an overview of their APIs. The configuration of the companion subsystems that run on the DeviceMate are discussed in Chapter 4 starting on page 29.

Library Support for Target Devices

An application program running on the target device must ask for all subsystems it wants to use. This is done using the macros shown in the table below.

Subsystem Name	Request for Service Macro	Libraries for Rabbit-Based Targets
E-mail	**USE_TC_SMTP**	**dm_smtp.lib**
File System	**USE_TC_FS**	**dm_fs.lib**
Message Logging	**USE_TC_LOG**	**dm_log.lib**
Remote Program Download	**USE_TC_LOADER**	None
TCP/IP	**USE_TC_TCPIP**	**dm_tcp.lib**
Watchdogs	**USE_TC_WD**	**dm_wd.lib**
Web Page Variables	**USE_TC_VAR**	**dm_var.lib**

The macro must be defined in the application program. For example:

```
#define USE_TC_VAR          // requesting the web page variables subsystem
```

After the request for service macros are defined, the application code must include one of the following statements:

```
#use "tc_conf.lib"          // for Rabbit-based targets
#include "tc_conf.h"        // for nonRabbit-based targets
```

> **Note:** In the rest of this chapter, instead of citing the files needed for both Rabbit and nonRabbit targets, only the **.lib** file will be mentioned.

The **tc_conf.lib** file uses the request for service macros to determine which subsystem libraries to include. For example if **USE_TC_VAR** is defined, then **tc_conf.lib** pulls in **dm_var.lib**.

The remote program download macro, **USE_TC_LOADER**, does not work the same way as the other request for service macros. There is no **dm_*.lib** file associated with it because the target is not running an application when the remote download feature is being used. If the downloaded program is a DeviceMate application, then including the statement **#define USE_TC_LOADER** in its code will configure the correct serial port for continued communication with the DMU while the downloaded program is running.

Serial Port Macros

The sample programs use serial port D by default. This can be changed by defining a serial port macro in the application code. The macros are considered in the order they appear below, e.g., if the macro for serial port B is defined, then serial port B is used unless **DEVMATE_SERA** is defined; the others (C and D) are not considered even if they are defined.

DEVMATE_SERA: Define this macro with **#define** to use serial port A.

DEVMATE_SERB: Define this macro with **#define** to use serial port B.

DEVMATE_SERC: Define this macro with **#define** to use serial port C.

DEVMATE_SERD: Define this macro with **#define** to use serial port D.

If the Development Board that came with the DeviceMate Development Kit is being used with the RS232 connector, then there are some hard-wired restrictions regarding serial ports. Please see the *DeviceMate Getting Started Manual* for more information.

Please note that serial port A is used by the programming cable. This is also the serial port used by the target used when the DeviceMate is remotely downloading a program to it.

The speed of data transfer through the serial port defaults to 115200 bps. This can be overwritten by the macro **DEVMATE_SERIAL_SPEED**. This value must match **TARGETPROC_SERIAL_SPEED** on the DeviceMate side.

Required Function Calls for Target Applications

There are 2 functions that must be called by any application running on the embedded target device. A call to **devmate_init()** is required to initialize target communications with the DeviceMate unit. To process packets and otherwise drive the system, repeated calls to **devmate_tick()** must be made. This is usually done in a tight, endless loop:

```
for (;;) {
   devmate_tick();
         .
         .
         .
}
```

Using µC/OS-II

The maximum number of events supported by µC/OS-II is defined by the macro **OS_MAX_EVENTS**. This macro includes the number of semaphores, message mailboxes and any message queues being used by µC/OS-II. Two must be added to **OS_MAX_EVENTS** for each of the DeviceMate subsystems that are used; also, one must be added if any of the included subsystems use XTC.

In the application code, follow the define of **OS_MAX_EVENTS** with a **#use "ucos2.lib"** statement. This ensures that the definition supplied outside of the library is used, rather than the default in the library. Also the inclusion of **ucos2.lib** must happen before the inclusion of any DeviceMate libraries. **OSInit()** and **OSStart()** must be called by the application to initialize and start the operating system.

For more information on the Dynamic C implementation of µC/OS-II please see the *Dynamic C User's Manual.* See sample programs **var_ucos.c** and **fs_ucos.c** for examples of using DeviceMate subsystems and µC/OS-II.

> **Note:** µC/OS-II is not available in the DeviceMate Special Edition of Dynamic C. It is available in Dynamic C Premier. The same is true for sample programs **var_ucos.c** and **fs_ucos.c**.

3.1 TCP/IP Subsystem

This subsystem makes both TCP and UDP sockets available to the target. The target can actively or passively open a TCP socket via the DeviceMate; the target can read and write the socket; the target can close or abort the socket; the target can check socket status and can ask whether or not an error has occurred.

This API allows TCP/IP functionality, including UDP and DNS lookups, to be accessed from the target. The target may send and receive data from an open UDP socket.

The TCP/IP subsystem on the DeviceMate side runs a complete TCP/IP stack.

3.1.1 Configuration Macros for TCP/IP Subsystem

These macros may be defined in the application running on the target before the inclusion of the library **`tc_conf.lib`**.

`DEVMATE_TCP_DEBUG`

Defaults to not defined. If defined, then Dynamic C debugging is turned on for all functions herein.

`DEVMATE_TCP_MAXCHANS`

Defaults to 4. This is the maximum number of XTC channels allowed.

`DEVMATE_TCP_MAXRESOLVE`

Defaults to 1. Specifies the maximum number of concurrent DNS (Domain Name Server) look-ups. Set to 0 if DNS is not required.

`DEVMATE_TCP_MAXSOCK`

Defaults to 4. Maximum number of sockets (implemented as XTC "channels") that are allowed. Generally, this limits the number of sockets on the target since the target is assumed to be more resource-constrained than the DeviceMate. The actual maximum is determined by the minimum value configured on the target or DeviceMate unit. Note that this value does not include the so-called "control" channel (channel 0). Available socket numbers run from 1 to **`DEVMATE_TCP_MAXCHANS`** inclusive. The function **`devmate_tcp_maxsocket()`** returns the actual negotiated maximum socket number.

`DEVMATE_TCP_NUMRXBUF`

Defaults to 3. Specifies the number of target communications receive buffers. A higher number allows greater performance, at the expense of additional memory usage.

`DEVMATE_TCP_NUMTXBUF`

Defaults to 1. Specifies the number of target communications transmit buffers. Do not set higher than 2. 2 is only necessary if the highest possible speed is required.

`DEVMATE_TCP_TCBUFSIZE`

Defaults to 133, which allows for 128-byte data packets plus the 5-byte XTC header.

`DEVMATE_TCP_XTCBUFSIZE`

Defaults to 256. Must be greater than or equal to **`DEVMATE_TCP_TCBUFSIZE`** - 5.

3.1.2 API Functions for TCP/IP Subsystem

The functions in **dm_tcp.lib** are nonblocking, i.e. they all set some internal state, but return immediately (possibly without actually accomplishing their intended goal). Thus, it is usually necessary to call the same function multiple times in order to complete the action, a necessity noted in the description of the function.

Most of the functions require a "socket" parameter. Unlike the native **dcrtcp.lib** TCP stack, where socket parameters are pointers to structures, sockets in this library are simply small numbers that are selected by the applications. Note that the socket number space is shared between TCP and UDP sockets. For applicatons with resource constraints, sockets should be closed when not in use. Allocation of socket numbers is entirely the application's responsibility.

Only a limited number of sockets may be in use at any one time. This number is configured in **tc_tcp.lib** via the **DEVMATE_TCP_MAXSOCK** define. This number defaults to 4, but may be overridden by the application. The total number of sockets is the minimum of the number configured for the target and the DeviceMate unit. This minimum number is available by calling the **devmate_tcp_maxsocket()** function, after **devmate_sock_init()**. The returned number may be less than **DEVMATE_TCP_MAXSOCK** if the DeviceMate unit does not support as many sockets as the target.

3.1.2.1 Functions Grouped by Task

The TCP/IP API functions below are linked to their full description in Chapter 5 “Function Reference for Target Applications.” Here the functions are listed to show what is available.

Initialize connection to DeviceMate:

```
devmate_sock_init()
devmate_tcp_status()
```

Open TCP or UDP sockets:

```
devmate_tcp_open()
devmate_tcp_listen()
devmate_udp_open()
```

Check socket status:

```
devmate_tcp_isestablished()
devmate_tcp_error()
devmate_tcp_readable()
devmate_tcp_writable()
devmate_tcp_isclosed()
```

Read or write TCP (stream) data:

```
devmate_tcp_fastread()
devmate_tcp_preread()
devmate_tcp_fastwrite()
```

Close socket:

```
devmate_tcp_close()
devmate_tcp_abort()
devmate_udp_close()
```

Read or write UDP (packet) data:

```
devmate_udp_sendto()
devmate_udp_send()
devmate_udp_recvfrom()
devmate_udp_recvdata()
```

Miscellaneous:

```
devmate_ip_resolve()
devmate_tcp_maxsocket()
```

3.1.3 TCP and UDP Sample Program

Could the perpetual flashing of "12:00" on VCRs and microwave ovens be a thing of the past? Yes, if their controller can be made to talk to a DeviceMate!

Specifically, in the sample program **tctcp_time.c**, the TCP/UDP tunneling facility of the DeviceMate is used to connect to a sequence of Time servers. The target handles the Time protocol, which is extremely simple, leaving the DeviceMate to handle the details of the ethernet/internet connection.

This is the basis for something useful. The nominated server(s) are queried for their UTC time. (UTC is the international time standard.) The result from each server is adjusted for the network delay (assumed to be half of the round-trip time). If there were more than 2 servers, then the servers whose times were "most distant" from the average of the others are discarded, until a minimum of 2 or half the original number of servers remain. The final computed time is the average of the remaining servers. If all remaining servers are within about 60 seconds of each other, then the discarding process is cut short.

If no servers can be found, the current real-time clock is used (**SEC_TIMER**).

When the time is computed, the target processor then opens its own Time and Daytime servers (RFC868 and RFC867). You can then telnet to port 13 on this processor to obtain the current time, or at least its best guess at the time.

Note that UDP is used to query the servers, but the target allows both TCP and UDP queries to its server sockets.

3.1.3.1 Running tctcp_time.c

The source code for this sample program is available in the **samples\dmtarget** folder. For brevity's sake, the code is not reproduced here. It is recommended that users run this sample program to learn more about it.

3.1.3.2 Porting Note

This demo (**tctcp_time.c**) is written for a Rabbit-based target. However, since DeviceMate functionality is specifically intended to be platform-independent, this program should work (with minor modifications) on other processors. The main areas to examine are:

- **SEC_TIMER** - define this to be a macro which returns the number of seconds since some convenient epoch. The Rabbit 2000 uses 1980. The RFC specifies midnight January first 1900. Redefine the **EPOCH_DIFF** symbol to be the correct number of seconds between the epochs. Must be a **uint32** value.
- **MS_TIMER** - should be a macro returning the number of milliseconds since an arbitrary epoch (e.g. bootup time). Should be a **uint16** or **uint32** value. The actual resolution does not have to be 1ms. 1/10 second would be OK.
- **#use** - change to appropriate **#include**(s).
- **format_timestamp()** - change this function to convert timestamps to a readable format.
- **main()** - examine the DeviceMate setup etc.

3.2 E-Mail Subsystem

The DeviceMate can be configured to send e-mail on behalf of the target. There is a target-side library to handle the SMTP protocol and connect to a mail server through a DeviceMate running TCP/IP services.

The library **dm_smtp.lib** is a modification of the regular **smtp.lib** in Dynamic C. The API is nearly identical. The functions are described in Chapter 5 starting on page 74.

3.2.1 E-Mail Sample Program

This sample program illustrates the e-mail subsystem. The e-mail subsystem is a layer on top of the TCP/IP subsystem. This requires that a call to initialize the TCP/IP subsystem be made for any application that uses the e-mail subsystem.

Program Name: smtp.c

```
#define USE_TC_SMTP
#ifdef CC_VER
   #memmap xmem
   #use "tc_conf.lib"
#else
   #include "tc_conf.h"
   #include <stdio.h>
#endif
#define MY_SMTP_SERVER "mail.somewhere.org"
#define MY_SMTP_DOMAIN "somewhere.org"
#define FROM "Devmate@somewhere.org"
#define TO  "nobody@nowhere.org,everybody@everywhere.org"
#define SUBJECT "test mail"
int main(){
   char message[256];
   devmate_init();
   printf("devmate_sock_init: %d\n",
     devmate_sock_init(NULL, NULL, NULL, NULL, NULL, NULL)
     );
   devmate_smtp_setserver(MY_SMTP_SERVER);
   devmate_smtp_setdomain(MY_SMTP_DOMAIN);
   strcpy(message,
     "Warning!\r\nTemperature in the computer room over 40 deg C!"
     );
   devmate_smtp_sendmail(TO, FROM, SUBJECT, message);
   while (devmate_smtp_mailtick() == DM_SMTP_PENDING);
   if (devmate_smtp_status() == DM_SMTP_SUCCESS)
      printf("Message sent\n");
   else
      printf("Error sending message\n");
   while (1)
      devmate_tick();
}
```

3.2.2 Configuration Macros for E-Mail Subsystem

These macros may be defined in the application running on the target before the inclusion of the library **`tc_conf.lib`**.

`DEVMATE_SMTP_DOMAIN`

The domain name to state when connecting to an SMTP server. This defaults to "" (none). Some SMTP servers require a client to declare it's domain name when connecting though, so this macro should be defined to an appropriate domain name.

`DEVMATE_SMTP_SERVER`

Defaults to "" (none). String constant defining the default SMTP server. This may be overridden at runtime using **`devmate_smtp_setserver()`**.

`DEVMATE_SMTP_SOCK`

Defaults to 2. Specifies the TCP/IP subsystem socket number to use when calling **`devmate_smtp_mailtick()`**. This may be overridden at runtime using **`devmate_smtp_setsocket()`**.

`DEVMATE_SMTP_TIMEOUT`

Defaults to 300. Specifies timeout for sending mail, in seconds. Whenever a response is received from the mail server, this timeout is reset.

3.3 Web Page Variables Subsystem

The target sends requests to the DeviceMate to create a variable or update a variable's value using the API in **dm_var.lib**. The variable is stored on the DeviceMate and is displayed whenever a web page containing the variable is displayed.

3.3.1 Using the File System Subsystem

The file system subsystem may be used to upload the files that contain variable references. This will be convenient for uploading the initial HTML file to the DeviceMate unit. After that, the web page variable subsystem can be used to update the variables in the HTML file without having to upload the entire file.

3.3.2 Displaying the Variables on Web Pages

Variables can be updated on web pages using SSI commands in the HTML code. SSI commands are an extension of the HTML comment command. The web page's HTML code must contain a line similar to:

```
<!--#echo var="variable_name" -->
```

This expression will be replaced by the value of **variable_name** by the web server when the page is requested by a web browser.

3.3.3 Frequency of Variable Updates

It is important to limit the rate of variable updates. Updating too frequently may flood the serial line. Notice that in **var.c**, an update is limited to once every 10 seconds. The updates should occur as frequently as the information is needed on the DeviceMate side, as opposed to the frequency with which new values may become available on the target side.

3.3.4 Variables Sample Program

If **TC_USE_VAR** is defined on the DeviceMate unit and the target, then the application **samples\dmunit\devmate.c** runs an HTTP server that accepts variable updates from the target that is running **samples\dmtarget\var.c**. The variables can be viewed by a web browser by requesting **var.shtml** at the DeviceMate's IP address. Do a refresh from the browser to see the updates.

Program Name: var.c

```
#define DEVMATE_VAR_MAXVARS 10
#define USE_TC_VAR
#ifdef CC_VER
   #memmap xmem
   #use "tc_conf.lib"
#else
   #include "tc_conf.h"
   #include <stdio.h>
#endif
int main(void){
   long starttime, oldseconds, seconds;
   char strvar[20];
   float floatvar;
   devmate_init();
   starttime = SEC_TIMER;
   oldseconds = 0;
   seconds = 0;
   strcpy(strvar, "Good bunny.");
   floatvar = 42.42;
   devmate_var_add("seconds", &seconds, VAR_INT32, "%ld", 0,
    VAR_SERVER_HTTP);
   devmate_var_add("string", strvar, VAR_STRING, "%s", 20,
    VAR_SERVER_HTTP);
   devmate_var_add("float", &floatvar, VAR_FLOAT32, "%f", 0,
    VAR_SERVER_HTTP);
   for (;;) {
      devmate_tick();
      seconds = SEC_TIMER - starttime;
      if (((seconds % 10) == 0) && (seconds > oldseconds)) {
         oldseconds = seconds;
         devmate_var_update("seconds");
         devmate_var_update("string");
         devmate_var_update("float");
         floatvar += 3.14159;
         if (strvar[0] == 'G')
            strcpy(strvar, "Bad bunny.");
         else
            strcpy(strvar, "Good bunny.");
      }
   }
}
```

3.3.5 Configuration Macros for Variables Subsystem

These macros may be defined in the application running on the target before the inclusion of the library **`tc_conf.lib`**.

`DEVMATE_VAR_DEBUG`

This macro is not defined by default. If defined, then debugging information will be output as the program executes.

`DEVMATE_VAR_MAXDATA`

Defaults to 32. This defines the maximum length of the data (or contents) for a variable.

`DEVMATE_VAR_MAXFORMAT`

Defaults to 8. This defines the maximum length of the printf-style format specifier for a variable, including the **`NULL`** terminator.

`DEVMATE_VAR_MAXNAME`

Defaults to 8. This defines the maximum length of a variable name, including the **`NULL`** terminator. This macro determines the maximum length of the argument **`char * name`** taken by many of the API functions.

`DEVMATE_VAR_MAXVARS`

Defaults to 10. Specifies the number of variables for which the target processor will keep information. This limits the number of variables that you can add. Note that the DeviceMate may not accept as many variables as specified here. See “SSPEC_MAXSPEC” on page 32 for more information.

`DEVMATE_VAR_NUMRXBUFS`

The number of variable receive buffers on the target. Defaults to 2.

`DEVMATE_VAR_NUMTXBUFS`

The number of variable transmit buffers on the target. Defaults to 1.

`DEVMATE_VAR_TCBUFSIZE`

Defaults to 133, which allows for 128-byte data packets plus the 5-byte XTC header.

`DEVMATE_VAR_XTCBUFSIZE`

This defaults to 128. This is used for XTC streams as the default buffer size for XTC.

3.4 File System Subsystem

The file system feature allows the embedded target to off load the running of the file system to the DeviceMate. The target can create and delete files on the DeviceMate where the files can be added to the **zserver.lib** spec table and thereby accessed over the Internet. For information on **zserver.lib**, please see the *Dynamic C TCP/IP User's Manual*.

3.4.1 File System Configuration

The Dynamic C file system (FS2) must be set up on the DeviceMate before the file system subsystem can be used by the target. If you are using one of the the sample program supplied by Z-World that makes use of the file system subsystem (e.g., **devmate_fs.c**) you will probably have to make some code changes to it to configure FS2 according to your needs.

3.4.2 File System Sample Program

The application **samples\dmtarget\ fs_tiny.c** makes minimal use of the file system subsystem. It will contact the DeviceMate and upload some files to it that will be made available to a web server on the DeviceMate. After running this application, the web page should be visible at:

http://[devmate_ip]/index.html

Program Name: fs_tiny.c

```
#ximport "samples/tcpip/http/pages/static.html" index_html
#ximport "samples/tcpip/http/pages/rabbit1.gif" rabbit1_gif

// #define FORMAT
#define USE_TC_FS
#use "tc_conf.lib"
DMFile file;

// Send a file to the DeviceMate, and return it's file ID
int send_file(long data, char *name){
   auto int retval;
   auto long big_len;
   auto int length, offset, writesize;
#define BUFSIZE 64
   auto char buf[BUFSIZE];
   auto FNumber id;
   if(TC_SUCCESS == devmate_fs_idlookupB(&file, name, &id, 0))
      devmate_fs_deleteB(&file, id, 0);
   if(xmem2root((char *)&big_len, data, 4))
      exit(1);
   length = (int)big_len;
   offset = 0;

   printf("Opening file '%s'...\n",name);

   while(TC_PENDING == (retval = devmate_fs_open(&file, 0, name,
    TC_FS_FLAGS_NEWID)))

      devmate_tick();
   if(TC_SUCCESS != retval)
      exit(2);
   printf("Sending the file...\n");
   while(offset < length) {
      writesize = length - offset;
      if(writesize > BUFSIZE)
         writesize = BUFSIZE;
      if(xmem2root(buf, data + 4 + offset, writesize))
         exit(3);
      retval = devmate_fs_append(&file, buf, writesize);
      if(-1 == retval)
         exit(4);
      offset += retval;
      devmate_tick();
   }
   printf("Send complete - closing the file...\n");
   while(TC_PENDING == (retval = devmate_fs_close(&file,&id)))
      devmate_tick();
   if(TC_SUCCESS != retval)
      exit(5);
   printf("File '%s' sent correctly!\n",name);
   return id;
}
```

Program Name: fs_tiny.c (continued)

```
void main(){
   int rv;
   long timeout;
   FNumber index_file, logo_file, test_id;

   serDclose();
   devmate_init();

#ifdef FORMAT
   rv = devmate_fs_syncB(&file,TC_FS_FORMAT,0);
#else
   rv = devmate_fs_syncB(&file,0,0);
#endif
   if(TC_SUCCESS != rv) {
      printf("Error in init!\n");
      exit(-1);
   } else {
      printf("INIT was successfull!\n");
   }
   /*
    * Send some data
    */
   index_file = send_file(index_html,"/index.html");
   logo_file = send_file(rabbit1_gif,"/rabbit1.gif");

   printf("All done!\n");
}
```

3.4.2.1 Blocking vs. NonBlocking Functions

In a working environment, blocking functions are almost never the way to go. They are resource hogs. Their one advantage is that they simplify the code and if the holding on to resources is not an issue they are useful alternatives to their nonblocking counterparts. The sample program above, **`fs_tiny.c`**, uses blocking functions. Look at **`samples\dmtarget\fs_gen.c`** to see an example using the file system with nonblocking functions.

3.5 Message Logging Subsystem

This feature allows the target controller to log variable-length messages to the DeviceMate unit. All records are time-stamped and sent immediately to the DeviceMate for storage.

The message logging subsystem uses XTC services.

3.5.1 Message Filtering

All messages have a priority code and a facility number. The logging subsystem can key on either or both of these when determining if a message should be logged. The default condition is to allow all messages to be logged. This can be changed by API functions devmate_log_setfacilityfilter() and devmate_log_setpriorityfilter().

Priority codes are 3 bits. There are no defined priority codes. You may wish to use the standard “syslog” encoding as follows:

0: Emergency; 1: Alert; 2: Critical; 3: General errors; 4: Warning; 5: Notice; 6: Informational; 7: Debug

Facility numbers are 5 bits. There are no defined facility numbers. Facilities are intended to be used as a way to separate messages by categories. For example, an application may define a facility number for analog inputs and another facility number for analog outputs. Up to 32 facilities may be defined.

3.5.2 Message Logging Sample Program

The application **samples\dmtarget\log100.c** demonstrates the use of DeviceMate to perform logging. It sends 100 "random" log messages to various destinations.

Program Name: log100.c

```
#define USE_TC_LOG
#ifdef CC_VER
   #memmap xmem
   #use "tc_conf.lib"
#else
   #include "tc_conf.h"
   #include <stdio.h>
#endif
int main(){
   auto int i, rc;
   auto uint16 timer;
   auto char buf[116];
#define NUM_TEST_MESSAGES 3
   static const char * test_messages[NUM_TEST_MESSAGES] =
      {"Z-World","Rabbit Semiconductor","Talk Async Serial!"};
#define NUM_TEST_INTERVALS 5
   static const uint16 test_intervals[NUM_TEST_INTERVALS] =
      {1000, 100, 100, 300, 700};
   devmate_init(); // Initialize target communications
reopen_log:
   devmate_log_init();
   while (!devmate_log_status()) devmate_tick();
   for (i = 0; i < 100; i++) { // send test messages
      sprintf(buf, "Log message %d.  %s", i,
test_messages[i%NUM_TEST_MESSAGES]);
      // alternate between facility 0 and 1
      rc = devmate_log_put((i&1)<<3, buf, strlen(buf));
      switch (rc) {
         case 0: // normal
            break;
         case 1: // not buffered, trying again in 400 ms
            timer = (uint16)MS_TIMER + 400;
            while ((int16)((uint16)MS_TIMER - timer) < 0)
               devmate_tick();
            i--;
            continue;
         case -2: // message too long
            break;
         default: // logging unavailable, attempt to re-establish
            goto reopen_log;
      }
      timer = (uint16)MS_TIMER + test_intervals[i%NUM_TEST_INTERVALS];
      while ((int16)((uint16)MS_TIMER - timer) < 0)
         devmate_tick();
   }
   while (1)
      devmate_tick();
   return 0;
}
```

The **samples\dmunit\logxtc.c** program must be running on the DeviceMate unit to give **log100.c** someone to talk to.

3.5.3 Data Types

Log data may take any form defined in the application. The data is sent by the message logging subsystem in binary format, i.e., a null terminator is not needed. Messages consist of a binary string of length 0 to 115 bytes.

3.5.4 Storage of Logging Messages

Logging messages may be stored in several places. On the DeviceMate unit they may be stored in an xmem buffer or in one or more files in FS2. (If FS2 and Flash memory are used, remember that Flash memory is limited by how many times it can be updated and should be used only when updates occur at most every couple of minutes.) Messages can also be logged to stdout.

In the sample program **log100.c**, messages alternate between facility 0 and 1. The default mapping maps facility 0 to the xmem buffer since the application running on the DeviceMate unit (**logxtc.c**) has defined **LOG_USE_XMEM**. The default mapping maps facility 1 to the stdio window.

The storage options are determined by the application running on the DeviceMate unit. If FS2 is used, it must be set up on the DeviceMate prior to running an application on the target that makes use of it.

3.5.5 Message Logging Configuration Macros

DEVMATE_LOG_TCBUFSIZE

This is the maximum number of bytes for the TC buffer. Defaults to 128+5

DEVMATE_LOG_XTCBUFSIZE

This is the maximum number of bytes for the XTC buffer. Defaults to 256

DEVMATE_LOG_NUMRXBUF

This is the maximum number of message logging receive buffers on the target. Defaults to 1.

DEVMATE_LOG_NUMTXBUF

This is the maximum number of message logging transmit buffers on the target. Defaults to 1.

3.6 Software Watchdogs Subsystem

Software watchdogs are basically decrementing timers that can alert the outside world that something is wrong. They minimize the amount of time that a target can stay in an unknown state by requiring the target to periodically communicate that everything is okay. The watchdog subsystem API allows the target to create and hit watchdogs.

XTC services are not used by this subsystem.

3.6.1 Watchdog Sample Program

This program is intended to be run on the target. It sets up a number of watchdogs on the DeviceMate and keeps hitting them. See **`\samples\dmunit\wd.c`** for the DeviceMate program intended to run with this program.

Program Name: wd.c

```
#define USE_TC_WD
#ifdef CC_VER          // if this is Dynamic C
   #memmap xmem
   #use "tc_conf.lib"
#else                 // else this is a nonRabbit target
   #include "tc_conf.h"
   #include <stdio.h>
#endif
int main(void)
{
   int rc;
   uint16 nexthit;
   devmate_init();
   // Setup watchdogs. Wait 5 seconds for response.
   if ((rc = devmate_wd_init(5000)) != DEVMATE_WD_ACKED) {
      printf("devmate_wd_init: failed %d\n", rc);
      exit(19);
   }
   // Add watchdog called FIDO. Wait 5 seconds for response.
   // Watchdog must be hit every 12 sec.
   rc = devmate_wd_add("FIDO", 12000, 5000);
   if (rc != DEVMATE_WD_ACKED) {
      printf("devmate_wd_add: unexpected response %d\n", rc);
      exit(20);
   }
   for(;;) {
      devmate_tick();
      // Hit watchdog every 2 seconds
      if ((int16)((uint16)MS_TIMER - nexthit) > 0) {
         devmate_wd_hit("FIDO");
         nexthit = (uint16)MS_TIMER + 2000;
      }
   }
}
```

3.6.2 Watchdog Subsystem Configurable Variables

These macros may be defined in the application running on the target before the inclusion of the library **`tc_conf.lib`**.

`DEVMATE_WD_MAXPENDING`

This is the maximum number of concurrent watchdogs on the target. Defaults to 12.

`DEVMATE_WD_RETRANSTIME`

Amount of time to wait before retransmitting a request. Defaults to 500 ms.

`DEVMATE_WD_HITTIME`

Minimum amount of time to wait before transmitting a watchdog hit. Defaults to 200 ms.

`DEVMATE_TXSPEED`

Minimum amount of time to wait before transmitting another watchdog packet. Defaults to 50 ms.

`DEVMATE_WD_NUMTXBUF`

This is the maximum number of transmit buffers available to the watchdog subsystem on the target. Default is 2.

`DEVMATE_WD_NUMRXBUF`

This is the maximum number of receive buffers available to the watchdog subsystem on the target. Default is 2.

Chapter 4. Applications Running on the DeviceMate Unit

There are several sample programs provided for the DeviceMate unit. Depending on the subsystems being used, an application running on the DeviceMate unit may set up FS2, run a web server, be a SMTP mail client and/or run a serial-based console.

This chapter will be mainly concerned with running **samples\dmunit\devmate.c**. This application enables the TCP/IP subsystem, the web page variables subsystem, the watchdog subsystem and the message logging subsystem. Any application running on the target that uses one or more of these subsystems will be able to "talk" to the DeviceMate unit.

Dynamic C v 7.10 or later is needed to configure, compile and load **devmate.c** onto the DeviceMate. In the documentation for Dynamic C the term "target" is used to refer to the controller board that is connected to Dynamic C via the programming cable. When "target" is used by Dynamic C it is limited to that definition. When using Dynamic C, any menus, dialog boxes, or other messages that use the term "target" are (almost) NEVER referring to a device that is serially connected to a DeviceMate unit. Dynamic C is (almost) ALWAYS referring to the controller board that it is connected to it via the programming cable. The exception is when using the remote program download feature. After the Ethernet loader is running on the DeviceMate unit, Dynamic C will recognize the device serially connected to the DeviceMate as the target.

In the documentation for a DeviceMate system (meaning there are 2 controller boards), the term "target" always refers to the device that is connected serially to the DeviceMate.

4.1 Configuration of Subsystems

A subsystem is enabled by an application by including a statement such as the following:

```
#define USE_TC_SMTP
```

in the code.

Subsystem Name	Request for Service Macro	Libraries for DeviceMate unit
E-mail	**USE_TC_SMTP**	**targetproc_tcp.lib**
File System	**USE_TC_FS**	**targetproc_fs.lib**
Message Logging	**USE_TC_LOG**	**targetproc_log.lib**
Remote Program Download	**USE_TC_LOADER**	**targetproc_loader.lib**
TCP/IP	**USE_TC_TCPIP**	**targetproc_tcp.lib**
Watchdogs	**USE_TC_WD**	**targetproc_wd.lib**
Web page Variables	**USE_TC_VAR**	**targetproc_var.lib**

4.1.1 Library Support

The request for service macros are located in **tc_conf.lib**. The statement

```
#use "tc_conf.lib"
```

must appear in an application program running on the DeviceMate unit that wishes to use one or more of the subsystems. This statement must follow any **USE_TC_*** defines.

Other Dynamic C libraries are required for several of the subsystems. An application that runs a web server must include the following statements:

```
#use dcrtcp.lib
#use http.lib
```

An application that uses the serial-based console must include:

```
#use dcrtcp.lib
#use zconsole.lib
```

And any application that uses the Dynamic C file system must include the statement:

```
#use fs2.lib
```

4.1.2 Function Chains

Applications running on the DeviceMate unit must call **targetproc_init()**. This function initializes the serial ISR, then it calls two function chains that initialize everything else.

To drive the software, an application running on the DeviceMate unit must repeatedly call **targetproc_tick()**. This function activates the function chain that calls tick functions for each of the subsystems.

4.1.3 Configuration Macros Common to all Subsystems

TC_I_AM_DEVMATE

This macro must be defined in an application running on the DeviceMate unit.

4.2 Sample Program Code

The following program will accept variable updates from the target. The sample program, **var.c**, running on the target will update the variables referenced in the web page **devmate_var.shtml**. Notice that this web page was imported via the **#import** directive. In an actual application **devmate_var.shtml** would probably be loaded from the target via the file system subsystem.

Program Name: devmate.c

```
#define TC_I_AM_DEVMATE
#define MY_IP_ADDRESS "10.10.6.100"
#define MY_NETMASK "255.255.255.0"
#define MY_GATEWAY "10.10.6.1"
#define MY_NAMESERVER "10.10.6.1"

#define SSPEC_MAXSPEC 10
#define TARGETPROC_VAR_BUFFERSIZE 1024
#define MAX_TCP_SOCKET_BUFFERS 8
#define MAX_UDP_SOCKET_BUFFERS 2
#define LOG_USE_XMEM
#define LOG_XMEM_SIZE 500

#memmap xmem

#define USE_TC_VAR
#define USE_TC_TCPIP
#define USE_TC_WD
#define USE_TC_LOG
#use "tc_conf.lib"

#use "dcrtcp.lib"
#use "http.lib"
#ximport "samples\dmunit\pages\devmate_var.shtml" var_html
const HttpType http_types[] ={
   { ".shtml", "text/html", shtml_handler}, // ssi
   { ".html", "text/html", NULL},           // html
   { ".cgi", "", NULL},                     // cgi
   { ".gif", "image/gif", NULL}
};

const HttpSpec http_flashspec[] ={
   { HTTPSPEC_FILE,"/var.shtml", var_html, NULL, 0, NULL, NULL}
};
void main(void){
   LogEntry loginfo;
   int status;
   char buf[200];

   targetproc_init();
   sock_init();
   http_init();
   tcp_reserveport(80);
#define LOG_TEST_STRING "~~~{ Started test run. }~~~"
   status = log_put(LOG_MAKEPRI(2,LOG_INFO), 0,
     LOG_TEST_STRING, strlen(LOG_TEST_STRING));

   if (status != 0)
      printf("Failed to add 1st message: %d\n", status);

   for (;;) {
      targetproc_tick();
      http_handler();
   }
}
```

4.3 Subsystem Distinctions

Each subsystem has a set of requirements apart from those that are shared by all of the subsystems.

4.3.1 TCP/IP Subsystem Configuration

A TCP/IP stack runs on the DeviceMate unit when **USE_TC_TCPIP** is defined. That means that the entire Dynamic C TCP/IP protocol suite is available. For more information on the protocols, see the *Dynamic C TCP/IP User's Manual*.

MY_IP_ADDRESS
MY_NETMASK
MY_GATEWAY
MY_NAMESERVER

Network address information is required. Modify these macros to match your system.

TARGETPROC_TCP_MAXRESOLVE

Defaults to 4. Specifies maximum number of concurrent DNS resolve requests that may be outstanding at any time. This macro may be defined to zero to disable the DNS resolver.

TARGETPROC_TCP_MAXSOCK

Defaults to 6. Maximum number of sockets (implemented as XTC "channels") that are allowed. Generally, this limits the number of sockets on the target since the target is assumed to be more resource-constrained than the DeviceMate. The actual maximum is determined by the minumum value configured on the target or DeviceMate processors. Note that this value does not include the so-called "control" channel (channel 0).

4.3.2 Web Page Variables Subsystem Configuration

Applications using the variables subsystem may run an HTTP server on the DeviceMate. This allows remote viewing of the variables.

MY_IP_ADDRESS
MY_NETMASK
MY_GATEWAY

Network address information is required. Modify these macros to match your system.

SSPEC_MAXSPEC

This limits the total number of variables that can be on the DeviceMate, even if **DEVMATE_VAR_MAXVARS** on the target processor is larger. Note that files are also included in **SSPEC_MAXSPEC**. It defaults to 10.

TARGETPROC_VAR_BUFFERSIZE

This defines the size in bytes of the xmem buffer in which to store the variables. It defaults to 1024.

4.3.2.1 HTML Files

To include an HTML file in the web server, add it to the **http_flashspec** array.

```
const HttpSpec http_flashspec[] =
{
   {HTTPSPEC_FILE, "/", index_html, NULL, 0, NULL, NULL},
   {HTTPSPEC_FILE, "/index.html", index_html, NULL, 0, NULL, NULL}
};
```

For more information on running a web server, please see the *Dynamic C TCP/IP User's Manual*.

4.3.3 File System Subsystem Configuration

The file system subsystem uses the Dynamic C filesystem mk 2 (FS2). To set up FS2 on the DeviceMate unit, the first thing to do is determine your file system needs. FS2 allows flexibility regarding the file system structure and placement. The choices that are available depend on the hardware being used and what the system is doing. FS2 can be in RAM, the first Flash, the second Flash (if one is available) or a combination of these. Further partitioning of these storage devices may be done.

Please read the chapter, "Flash File System" in the *Dynamic C User's Manual*. There you will find all the necessary information for setting up FS2. You may need to edit **Bios\RabbitBios.c** to make room for the file system.

After determining your file system needs, the second step is configuration of both the file system subsystem and FS2. This is done using the configuration macros and functions in the next two sections.

4.3.3.1 FS2 Configuration Macros

The **FS2_*** macros are from the Dynamic C library **fs2.lib**.

CONSOLE_TARGETPROC_FS_BACKUP

This is a convenience macro for the **console_backup[]** array (see the documentation for **zconsole.lib**). This macro will include support for backing up the table that holds the file name to file number mapping information, aka the spec table.

CONSOLE_TARGETPROC_FS_WITH_HTTP_BACKUP

This is another convenience macro for the **console_backup[]** array. This one includes support for backing up information for the HTTP server (that is, supporting the GET, PUT, GETV, PUTV, CREATEV commands of the console). If this is used in the **console_backup[]** array, then **CONSOLE_HTTP_BACKUP** should not be used (this includes everything in **CONSOLE_HTTP_BACKUP**).

FS2_USE_PROGRAM_FLASH

Default: 0, do not use first (program) flash. To use the first flash, **#define** this macro to be the number of kilobytes to allocate to FS2. The minimum of **FS2_USE_PROGRAM_FLASH** and **XMEM_RESERVE_SIZE** will be used (defined in **Bios\RabbitBios.c**).

FS_MAX_DEVICES

This macro defines the maximum number of physical media. If it is not defined in the program code, it will default to 1, 2, or 3, depending on the values of **FS2_USE_PROGRAM_FLASH**, **XMEM_RESERVE_SIZE** and **FS2_RAM_RESERVE**.

- **XMEM_RESERVE_SIZE** is defined in the BIOS and determines the amount of space reserved in the first flash that will not be used for xmem code space. This is defined to 0x0000.
- **FS2_RAM_RESERVE** is also defined in the BIOS and determines the amount of space used for FS2 in RAM. If some battery-backed RAM is to be used by FS2, then this macro must be modified to specify the amount of RAM to reserve. This should be a multiple of 4096 bytes. The memory is reserved near the top of RAM. Note that this RAM will be reserved whether or not the application actually uses FS2.

FS_MAX_FILES

This macro is used to specify the maximum number of files that are allowed to coexist in the entire file system. Most applications will have a fixed number of files defined, so this parameter can be set to that number to avoid wasting root data memory. The default is 12 files. The maximum value for this parameter is 255.

FS_MAX_LX

This macro defines the maximum logical extents. You must increase **FS_MAX_LX** by 1 for each new partition your application creates. If this is not defined in the program code it will default to **FS_MAX_DEVICES**.

MY_IP_ADDRESS
MY_NETMASK

If network address information is required, modify these macros to match your system.

TARGETPROC_FS_BACKUP_FILE1

File number used for the first backup file for the spec table. It defaults to 254. The spec table holds the mapping of file names to file numbers.

TARGETPROC_FS_BACKUP_FILE2

File number used for the second backup file for the spec table. It defaults to 255. Two backup files are used for the spec table so that the file name/number mapping is preserved even if the power cycles while data is being saved.

TARGETPROC_FS_ENABLE_BACKUP

This macro enables the backing up of the spec table.

TARGETPROC_FS_ENABLE_ZCONSOLE

This macro will allow the console to backup the spec table (the mapping from file name to file number). Enabling this means that the user should not use **TARGETPROC_FS_BACKUP_FILE1**, **TARGETPROC_FS_BACKUP_FILE2**, or **TARGETPROC_FS_USEBACKUPLX**, since **zconsole.lib** handles the details of backing up the spec table.

TARGETPROC_FS_USEBACKUPLX

This macro indicates the logical extent (LX) to use for backing up the spec table. The backup logical extent can be separate from the file logical extent, so that the user can always guarantee that enough space is available for backing up the spec table.

TARGETPROC_FS_USELX

This macro identifies the logical extent in FS2 that the target can access. This must be defined. It is used in the following way in the application code:

```
FSLXnum fs_ext;                          // Used to store a number that
                                         // identifies a logical extent in FS2.
#define TARGETPROC_FS_USELX (fs_ext)     // The variable fs_ext will be
                                         // filled in later.
```

The LX number of interest is determined at runtime, so have a statement similar to the following one in **main()**:

```
fs_ext = fs_get_flash_lx();
```

4.3.3.2 Configuration Functions

All of the FS2 API is available to an application running on the DeviceMate unit. Some of the functions are listed below. Full function descriptions for the FS2 API are available through the Help menu in Dynamic C and in the *Dynamic C User's Manua*l.

- **fs_init()** - Initialize the Dynamic C file system.
- **fs_get_lx()** - Return the current logical extent (LX) number for file creation.
- **fs_get_flash_lx()** - Returns the LX number for the 2nd flash device if it is available. If it is not available the LX number for the program flash is returned if space was set aside in the program flash for the FS2. (This is done with **FS2_USE_PROGRAM_FLASH** and **XMEM_RESERVE_SIZE**.)
- **fs_get_ram_lx()** - Returns the LX number for the RAM file system if space has been allocated for this use in the BIOS.
- **fs_get_other_lx()** - Returns the LX number of the non-preferred flash device. This is the program flash if it has been configured for use by the file system.
- **lx_format()** - Format a specified logical extent.

4.3.3.3 Backup Files for Spec Table

The following two functions are intended for use by an application running on the DeviceMate unit. Look at **`samples\dmunit\devmate_fs.c`** to see how the spec table is backed up using FS2 partitioning capabilities and **`targetproc_fs_backup_bytes()`**.

`targetproc_fs_backup_loaded`

```
int targetproc_fs_backup_loaded(void);
```

DESCRIPTION

Indicates if the backup spec table (that is, the index of file names) has been successfully loaded. This function should be called after **`targetproc_init()`** to give the filesystem subsystem an opportunity to load the spec table. If it returns zero, then the user code may want to take some action, such as reformatting the FS2. Note that this function will always return zero if **`TARGETPROC_FS_ENABLE_BACKUP`** has not been defined.

RETURN VALUE

`0`: Error, the spec table was not successfully loaded.
`1`: The spec table was successfully loaded.

LIBRARY

`targetproc_fs.lib`

`targetproc_fs_backup_bytes`

```
int targetproc_fs_backup_bytes(void);
```

DESCRIPTION

Returns the number of bytes that the file system subsystem needs for a spec table backup file (i.e., the table that maps file names to numbers). There are two backup files, so space must be reserved in the file system for both of them.

RETURN VALUE

≥**`0`**: Number of bytes in a spec table backup file.
`-1`: **`TARGETPROC_FS_ENABLE_BACKUP`** is not defined.

LIBRARY

`targetproc_fs.lib`

4.3.4 Message Logging Subsystem

The message logging subsystem on the DeviceMate unit acts as an interface between the target (over XTC) and the logging back-end implemented by **log.lib**. Messages may be stored in the file system (FS2), in an xmem buffer, or sent to the stdio window for debugging.

If FS2 is used, it must be set up on the DeviceMate unit prior to running an application on the target that makes use of it. For information on setting up FS2, please see the *Dynamic C User's Manual*.

4.3.4.1 Storage of Log Entries

The API function to write a logging message, **log_put()**, is called with a facility/priority code as its first parameter. This code is mapped to a storage destination (or destinations) via the configuration macro, **LOG_MAP(facpri)**. From zero to 4 destinations may be specified for any facility/priotity code.

The storage destination is comprised of two components: the destination class and a stream number. The class is specified by the 2 most significant bits of the 8-bit destination number, and the stream is the lower 6 bits. The storage destinations are defined as follows:

Table 1. Storage Destination Macros

Name of Macro	Value of Macro
LOG_DEST_NONE	0x00
LOG_DEST_STDOUT	0x01
LOG_DEST_FS2	0x40
LOG_DEST_XMEM	0xC0

> **Note:** The logging message will only print to the stdio window if the format specifier for the message is zero, meaning ASCII. The format specifier is communicated by the application running on the target as the high 8 bits of the first parameter in a call to **devmate_log_put()**.

Some storage destinations do not have streams, such as the xmem buffer and stdout. Currently, only the file system storage option uses streams; up to 64 streams are allowed. An FS2 stream is the file in which to store the log entries. The log entries may be stored in multiple files, but by default there is only one FS2 stream defined. This can be changed by **LOG_FS2_MAXSTRM**.

The facility/priority code used in the storage destination mapping is communicated by the application running on the target as the low 8 bits of the first parameter in a call to **devmate_log_put()**.

4.3.4.2 Configuration Macros

There are numerous macros associated with the the message logging subsystem. Most are in **log.lib**. The defaults are suitable for an initial test configuration.

LOG_USE_FS2
LOG_USE_XMEM

One or both of these macros may be defined to allow the specified storage class to be used as a logging destination. Both of these destination classes are "retrievable" in that the stored entries may be retrieved at runtime using **log_seek()**, **log_next()** and **log_prev()**. If you define **LOG_USE_FS2**, then the **fs2.lib** library is automatically included.

You must initialize the filesystem before calling any logging functions, since the logging subsystem does not do this!

If you do not define **LOG_USE_FS2**, then the filesystem library is not included, and you do not have to initialize it (unless your application uses it independently).

If you define **LOG_USE_XMEM**, then a single xmem buffer is allocated for storing log entries. If FS2 is not also used, this has the advantage of being faster and using much less root code space. The disadvantage is that the log entries will be lost when the application boots up, and obviously there will be less xmem space for the application.

LOG_FS2_MAXSTRM

Define the maximum number of FS2 streams. Default 1.

LOG_FS2_FILENO(strm)

Define the FS2 file number (0-255) to use for a given stream. Defaults to 123 independent of stream (i.e., every stream maps to file number 123).

LOG_FS2_DATALX(strm)
LOG_FS2_METALX(strm)

Define FS2 logical extent numbers for given stream. Defaults to the preferred flash extent, independent of stream.

LOG_FS2_SIZE (strm)

Define the FS2 stream size in bytes. When the stream exceeds this amount, either the oldest messages will be discarded (if circular) or the new message will not be added (non-circular). The actual file size may exceed the specified value by up to 256 bytes. The default is 8000 bytes. By carefully specifying this value, you can ensure that the filesystem capacity will not be exceeded because of an unexpected number of log messages.

LOG_MAP(facpri)

Map from facility/priority to destination class and stream. The result should be an unsigned long, which allows up to 4 dest/streams to be specified. Each dest/stream is a single byte, with the 2 MSBs specifying the destination class, and the 6 LSBs specifying the stream number within that class. A zero byte is special, it means "no destination."

The mapping must be constant—a particular input always gives the same output. The mapping must not be a function of anything other than the input parameter.

In general, this macro could be overridden to an arbitrarily complex function of the **facpri**, however the default should be sufficient for most purposes.

Table 2. Default destination mapping

Facility	Priority	Destination
0	any	The first of FS2 or XMEM which is configured. If neither is configured, then STDOUT. Stream 0.
1	any	STDOUT (intended for debugging).
2	any	Same as for facility zero, plus STDOUT.
3	any	Reserved for future use.
4-31	any	Ignored.

These defaults are implemented by the function **_log_default_map()**. You can change this function rather than changing the macro to achieve the same result.

If any of the returned destinations are invalid (e.g. stream number higher than the maximum configured for that class), then the destination will be deemed to be "nowhere" —no error is returned in this case. If more than one of the four possible destinations is the same, then the same message will be logged multiple times to that destination.

LOG_XMEM_SIZE

Define to the xmem buffer size. This is similar to the meaning of **LOG_FS2_SIZE**, except that the size is never exceeded.

LOG_FS2_CIRCULAR (strm)

Define to 1 if the specified FS2 stream is "circular" i.e. when full, the oldest entries are automatically deleted to make room for the new entry. If defined to zero, then the message logging subsystem will refuse to add new entries when the stream becomes full. When a non-circular log is full, it can only be re-used by calling **log_clean()** on it. A circular log would not normally ever need cleaning. Defaults to 1.

LOG_XMEM_CIRCULAR

Define to 0 or 1 to make the xmem buffer log non-circular (0) or circular (1). See **LOG_FS2_CIRCULAR** above for details. Defaults to 1.

FS2_USE_PROGRAM_FLASH

Set aside this many kilobytes in the program flash. You will also need to edit **bios\rabbitbios.c** to set an appropriate value to **XMEM_RESERVE_SIZE**. The definition here may be larger or smaller—the minimum of the two specifications will be used.

4.3.4.3 Function Reference

These API functions are not exclusively used by applications running on a DeviceMate unit. They are from the back-end logging utility available in Dynamic C v 7.10 (or later).

log_clean

```
int log_clean( LogDest ld );
```

DEFINITION

Reset only the specified destination class and stream (encoded as a **LogDest** value). This is only applicable to filesystem or XMEM destinations since they are locally persistent storage. XMEM is automatically cleaned at start-up time, since it is not assumed to be non-volatile. If this operation is not applicable, zero is returned with no further action.

PARAMETERS

ld	Destination class and stream. Use one of the constants **LOG_DEST_FS2** or **LOG_DEST_XMEM**, then OR in the stream number (0-63).

RETURN VALUE

0: Success
-2: Stream out of range for the storage class specified

LIBRARY

log.lib

log_close

```
int log_close( LogDestClass ldc );
```

DEFINITION

Close the specified class, enumerating all streams. If the destination class is already closed, returns success.

PARAMETERS

ldc	Destination class. Use one of the constants **LOG_DEST_FS2**, **LOG_DEST_XMEM**, or **LOG_DEST_ALL**. The latter case closes all open destinations.

RETURN VALUE

0: Success

LIBRARY

log.lib

log_condition

```
int log_condition(LogDest ldst);
```

DEFINITION

Return the state of the specified log destination.

PARAMETERS

ldst	Destination class and stream. Use one of the constants **LOG_DEST_FS2** or **LOG_DEST_XMEM**, then OR in the stream number (0-63).

RETURN VALUE

0: Destination not open
1: Destination OK
2: Destination reached limit of its space quota
-1: Error in destination.
-2: Destination class or stream is not configured

LIBRARY

log.lib

log_format

```
char * log_format(LogEntry * le, char * buffer, int length, int
   pfx);
```

DEFINITION

Given the log entry returned by **log_next()** or **log_prev()**, format the entry as an ASCII string. The string is constructed in Unix "syslog" format:

```
<%d>%.15s %.8s[%d]: %s
```

where the substitutions are:

%d: facility/priority as decimal number (0-255)
%.15s: date/time as "Mon dd hh:mm:ss"
%s: process name - taken from **LOG_UDP_PNAME(0)** if defined, else "" (empty).

%d: process ID, but the entry serial number is used instead.

%s: the log entry data.

A null terminator is always added at **buffer**[**length**-1], or at the end of the string if it fits in the buffer. If **pfx** is zero, then the above syslog prefix is not generated.

PARAMETERS

le	Log entry result from **log_next/log_prev()**.
buffer	Storage for result. Must be at least **length** bytes long.
length	Length of buffer. For the maximum sized log entry, the buffer should be 158 bytes. The minimum length must be greater than or equal to 43 (if **pfx** true) else 1. If a bad length is passed, the function returns without writing to **buffer**.
pfx	**0**: message text only; do not generate syslog prefix. **1**: prefix plus message text. **2**: prefix only (up to ']', then null terminator).

RETURN VALUE

Address of **buffer**, or **NULL** if bad length passed.

LIBRARY

log.lib

SEE ALSO

log_next, log_prev

log_map

```
uint32 log_map(LogFacPri lfp);
```

DEFINITION

Return the log destination class and stream, for a given facility/priority code. The result is up to 4 destinations packed into a longword. This function merely invokes the macro **LOG_MAP()**, which may be overridden by the application, but defaults to just the file-system.

PARAMETERS

lfp	Facility/priority code. This is a single-byte code specified whenever any log message is added. Facility is coded in the 5 MSBs, and priority in the 3 LSBs.

RETURN VALUE

Up to 4 destinations for a message of the specified facility and priority. Each byte in the resulting long word represents a destination/stream. A zero byte indicates no destination. If the result is all zeros, then a message of this type would be discarded.

LIBRARY

log.lib

log_next

```
int log_next(LogDest ldst, LogEntry * le);
```

DEFINITION

Retrieve next log entry. You must call **log_seek()** before calling this function the first time. Retrieval of stored log messages proceeds, for example, as follows:

```
log_seek(ldst, 0);      // seek to start
log_next(ldst, &L);     // get 1st entry
log_next(ldst, &L);     // get 2nd entry
log_prev(ldst, &L);     // get 2nd entry again
log_prev(ldst, &L);     // get 1st entry
log_prev(ldst, &L);     // returns -1
```

PARAMETERS

ldst — Destination class and stream. Use one of the constants **LOG_DEST_FS2** or **LOG_DEST_XMEM**, then OR in the stream number (0-63).

le — Storage for result.

RETURN VALUE

≥**0**: Length of log entry data.
-1: End of log or not open.
-2: Not a readable log destination class.

LIBRARY

log.lib

log_open

```
int log_open( LogDestClass ldc, int clean );
```

DEFINITION

Open the specified logging destination class. If necessary, this enumerates all possible streams within the class, opening them all (necessary only for FS2 class, since each file needs to be opened). Class **LOG_DEST_ALL** opens all configured classes.

If **clean** is true, then the destination will be erased before it is used, if that makes sense for the class.

PARAMETERS

ldc	Destination class: **LOG_DEST_FS2**, **LOG_DEST_XMEM** or **LOG_DEST_ALL**.
clean	Boolean, should the destination be erased before using?

RETURN VALUE

0: Success.
-1: Error, unknown destination class.

LIBRARY

log.lib

log_prev

```
int log_prev(LogDest ldst, LogEntry * le);
```

DEFINITION

Retrieve previous log entry. You must call **log_seek()** before calling this function the first time. Retrieval of stored log messages proceeds, for example, as follows:

```
log_seek(ldst, 1);  // seek to end
log_prev(ldst, &L); // get last entry
log_prev(ldst, &L); // get 2nd last entry
log_next(ldst, &L); // get 2nd last entry again
log_next(ldst, &L); // get last entry
log_next(ldst, &L); // returns -1
```

PARAMETERS

ldst	Destination class and stream. Use one of the constants **LOG_DEST_FS2** or **LOG_DEST_XMEM**, then OR in the stream number (0-63).
le	Storage for result.

RETURN VALUE

≥**0**: Length of log entry data.
-1: Start of log or not open.
-2: Not a readable log destination class.

LIBRARY

log.lib

SEE ALSO

log_seek, log_next

log_put

```
int log_put(LogFacPri ifp, uint8 fmt, char * data, int length);
```

DEFINITION

Add a log entry. The specified facility/priority (**ifp**) is mapped to the appropriate destination(s), as configured by the macros. If the destination exists, then the log entry is added. Otherwise, the entry is quietly ignored. If a destination is unable to fit the log entry, and the destination is configured as circular, then the first few entries may be deleted to make room. If this cannot be done, or an unrecoverable error occurs, then -2 is returned. For non-circular destinations, -2 is returned when the destination becomes full.

Since multiple log destinations can result from the given facility/priority, it can be difficult to determine which actual destination caused an error. You can use the **log_map()** function to determine the destinations, then check each destination's state using **log_condition()**.

PARAMETERS

ifp	Facility/priority code. Facility in 5 MSBs, priority in 3 LSBs.
fmt	Format code. 0 for ascii string, others user-defined.
data	Pointer to first byte of data to store.
length	Length of data. Must be between 0 and 115 (**LOG_MAX_MESSAGE**) inclusive.

RETURN VALUE

0: Success.

-1: Message too long (over 115 bytes).

-2: Unrecoverable error in destination. This return code usually means that the destination is unusable and further entries for that destination will probably meet the same fate. This can also mean that the destination has not been opened.

LIBRARY

log.lib

log_seek

```
int log_seek(LogDest ldst, int whence);
```

DEFINITION

Position log for readback. The next call to **log_next()** will return the first entry in the log (if **whence**=0), or **log_prev()** will return the last entry (if **whence**=1).

PARAMETERS

ldst	Destination class and stream. Use one of the constants **LOG_DEST_FS2** or **LOG_DEST_XMEM**, then OR in the stream number (0-63).
whence	Location to start reading back from. 0: first entry 1: last entry Other values are reserved.

RETURN VALUE

0: Success.
-1: Log empty.
-2: Unrecoverable error or not open.
-3: Not a seekable or configured log destination class.
-4: Invalid **whence** parameter.

LIBRARY

log.lib

SEE ALSO

log_next, log_prev

4.3.5 Remote Program Download

This feature is available for any Rabbit-based target. The DeviceMate unit will allow the download of a program to the target from a remote location. The remote program download feature is different from the other DeviceMate features because it works only with Rabbit-based targets and it does not require an application to be running on the target. The target may be a blank slate.

4.3.5.1 Setting Up the DeviceMate as a Conduit

1. Open up Dynamic C v 7.10 or later.
2. Open the file **samples\dmunit\loader.c** and change the IP address information to match the address of the DeviceMate unit.
3. Download **loader.c** to the DMU using the serial connection. Run the program. If **TARGETPROC_LOADER_DEBUG** is defined, a message will print to the stdio window letting you know **loader.c** is listening for a connection.
4. Using the **Options | Communications** dialog box, choose "Use TCP/IP Connection" and fill in the Network Address and Control Port information for the DeviceMate. If the DeviceMate is behind a firewall you will probably have to enter the proxy IP/Port numbers for the firewall instead. Click "OK."

From this point the DeviceMate is a conduit for communication between the remote location and the DeviceMate's target device. The DeviceMate reads STATUS and drives RESET, SMODE0 and SMODE1on the target through the serial interface.

Hit **Ctrl-Y** to reset the target. Now you can download a program to the target in the usual way.

4.3.5.2 Communication Between DeviceMate Unit and Target

Any program that is downloaded to the target through the DeviceMate must have the following code in it to force all communications to talk directly to the Ethernet Loader (**loader.c**) running on the DMU.

```
#define USE_TC_LOADER
#use "tc_conf.lib"
```

Both of these statements are in **loader.c**. The current software forces the DeviceMate to use its serial port B for downloading to the target's serial port A.

4.3.6 Watchdog Subsystem Configuration

The watchdog subsystem running on the DeviceMate unit accepts messages from the target to create, remove and hit watchdogs. If a watchdog is not hit within the amount of time specified in the call to **`devmate_wd_add()`**, the DeviceMate calls a function that will reset the target .

Developers working with nonRabbit targets must supply a pointer to a reset function for their device.

`TARGETPROC_WD_NUMTXBUF`

This determines the maximum number of transmit buffers on DeviceMate. Default is 1.

`TARGETPROC_WD_NUMRXBUF`

This determines the maximum number of receive buffers on DeviceMate. Default is 2.

`TARGETPROC_WD_MAXWD`

This determines the maximum number of watchdogs on DeviceMate. Default is 12.

`TARGETPROC_WD_RESETFUNCTION`

This is a pointer to a function that will reset the target. If it is not defined in the application code, it defaults to **`_targetproc_wd_resetfunction`**.

Chapter 5. Function Reference for Target Applications

5.1 TCP/IP Subsystem

devmate_ip_resolve

```
int devmate_ip_resolve(uint16 * request_id, char * remote_host,
   uint32 * ipaddr);
```

DESCRIPTION

Perform a DNS (Domain Name Server) lookup of an internet host name. The host name is resolved into a binary IP address. Since DNS lookups can be time-consuming, this function must be called several times to complete the process of looking up one host-name. Several hostnames may be resolved simultaneously if the appropriate state information is retained.

PARAMETERS

request_id Pointer to a "handle" that is used to identify the resolver request. This handle should be initialized to zero for the first call for a given host name. On return, the handle will be set to a unique identifier. On subsequent calls, the same handle must be passed until success or failure is declared.

remote_host The host name to resolve. This may be an internet host name, or a "dotted quad" (e.g. "10.10.6.2"). The same host name should be passed on all calls to this function (for a particular request id) until the name is resolved.

ipaddr Pointer to place to store the resolved IP address.

RETURN VALUE

0: Try again later.
1: Name successfully resolved.
-2: DeviceMate not initialized or no name resolver has been configured.
Other negative values: Error.

LIBRARY

```
dm_tcp.lib (Rabbit-based targets)
dm_tcp.c (nonRabbit-based targets)
```

devmate_sock_init

```
int devmate_sock_init(uint32 *ipaddress, uint32 *netmask,
   uint32 *gateway, uint32 *nameserver, char * classid, char *
   clientid);
```

DESCRIPTION

Establish communication link to the DeviceMate. If a link is already established, then any open sockets are aborted and the link is re-established.

PARAMETERS

All parameters are reserved for future use. At present, **NULL** should be given for each parameter.

RETURN VALUE

1

LIBRARY

dm_tcp.lib (Rabbit-based targets)
dm_tcp.c (nonRabbit-based targets)

SEE ALSO

devmate_tcp_status

devmate_tcp_abort

```
int devmate_tcp_abort(int socket);
```

DESCRIPTION

Close a socket forcibly. This is used to signal the peer that the stream is in error. The peer may or may not actually receive the abort indicator; however, the library considers the socket to be closed. After calling this function, **devmate_tcp_error()** will return a non-zero code.

Note that sockets are sometimes forcibly closed by other library functions. For example, if **devmate_sock_init()** is called, then any open sockets will be aborted.

This function is also used to close UDP sockets. In this case, there is no special meaning to "abort," since UDP sockets are not connection-oriented so either peer may close its socket at any time.

PARAMETERS

socket The socket number as used by **devmate_tcp_open()**, **devmate_tcp_listen()** and **devmate_udp_open()**.

RETURN VALUE

0

LIBRARY

```
dm_tcp.lib (Rabbit-based targets)
dm_tcp.c (nonRabbit-based targets)
```

SEE ALSO

```
devmate_tcp_close
```

devmate_tcp_maxsocket

```
int devmate_tcp_maxsocket(void);
```

DESCRIPTION

Return the maximum allowable socket number. This is the minimum of the configured value for the target and DeviceMate. The value is only available after calling **devmate_sock_init()**.

RETURN VALUE

0: Value not yet available.

>0: Maximum socket number. Valid socket numbers on future calls to the TCP/IP subsystem API must fall between 1 and this number inclusive.

LIBRARY

```
dm_tcp.lib (Rabbit-based targets)
dm_tcp.c (nonRabbit-based targets)
```

SEE ALSO

devmate_sock_init, devmate_tcp_open

devmate_tcp_close

```
int devmate_tcp_close(int socket);
```

DESCRIPTION

Close a socket gracefully. Calling this function indicates that no more data will be written. The peer is notified. Data may be read until the peer performs a corresponding close, or the socket is forcibly closed using **devmate_tcp_abort()**. After calling this function, **devmate_tcp_writable()** returns 0, and **devmate_tcp_readable()** may or may not return 0.

This function is a macro alias of **devmate_tcp_shutdown()**. It is only used for TCP sockets.

PARAMETERS

socket The socket number as used by devmate_tcp_open/listen()

RETURN VALUE

0: Closed successfully.
1: Socket not open, or is in the process of closing.

LIBRARY

```
dm_tcp.lib (Rabbit-based targets)
dm_tcp.c (nonRabbit-based targets)
```

SEE ALSO

```
devmate_tcp_abort, devmate_udp_close
```

devmate_tcp_error

```
int devmate_tcp_error(int socket);
```

DESCRIPTION

Check the socket to see if any error has occurred. Errors may occur because of programming error, link problems with the DeviceMate, or errors on the actual TCP/IP connection.

PARAMETERS

socket The socket number as used by devmate_tcp_open/listen() or **devmate_udp_open()**

RETURN VALUE

0: No error has occurred.

!0: Error occurred; the socket may be immediately re-opened. The error codes are not officially defined, but may be logged for debugging purposes.

LIBRARY

dm_tcp.lib (Rabbit-based targets)
dm_tcp.c (nonRabbit-based targets)

SEE ALSO

devmate_tcp_isestablished, devmate_tcp_readable, devmate_tcp_writable, devmate_tcp_isclosed

devmate_tcp_fastread

```
int devmate_tcp_fastread(int socket, void* buffer, uint16
   length);
```

DESCRIPTION

Read up to **length** bytes of data from the socket receive buffer into the given buffer. In general, less than **length** bytes may be moved. In this case, the caller must recall this function to deliver the remainder of data. This is typically performed via a loop that keeps track of the total data amount to read and the amount that has been successfully read. Applications that can afford to wait until all data are available can spin on this function.

If **devmate_tcp_readable()** returns N for a given socket, then a request for less than or equal (N-1) bytes is guaranteed to return the full amount requested.

PARAMETERS

socket	The socket number as used by devmate_tcp_open/listen()
buffer	Pointer to the first byte of the local buffer
length	Desired length of data to read

RETURN VALUE

-1: Error occurred; socket is not readable.
0: No data is available for reading.
1..(length-1): Data was partially read.
length: Data was completely read.

LIBRARY

dm_tcp.lib (Rabbit-based targets)
dm_tcp.c (nonRabbit-based targets)

SEE ALSO

devmate_tcp_readable, devmate_tcp_fastwrite,
devmate_tcp_preread

devmate_tcp_fastwrite

```
int devmate_tcp_fastwrite(int socket, void* buffer, uint16
   length);
```

DESCRIPTION

Write up to **length** bytes of data from **buffer** into the socket transmit buffer. In general, less than **length** bytes may be buffered. In this case, the caller must recall this function to deliver the remainder of data. Typically, this is performed via a loop that keeps track of the total data amount to transmit and the amount that has been successfully buffered. Applications that can afford to wait until all data is buffered can spin on this function. Once the data is buffered, then the underlying library will ensure that it is transmitted to the DeviceMate, so long as the application keeps calling the **devmate_tick()** function.

If **devmate_tcp_writable()** returns N for a given socket, then a request for less than or equal (N-1) bytes is guaranteed to buffer the full amount requested.

PARAMETERS

socket	The socket number as used by devmate_tcp_open/listen().
buffer	Pointer to the first byte to write to the buffer.
length	Desired length of data to write.

RETURN VALUE

-1: Error occurred; the socket is not writable.
0: No data could be written because the transmit buffer is full.
1..(length-1): Data was partially buffered.
length: Data was completely buffered.

LIBRARY

dm_tcp.lib (Rabbit-based targets)
dm_tcp.c (nonRabbit-based targets)

SEE ALSO

devmate_tcp_writable, devmate_tcp_fastread

devmate_tcp_isclosed

```
int devmate_tcp_isclosed(int socket);
```

DESCRIPTION

Check the socket to see if a TCP (stream) connection is currently closed. It is necessary to call this function on a socket that has been closed using **devmate_tcp_close()** since the socket will not actually close until the peer has finished transmitting data. After this function returns 1, then **devmate_tcp_error()** should also be called to ensure that the connection closed normally i.e. all data was transferred successfully.

PARAMETERS

socket The socket number as used by devmate_tcp_open/listen()

RETURN VALUE

0: Connection currently established.
1: Socket completely closed, and may be immediately re-opened.

LIBRARY

```
dm_tcp.lib (Rabbit-based targets)
dm_tcp.c (nonRabbit-based targets)
```

SEE ALSO

```
devmate_tcp_close, devmate_tcp_readable,
devmate_tcp_writable, devmate_tcp_abort, devmate_tcp_error
```

devmate_tcp_isestablished

```
int devmate_tcp_isestablished(int socket);
```

DESCRIPTION

Check the socket to see if a TCP (stream) connection is currently, or was recently, established. After passively opening a socket with **devmate_tcp_listen()**, this function is called to determine if a connection is, or was, made. This function may also be used for actively opened TCP sockets to check that the connection is still open.

PARAMETERS

socket The socket number as used by devmate_tcp_open/listen()

RETURN VALUE

0: No connection is currently established.

1: Connection is currently established, or was established but immediately closed with no transfer of data. It is possible to read zero or more bytes of data.

-1: Connection was established but immediately aborted. **devmate_tcp_error()** will return a non-zero code for further diagnostic information.

LIBRARY

```
dm_tcp.lib (Rabbit-based targets)
dm_tcp.c (nonRabbit-based targets)
```

SEE ALSO

```
devmate_tcp_listen, devmate_tcp_readable,
devmate_tcp_writable, devmate_tcp_isclosed, devmate_tcp_error
```

devmate_tcp_listen

```
int devmate_tcp_listen(uint8 socket, uint16 local_port, char *
   remote_host, uint16 remote_port);
```

DESCRIPTION

Open a TCP socket via the DeviceMate. This is implemented as a macro which invokes **devmate_tcp_start()**. The socket is opened passively, meaning that it waits for active connections from other host(s). The application must periodically call **devmate_tcp_isestablished()** to check whether any connection has been made.

PARAMETERS

socket — Socket number to use. This may be a number between 1 and **TCT_MAXTCPSOCK** inclusive. The application must select a socket number that is not already open or in use. This may also be 0, in which case an unused socket number will be selected (and returned as a positive number).

local_port — A local port number to use. This must be specified.

remote_host — This is normally **NULL**, to allow any remote host to establish a connection. If not **NULL**, it specifies the only host allowed to establish a connection with this socket.

remote_port — This is normally zero, which allows connections from any port on the remote host. If non-zero, then the remote host must connect from this port number.

RETURN VALUE

0: The open request could not be performed; try again later. This may be because the DeviceMate is not yet fully linked, or because there is temporary buffer shortage.

>0: Opened successfully. The return value is the same socket number as passed in the first parameter or the assigned socket number if the parameter was 0.

-1: Error; host lookup failure or DeviceMate not linked.
-2: Error; DeviceMate not linked because **devmate_sock_init()** was not called.
-3: Error; requested socket number out of range.
-4: Error; requested socket number already in use.
-5: Error; requested socket number zero, but none free.

LIBRARY

```
dm_tcp.lib (Rabbit-based targets)
dm_tcp.c (nonRabbit-based targets)
```

SEE ALSO

```
devmate_tcp_open, devmate_udp_open, devmate_tcp_close
```

devmate_tcp_open

```
int devmate_tcp_open(uint8 socket, uint16 local_port, char *
   remote_host, uint16 remote_port);
```

DESCRIPTION

Open a TCP socket via the DeviceMate. This is implemented as a macro which invokes **devmate_tcp_start()**.

PARAMETERS

socket	Socket number to use. This may be a number between 1 and **devmate_tcp_maxsocket()** inclusive. The application must select a socket number that is not already open or in use.This may also be 0, in which case an unused socket number will be selected (and returned as a positive number).
local_port	A local port number to use. If zero, then a port number will be assigned automatically.
remote_host	The name of a remote host to connect to. This may be a dotted quad (e.g. "10.10.6.2") or a host name (e.g. "ftp.zworld.com" or "server_host"). If a host name is given, then the DeviceMate will look up the host's IP address using DNS.
remote_port	The port on the remote host to connect to.

RETURN VALUE

0: The open request could not be performed; try again later. This may be because the DeviceMate is not yet fully linked, or because there is temporary buffer shortage.

>0: Opened successfully. The return value is the same socket number as passed in the first parameter or the assigned socket number if the parameter was 0.

-1: Error; remote host not available.
-2: Error; DeviceMate not linked because **devmate_sock_init()** was not called.
-3: Error; requested socket number out of range.
-4: Error; requested socket number already in use.
-5: Error; requested socket number zero, but none free.

LIBRARY

```
dm_tcp.lib (Rabbit-based targets)
dm_tcp.c (nonRabbit-based targets)
```

SEE ALSO

```
devmate_tcp_listen, devmate_udp_open, devmate_tcp_close
```

devmate_tcp_preread

```
int devmate_tcp_preread(int socket, void* buffer, uint16
   length);
```

DESCRIPTION

Read up to length bytes of data from the socket receive buffer into the given buffer. In general, less than length bytes may be moved. Unlike **devmate_tcp_fastread()**, this function does not actually remove the data from the receive buffer. This function is used to give a "sneak preview" of the data in the buffer. Sequential calls to this function will return the same data each time, until **devmate_tcp_fastread()** is used to read and remove the data. Each sequential call to this function may return more data than the previous time, since there may be new data appended to the receive buffer.

If **devmate_tcp_readable()** returns N for a given socket, then a request for less than or equal (N-1) bytes is guaranteed to return the full amount requested.

PARAMETERS

socket	The socket number as used by devmate_tcp_open/listen()
buffer	Pointer to the first byte of the local buffer
length	Desired length of data to read

RETURN VALUE

-1: Error occurred; socket not readable.
0: No data available for reading.
1..(length-1): Data partially copied.
length: Data completely copied.

LIBRARY

dm_tcp.lib (Rabbit-based targets)
dm_tcp.c (nonRabbit-based targets)

SEE ALSO

devmate_tcp_readable, devmate_tcp_fastread

devmate_tcp_readable

```
uint16 devmate_tcp_readable(int socket);
```

DESCRIPTION

Return the number of data bytes which may be read from the socket receive buffer. The return value is encoded such that it can be interpreted as a boolean indicator of whether the socket is in a readable state, or as the actual number of bytes which can be read with one call to **devmate_tcp_fastread()**.

PARAMETERS

socket The socket number as used by devmate_tcp_open/listen().

RETURN VALUE

0: the socket is not readable e.g. it is not open, or encountered an error.

!0: the number of bytes which can be read, plus 1. E.g. if the return value is 5, then 4 bytes can be read immediately using **devmate_tcp_fastread()**. If the return value is 1, then the socket is readable but there is currently no data in the receive buffer.

LIBRARY

```
dm_tcp.lib (Rabbit-based targets)
dm_tcp.c (nonRabbit-based targets)
```

SEE ALSO

```
devmate_tcp_isestablished, devmate_tcp_fastread,
devmate_tcp_writable, devmate_tcp_isclosed
```

devmate_tcp_status

```
int devmate_tcp_status(void);
```

DESCRIPTION

Test communication link to the DeviceMate. The link status indicates whether TCP/IP functions can be accessed on the DeviceMate.

RETURN VALUE

0: The link is currently being established.

1: The link is successfully established.

-1: An error has occurred. Call **devmate_sock_init()** to re-establish the link. An error typically occurs because there is either no DeviceMate, or because the DeviceMate is not configured for TCP/IP support.

LIBRARY

dm_tcp.lib (Rabbit-based targets)
dm_tcp.c (nonRabbit-based targets)

SEE ALSO

devmate_sock_init

devmate_tcp_writable

```
uint16 devmate_tcp_writable(int socket);
```

DESCRIPTION

Return the number of data bytes that may be written to the socket transmit buffer. The return value is encoded such that it can be interpreted as a boolean indicator of whether the socket is in a writable state, or as the actual number of bytes which can be written with one call to **devmate_tcp_fastwrite()**.

PARAMETERS

socket The socket number as used by devmate_tcp_open/listen().

RETURN VALUE

0: The socket is not writable e.g. it is not open, or encountered an error.

!0: The number of bytes which can be written, plus 1. E.g. if the return value is 200, then 199 bytes can be written immediately using **devmate_tcp_fastwrite()**. If the return value is 1, then the socket is writable but there is currently no space for new data in the transmit buffer. The latter situation occurs when new data is being generated faster than it can be transmitted to the DeviceMate or remote host.

LIBRARY

```
dm_tcp.lib (Rabbit-based targets)
dm_tcp.c (nonRabbit-based targets)
```

SEE ALSO

```
devmate_tcp_isestablished, devmate_tcp_readable,
devmate_tcp_fastwrite, devmate_tcp_isclosed
```

devmate_udp_close

```
int devmate_udp_close(int socket);
```

DESCRIPTION

Close a UDP socket. This is actually a macro alias for **devmate_tcp_abort()**. UDP sockets are normally opened for the lifetime of the application; however, if it is desired to re-use the socket number taken by a UDP socket, then it is necessary to first close the UDP socket to release resources held for it on the target and DeviceMate processors.

PARAMETERS

socket The socket number as used by **devmate_udp_open()**.

RETURN VALUE

0

LIBRARY

dm_tcp.lib (Rabbit-based targets)
dm_tcp.c (nonRabbit-based targets)

SEE ALSO

devmate_tcp_close, devmate_tcp_abort

devmate_udp_open

```
int devmate_udp_open(uint8 socket, uint16 local_port, char *
   remote_host, uint16 remote_port);
```

DESCRIPTION

Open a UDP socket via the DeviceMate. This is implemented as a macro which invokes **devmate_tcp_start()**. UDP datagrams are delivered reliably (via an XTC channel) to the DeviceMate, which then sends the datagram using normal UDP which, by its nature, is an unreliable delivery service.

PARAMETERS

socket — Socket number to use. This must be a number between 1 and **devmate_tcp_maxsocket()** inclusive. The application must select a socket number that is not already open or in use. This may also be 0, in which case an unused socket number will be selected (and returned as a positive number).

local_port — A local port number to use. If zero, then a port number will be assigned automatically.

remote_host — The name of a remote host to connect to. This may be a dotted quad (e.g. "10.10.6.2") or a host name (e.g. "ftp.zworld.com" or "server_host"). If a host name is given, then the DeviceMate will look up the host's IP address using DNS. For UDP, this parameter may also be **NULL**. In this case, no host is selected as a default; **devmate_udp_sendto()** must be used to specify a remote host on a call-by-call basis. If a non-**NULL** host is specified, then that host is used as a default destination if **devmate_udp_send()** is used; however **devmate_udp_sendto()** may still be used to direct datagrams to other hosts.

remote_port — The port on the remote host to connect to. This may be zero if **devmate_udp_sendto()** is used consistently to specify the remote port on a call-by-call basis.

devmate_udp_open (continued)

RETURN VALUE

0: The open request could not be performed; try again later. This may be because the DeviceMate is not yet fully linked, or because there is temporary buffer shortage.

>0: Opened successfully. The return value is the same socket number as passed in the first parameter.

-1: Error; remote host not available.
-2: Error; DeviceMate not linked because devmate_sock_init has not been called.
-3: Error; requested socket number out of range.
-4: Error; requested socket number already in use.
-5: Error; requested socket number zero, but none free.

LIBRARY

```
dm_tcp.lib (Rabbit-based targets)
dm_tcp.c (nonRabbit-based targets)
```

SEE ALSO

```
devmate_tcp_listen, devmate_tcp_open, devmate_udp_close
```

devmate_udp_recvdata

```
int devmate_udp_recvdata(int socket, void* buffer, uint16
   max_length, uint16 recv);
```

DESCRIPTION

Receive datagram payload data. This function may be called after **devmate_udp_recvfrom()** indicates that a datagram has arrived. It should be called repeatedly (updating **recv**) until all required data has been read. It is not necessary to read all data before calling **devmate_udp_recvfrom()** again. When recalling, do not change any of the passed parameters except for **recv**.

PARAMETERS

socket	The socket number as used by **devmate_udp_open()**
buffer	Pointer to start of datagram buffer
max_length	Maximum length of the datagram buffer
recv	Number of bytes transferred by previous calls to this function. This should be zero for the first call, then should be the returned value from the previous call. This allows the datagram to be transferred piecemeal.

RETURN VALUE

-1: Error occurred.

≥**0**: The total number of bytes transferred so far. If this is less than the length originally returned by **devmate_udp_recvfrom()**, then this function may be called again (with this return value passed as the **recv** parameter) to get the remainder of the datagram.

LIBRARY

```
dm_tcp.lib (Rabbit-based targets)
dm_tcp.c (nonRabbit-based targets)
```

SEE ALSO

```
devmate_udp_sendto, devmate_udp_open, devmate_udp_recvfrom
```

devmate_udp_recvfrom

```
int devmate_udp_recvfrom(int socket, uint32* remote_host,
   uint16* remote_port);
```

DESCRIPTION

Receive notification of UDP datagram availability from the given socket. If a datagram is available, the data may be retrieved using one or more calls to **devmate_udp_recvdata()**. This two-phase receive process is a consequence of the reliable transport between the target and DeviceMate processors, and the possibly limited buffer space on the target.

This function only provides notification of the presence of a new datagram. Usually the data would be retrieved using **devmate_udp_recvdata()**. However, if this function is called again before reading the data (or partially reading it), then the rest of the previous datagram will be discarded. It is thus not necessary to call **devmate_udp_recvdata()** if the actual data is not important.

PARAMETERS

socket	The socket number as used by **devmate_udp_open()**.
remote_host	Pointer to storage for the IP address of the sender. This may be **NULL** if the sender's address is not required.
remote_port	Pointer to storage for the sender's port number. May be **NULL** if not required, however if **NULL** it is not possible to distinguish between reception of a zero-length datagram and no datagram at all.

RETURN VALUE

-1: Error occurred.

0: Datagram not available (try again later), or a zero-length datagram is available. These two cases are distinguished by examining the returned remote port: this will be zero if there is no datagram, or non-zero if there is a zero-length datagram.

≥**0**: The received datagram length.

LIBRARY

```
dm_tcp.lib (Rabbit-based targets)
dm_tcp.c (nonRabbit-based targets)
```

SEE ALSO

```
devmate_udp_sendto, devmate_udp_open, devmate_udp_recvdata
```

devmate_udp_sendto

```
int devmate_udp_sendto(int socket, void* buffer, uint16 length,
   uint16 trans, uint32 remote_host, uint16 remote_port);
```

DESCRIPTION

Transmit a UDP datagram to the specified remote host and port number. UDP datagrams are sent in two phases: the data is reliably transmitted to the DeviceMate, then the DeviceMate sends the entire datagram on the network. The initial phase may require sending the datagram in more than one chunk, if the local transmit buffer is not large enough to contain the entire datagram (and header). This requires that the application call this function repeatedly until the return value indicates that all data has been buffered. When re-calling, do not change any of the passed parameter values except for trans.

PARAMETERS

socket	The socket number as used by **devmate_udp_open()**.
buffer	Pointer to the start of the datagram to transmit
length	Total length of the datagram
trans	Zero on first call to transmit a new datagram. Subsequently, this parameter is the number of bytes that have been transferred so far; i.e. the return value from the previous call to this function.
remote_host	IP address of the remote host. This may be zero to use the host selected on the initial call to open the UDP socket. Otherwise, this must be a binary IP address either hard-coded or obtained from **devmate_ip_resolve()**.
remote_port	Remote UDP port number. May be zero to use the port selected on the initial call to open the UDP socket

RETURN VALUE

-1: Error occurred.

≥**0**: The total number of bytes that have been successfully transferred from this datagram. If this is less than **length**, then this return value should be provided to a subsequent call to this function to complete the transfer of this datagram. If equal to **length**, then the datagram has been fully transferred so that a new datagram may be transmitted.

LIBRARY

```
dm_tcp.lib (Rabbit-based targets)
dm_tcp.c (nonRabbit-based targets)
```

SEE ALSO

```
devmate_udp_send, devmate_udp_open, devmate_udp_recvfrom,
```

devmate_udp_recvdata

devmate_udp_send

```
int devmate_udp_send(int socket, void* buffer, uint16 length,
   uint16 trans);
```

DESCRIPTION

Transmit a UDP datagram to the remote host and port number which were specified on the original call to **devmate_udp_open()** for this socket. This is implemented as a macro which invokes **devmate_udp_sendto()** with zero remote host and port numbers. See the description of **devmate_udp_sendto()** for more details.

PARAMETERS

socket	The socket number as used by **devmate_udp_open()**
buffer	Pointer to the start of the datagram to transmit
length	Total length of the datagram
trans	Zero on first call to transmit a new datagram. Subsequently, this parameter is the number of bytes that have been transferred so far; i.e. the return value from the previous call to this function.

RETURN VALUE

-1: Error occurred.

≥**0**: Total number of bytes successfully transferred from this datagram.

LIBRARY

dm_tcp.lib (Rabbit-based targets)
dm_tcp.c (nonRabbit-based targets)

SEE ALSO

devmate_udp_sendto, devmate_udp_open, devmate_udp_recvfrom, devmate_udp_recvdata

5.2 E-Mail Subsystem

devmate_smtp_mailtick

```
int16 devmate_smtp_mailtick(void);
```

DESCRIPTION

Repetitively call this function until e-mail is completely sent. For a small message, this function will need to be called about 20 times to send the message. The number of times will vary depending on the latency of the connection to the mail server and the size of your message.

This function should be called repeatedly to iterate through the steps of the SMTP protocol after calling one of the sendmail functions.

RETURN VALUE

DM_SMTP_SUCCESS: E-mail sent successfully.
DM_SMTP_PENDING: E-mail not sent, call **devmate_smtp_mailtick()** again.
DM_SMTP_TIME: E-mail not sent within **DM_SMTP_TIMEOUT** seconds.
DM_SMTP_UNEXPECTED: Received an invalid response from SMTP server.

LIBRARY

```
dm_smtp.lib (Rabbit-based target)
dm_smtp.c (nonRabbit-based target)
```

SEE ALSO

devmate_smtp_sendmail, devmate_smtp_status

devmate_smtp_sendmail

```
void devmate_smtp_sendmail(char* to, char* from, char* subject,
   char* message);
```

DESCRIPTION

Begin sending an e-mail message. The sum of the string lengths of **to**, **from** and **subject** must be less than or equal to 224. You must call **devmate_smtp_mailtick()** to complete processing.

The parameter strings must be held constant until the **devmate_smtp_mailtick()** returns something other than **DM_SMTP_PENDING**.

PARAMETERS

to	String containing the e-mail address of the recipient, or a comma-delimited list of recipients (e.g. "nobody@nowhere.org" or "foo@bar.com,baz@bar.com").
from	String containing the e-mail address of the sender, or at least an address of a mailbox for the server to post back any error messages.
subject	A subject string
message	A string containing the message body. This string must NOT contain the byte sequence "\r\n.\r\n" (CRLF.CRLF), as this is used to mark the end of the e-mail, and will be appended to the e-mail automatically. The maximum length of this string is limited by the lesser of root data space, or 32k. For strict RFC compliance, lines should be delimited with "\r\n" sequences, not just "\n". For longer messages, see **devmate_smtp_sendmailxmem()**.

LIBRARY

```
dm_smtp.lib (Rabbit-based target)
dm_smtp.c (nonRabbit-based target)
```

SEE ALSO

devmate_smtp_mailtick, devmate_smtp_status, devmate_smtp_sendmailxmem

devmate_smtp_sendmailxmem

```
void devmate_smtp_sendmailxmem(char* to, char* from, char*
   subject, faraddr_t message, uint32 messagelen);
```

DESCRIPTION

Begin sending an e-mail message. The sum of the string lengths of to, from and subject must be less than or equal to 224 bytes. You must call **devmate_smtp_mailtick()** to complete processing.

This function is identical to **devmate_smtp_sendmail()** except that longer messages can be contained in extended ("far") memory.

PARAMETERS

to	String containing the e-mail address of the recipient, or a comma-delimited list of recipients (e.g. "nobody@nowhere.org" or "foo@bar.com,baz@bar.com").
from	String containing the e-mail address of the sender, or at least an address of a mailbox for the server to post back any error messages.
subject	Subject string
message	String containing the message. This string must NOT contain the byte sequence "\r\n.\r\n" (CRLF.CRLF), as this is used to mark the end of the e-mail, and will be appended to the e-mail automatically. The maximum length of this string is limited by the lesser of root data space, or 32k. This string resides in far memory, and does not have to be null-terminated since the length is specified in the next parameter.
messagelen	Number of bytes of message body

LIBRARY

```
dm_smtp.lib (Rabbit-based target)
dm_smtp.c (nonRabbit-based target)
```

SEE ALSO

devmate_smtp_mailtick, devmate_smtp_status, devmate_smtp_sendmail

devmate_smtp_setdomain

```
int devmate_smtp_setdomain(char* domain);
```

DESCRIPTION

Sets the SMTP domain. This value overrides **DEVMATE_SMTP_DOMAIN**. Note that this value should be set to the sender's domain name, since some SMTP servers may be configured to perform a reverse DNS query to verify the identity of the sender. If **DEVMATE_SMTP_DOMAIN** is defined to a suitable default (in tc_conf.lib/h) then there is no need to set the domain using this function.

PARAMETER

domain String containing the domain name e.g. "bunny.org"

RETURN VALUE

DM_SMTP_OK: Domain name set successfully.
DM_SMTP_NAMETOOLONG: Domain name was too long (more than **DM_SMTP_MAX_SRVLEN**=100 chars).

LIBRARY

```
dm_smtp.lib (Rabbit-based target)
dm_smtp.c (nonRabbit-based target)
```

SEE ALSO

devmate_smtp_sendmail, devmate_smtp_setserver

devmate_smtp_setsocket

```
int devmate_smtp_setsocket(uint8 socket);
```

DESCRIPTION

Sets the DeviceMate socket number to use for SMTP. A socket needs to be set since sockets are shared between SMTP and any direct use of the TCP/IP subsystem by the application. If this function is not called, the socket number will default to that defined in **tc_conf.lib**; currently 2. The socket is used only while sending mail. When **devmate_smtp_mailtick()** returns a value other than **DM_SMTP_PENDING**, the socket may be used for other TCP connections, until it is desired to send another e-mail.

PARAMETER

socket Socket number

RETURN VALUE

DM_SMTP_OK: Socket number set successfully.

DM_SMTP_BADSOCKNO: Socket number is invalid. The requested socket number must be between 1 and **devmate_tcp_maxsocket()** inclusive.

LIBRARY

```
dm_smtp.lib (Rabbit-based target)
dm_smtp.c (nonRabbit-based target)
```

SEE ALSO

devmate_smtp_sendmail, devmate_smtp_setserver

devmate_smtp_setserver

```
int16 devmate_smtp_setserver(char* server);
```

DESCRIPTION

Sets the SMTP server. This value overrides **DEVMATE_SMTP_SERVER**. If **DEVMATE_SMTP_SERVER** is defined to a suitable default (in tc_conf.lib/h) then there is no need to set the server using this function.

PARAMETER

server String containing the server name e.g. "mail.foo.org"

RETURN VALUE

DM_SMTP_OK: Server name was set successfully.
DM_SMTP_NAMETOOLONG: Server name was too long.

LIBRARY

dm_smtp.lib (Rabbit-based target)
dm_smtp.c (nonRabbit-based target)

SEE ALSO

devmate_smtp_sendmail, devmate_smtp_setdomain

devmate_smtp_status

```
int devmate_smtp_status(void);
```

DESCRIPTION

Returns a status code for the current message being sent.

RETURN VALUE

DM_SMTP_SUCCESS: E-mail sent.
DM_SMTP_PENDING: E-mail not sent, call **devmate_smtp_mailtick()** again.
DM_SMTP_TIME: E-mail not sent within **DM_SMTP_TIMEOUT** seconds.
DM_SMTP_UNEXPECTED: Received an invalid response from SMTP server.

LIBRARY

dm_smtp.lib (Rabbit-based target)
dm_smtp.c (nonRabbit-based target)

SEE ALSO

devmate_smtp_sendmail

5.3 Web Page Variables Subsystem

devmate_var_add

```
int devmate_var_add(char* name, void* variable, uint16 type,
   char* format, uint16 maxvarlen, uint16 servermask);
```

DESCRIPTION

Add a variable entry to be sent to the DeviceMate. The variable is queued for transmission to the DeviceMate. The status of the addition can be checked with **devmate_var_check_status()**.

PARAMETERS

name	A pointer to a text identifier for the variable. This name must match the name given in the HTML page that references the variable.
variable	A pointer to the actual variable
type	The type of the variable: **VAR_UINT8**, **VAR_INT16**, **VAR_INT32**, **VAR_FLOAT32**, or **VAR_STRING**
format	The printf-style specifier that will be used to display the variable in a web page.
maxvarlen	Maximum length of the variable's contents. Only necessary for **VAR_STRING**-type variables; for all other variable types it is ignored.
servermask	A bitmask selecting which servers may use this variable: **VAR_SERVER_HTTP** **VAR_SERVER_USER**

RETURN VALUE

0: Success.
1: Error (e.g. the variable table is full).

LIBRARY

dm_var.lib (Rabbit-based targets)
dm_var.c (nonRabbit-based targets)

SEE ALSO

devmate_var_update, devmate_var_check_status

devmate_var_check_status

```
int devmate_var_check_status(char* name);
```

DESCRIPTION

Returns the status of a given variable.

PARAMETERS

name The name of the variable to check

RETURN VALUE

DEVMATE_VAR_STATUS_OK: Variable has no updates pending and last operation succeeded.
DEVMATE_VAR_STATUS_UPDATE_PENDING: Variable has an update pending.
DEVMATE_VAR_STATUS_UPDATE_SENT: Update sent, but no rely received yet.
DEVMATE_VAR_ERROR_FULL: Last operation failed —DeviceMate variable table is full.
DEVMATE_VAR_ERROR_MISC: Last operation failed because of miscellaneous error (such as a communications problem).
DEVMATE_VAR_BAD_VARIABLE: The given variable name does not exist.

LIBRARY

```
dm_var.lib (Rabbit-based targets)
dm_var.c (nonRabbit-based targets)
```

SEE ALSO

devmate_var_add, devmate_var_update

devmate_var_update

```
int devmate_var_update(char* name);
```

DESCRIPTION

Schedules an update of the given variable.

PARAMETERS

name The name of the variable to update

RETURN VALUES

0: Success.

1: Error (e.g. the variable name does not exist in the internal variable table).

LIBRARY

dm_var.lib (Rabbit-based targets)
dm_var.c (nonRabbit-based targets)

SEE ALSO

devmate_var_add, devmate_var_check_status

5.4 File System Subsystem

Many of these API functions require a subsequent call to **devmate_fs_finish()**, which is noted in the relevant function descriptions.

devmate_fs_append

```
int devmate_fs_append(DMFile *file, char *buf, int length);
```

DESCRIPTION

This function will write the data in the specified buffer to the specified file. It works similarly to a standard "write" function.

devmate_fs_append() must be called as part of a *transaction:* flanked by calls to **devmate_fs_open()** and **devmate_fs_close()**.

Files can be built piece by piece as part of one transaction, i.e., multiple calls to **devmate_fs_append()** flanked by the open/close functions. Once **devmate_fs_close()** has been called and has returned successfully, that file can not be written to again.

If this function is called on a file that already exists, an error is returned. On error, the entire transaction has been terminated and must begin from the beginning i.e., starting with the call to **devmate_fs_open()**.

PARAMETERS

file	Pointer to data structure that identifies the transaction
buf	Pointer to user's data
length	The number of bytes from **buf** to append to the specified file

RETURN VALUE

≥**0**: The number of bytes actually written.
-1: Error, the transaction has been terminated.

LIBRARY

```
dm_fs.lib (Rabbit-based targets)
dm_fs.c (nonRabbit-based targets)
```

devmate_fs_close

```
int devmate_fs_close(DMFile *file, FNumber *id);
```

DESCRIPTION

Spin on this function while it returns **TC_PENDING**. This will flush any remaining data and complete the transaction. The number that identifies the file is stored at the location pointed to by the 2nd parameter, **id**, when the transaction is complete.

PARAMETERS

file	Pointer to transaction identifier
id	Pointer to number that identifies the file

RETURN VALUE

TC_PENDING: File is still closing, call function again
TC_ERROR: Entire transaction must be redone
TC_SUCCESS: Transaction completed

LIBRARY

```
dm_fs.lib (Rabbit-based targets)
dm_fs.c (nonRabbit-based targets)
```

devmate_fs_delete

```
int devmate_fs_delete(DMFile *file, FNumber id);
```

DESCRIPTION

This function causes the specified file to be deleted on the DeviceMate. A subsequent call to **devmate_fs_finish()** is required.

PARAMETERS

file	Pointer to transaction identifier
id	File to delete

RETURN VALUE

TC_PENDING: Call **devmate_fs_finish()** again.
TC_ERROR: File doesn't exit or some other error occurred.
TC_SUCCESS: File removal succeeded.

LIBRARY

```
dm_fs.lib (Rabbit-based targets)
dm_fs.c (nonRabbit-based targets)
```

devmate_fs_deleteB

```
int devmate_fs_deleteB(DMFile *file, FNumber id, long timeout);
```

DESCRIPTION

Blocking version of **devmate_fs_delete()**: causes the specified file to be deleted on the DeviceMate.

PARAMETERS

file	Pointer to transaction identifier
id	File to delete
timeout	Number of milliseconds to wait before timing out. Zero means wait forever.

RETURN VALUE

TC_ERROR: File doesn't exit or some other error occurred.
TC_SUCCESS: File removal succeeded.

LIBRARY

```
dm_fs.lib (Rabbit-based targets)
dm_fs.c (nonRabbit-based targets)
```

devmate_fs_finish

```
int devmate_fs_finish(DMFile *file);
```

DESCRIPTION

This function is required by some of the other functions to complete their requests. After one of these functions is called:

- **devmate_fs_init()**
- **devmate_fs_rename()**
- **devmate_fs_delete()**
- **devmate_fs_idlookup()**

a call to **devmate_fs_finish()** should be made until it stops returning **TC_PENDING** and returns the return value of the function that required its use.

PARAMETERS

file	Pointer to transaction identifier: identifies the command that was requested.

RETURN VALUE

TC_PENDING: The transaction is still being processed.
TC_ERROR: Error, e.g., unknown file command.
Other: Return value of function that needed call to **devmate_fs_finish()**.

LIBRARY

```
dm_fs.lib (Rabbit-based targets)
dm_fs.c (nonRabbit-based targets)
```

devmate_fs_idlookup

```
int devmate_fs_idlookup(DMFile *file, char *name, FNumber *id);
```

DESCRIPTION

Look up the ID of the file based on its name. When the results arrive, they will be stored at ***id**. A subsequent call to **devmate_fs_finish()** is required.

PARAMETERS

file	Pointer to transaction identifier
name	The name of the file
id	Pointer to where file ID number will be if the function calls returns successfully.

RETURN VALUE

TC_PENDING: Call **devmate_fs_finish()** again.
TC_ERROR: File doesn't exit or some other error occurred.
TC_SUCCESS: Success, ID of the file is in ***id**.

LIBRARY

```
dm_fs.lib (Rabbit-based targets)
dm_fs.c (nonRabbit-based targets)
```

devmate_fs_idlookupB

```
int devmate_fs_idlookupB(DMFile *file, char *name, FNumber *id
   long timeout);
```

DESCRIPTION

Blocking version of **devmate_fs_idlookup()**: look up the ID of the file based on its name. When the results arrive, they will be stored at ***id**.

PARAMETERS

file	Pointer to transaction identifier
name	The name of the file
id	Pointer to where file ID will be if the function calls returns successfully.
timeout	Number of milliseconds to wait before timing out. Zero means wait forever.

RETURN VALUE

TC_ERROR: File doesn't exit or some other error occurred.
TC_SUCCESS: Success, ID of the file is at ***id**.

LIBRARY

dm_fs.lib (Rabbit-based targets)
dm_fs.c (nonRabbit-based targets)

devmate_fs_open

```
int devmate_fs_open( DMFile *file, FNumber id, char *name, int
   flags );
```

DESCRIPTION

This function starts a transaction. This will initiate a file download and will associate a name with the downloaded file. After this functions returns **TC_SUCCESS**, the transaction is in progress and **devmate_fs_append()** may be called for this file. No blocking version exists for a transaction, as errors are reported on subsequent calls automagically.

PARAMETERS

file	Pointer to transaction identifier
id	The file's id number
name	The file's name
flags	• **TC_FS_FLAGS_NEWID**: recommended value for this parameter, causes DeviceMate to issue a file id number for the specified file that is guaranteed to be unique. • **TC_FS_FLAGS_REPLACE**: Replaces an existing file.

RETURN VALUE

TC_SUCCESS: Transaction has started successfully.
TC_PENDING: Couldn't store the filename - try again later.
TC_ERROR: Error, e.g., filename too long.

LIBRARY

```
dm_fs.lib (Rabbit-based targets)
dm_fs.c (nonRabbit-based targets)
```

devmate_fs_rename

```
int devmate_fs_rename(DMFile *file, FNumber id, char *name);
```

DESCRIPTION

Rename the specified file to the new name. A subsequent call to **devmate_fs_finish()** is required.

PARAMETERS

file	Pointer to transaction identifier
id	Number internal to the file system that identifies the file
name	New name for the file

RETURN VALUE

TC_PENDING: Call **devmate_fs_finish()**.
TC_ERROR: Error, e.g. file doesn't exist.
TC_SUCCESS: Success, file was renamed.

LIBRARY

```
dm_fs.lib (Rabbit-based targets)
dm_fs.c (nonRabbit-based targets)
```

devmate_fs_renameB

```
int devmate_fs_renameB(DMFile *file, FNumber id, char *name,
   long timeout);
```

DESCRIPTION

A blocking version of **devmate_fs_rename()**: rename the specified file to the given name.

PARAMETERS

file	Pointer to transaction identifier
id	Number internal to the file system that identifies the file
name	New name for the file
timeout	Number of milliseconds to wait before timing out. Zero means wait forever.

RETURN VALUE

TC_ERROR: Error, e.g. file doesn't exist.
TC_SUCCESS: Success, file was renamed.

LIBRARY

```
dm_fs.lib (Rabbit-based targets)
dm_fs.c (nonRabbit-based targets)
```

devmate_fs_sync

```
int devmate_fs_sync(DMFile *file, int flags);
```

DESCRIPTION

Initialize the file system on the DeviceMate. This function does a handshake between the target and the DeviceMate, so that both sides know that everything has reset and is starting from the beginning. A subsequent call to **devmate_fs_finish()** is required.

PARAMETERS

file	Pointer to transaction identifier
flags	These are the options that the target can request from the DeviceMate. Initially: • **TC_FS_FORMAT**: Erase the file system • **0x0000**: Do not erase the file system

RETURN VALUE

TC_ERROR: Something is wrong.
TC_PENDING : The finish function should be used.
TC_SUCCESS: The file system was successfully initialized.

LIBRARY

```
dm_fs.lib (Rabbit-based targets)
dm_fs.c (nonRabbit-based targets)
```

devmate_fs_syncB

```
int devmate_fs_syncB(DMFile *file, int flags, long timeout);
```

DESCRIPTION

Blocking version of **devmate_fs_sync()**: initialize the file system on the DeviceMate. This function does a handshake between the target and the DeviceMate, so that both sides know that everything has reset and is starting from the beginning.

PARAMETERS

file	Pointer to transaction identifier
flags	These are the options that the target can request from the DeviceMate. Initially: • **TC_FS_FORMAT**: Erase the file system • **0x0000**: Do not erase the file system
timeout	Number of milliseconds to wait before timing out. Zero means wait forever.

RETURN VALUE

TC_ERROR: Something is wrong.
TC_SUCCESS: The file system was successfully initialized.

LIBRARY

```
dm_fs.lib (Rabbit-based targets)
dm_fs.c (nonRabbit-based targets)
```

5.5 Message Logging Subsystem

All the functions in **dm_log.lib** are non-blocking, i.e. they all set some internal state, but return immediately. A call to **devmate_log_put()** may return without actually accomplishing its intended goal; it may be necessary to call it multiple times in order to complete the desired action.

devmate_log_init

```
int devmate_log_init(void);
```

DESCRIPTION

Establish communication link to the DeviceMate unit. If a link is already established, then it is re-established.

RETURN VALUE

1

LIBRARY

dm_log.lib

SEE ALSO

devmate_log_status

devmate_log_put

```
int devmate_log_put(uint16 ffp, char * data, int length);
```

DESCRIPTION

Write a log message. Note that if the message is filtered, then the return code is zero, which is the same as "successfully buffered."

Priority codes are not defined, however you may wish to use the standard "syslog" encoding as follows:

0: Emergency; 1: Alert; 2: Critical; 3: Error; 4: Warning; 5: Notice; 6: Informational; 7: Debug.

The facility number is decoded by the DeviceMate into the appropriate actual destination - see filesystem\log.lib for details. Unless configured otherwise, facility 0 goes to the filesystem; 1 goes to "stdout" (for debugging); 2 goes to both FS and stdout. Other facilities are discarded unless specifically configured.

PARAMETERS

ffp	Format, facility and priority coded as a 16-bit word: bits 0-2: priority (0-7) bits 3-7: facility (0-31) bits 8-15: format (0 for ASCII string; others as defined by user)
data	Pointer to the first byte of log message
length	Length of log message. Must be ≤ 115

RETURN VALUE

-1: Error occurred. Logging not available.
-2: Message too long (more than 115 chars).
0: Message buffered for sending (or filtered out).
1: Message could not be buffered. Retry same message later if desired.

LIBRARY

`dm_log.lib`

EXAMPLE

```
devmate_log_put(7, "Hello log", 9)
```

writes "Hello log" as a priority 7 message to facility 0. The format is also 0 (ASCII string). Note that the null terminator need not be included, since the log basically treats all data as binary. If the length was given as '10' in the above example then the null terminator would be included in the log, however this would waste one byte of space.

devmate_log_setfacilityfilter

```
uint32 devmate_log_setfacilityfilter(uint32 filter_allow,
   uint32 filter_prevent);
```

DESCRIPTION

Set local log message filtering, based on message facility. The parameters are bit masks. If bit 0 (the LSB) is set, this corresponds to facility 0; bit 31 corresponds to facility 31. For example:

```
devmate_log_setfacilityfilter(1L<<1, 1L<<9)
```

allows facility 1 and prevents facility 9. All other facilities are left in their previous state. Initially, all facilities are allowed.

PARAMETERS

filter_allow	Each bit set to '1' allows the corresponding facility messages to be sent
filter_prevent	Each bit set to '1' prevents the corresponding facility messages from being sent. If the same bit is set in both parameters, then the message is prevented.

RETURN VALUE

The previous "allow" mask.

LIBRARY

`dm_log.lib`

devmate_log_setpriorityfilter

```
uint8 devmate_log_setpriorityfilter(uint8 filter_allow, uint8
   filter_prevent);
```

DESCRIPTION

Set local log message filtering, based on message priority. The parameters are bit masks. If bit 0 (the LSB) is set, this corresponds to priority 0 (highest priority); bit 7 corresponds to the lowest priority (7). For example:

```
devmate_log_setpriorityfilter(1<<3, 1<<5)
```

allows priority 3 and prevents priority 5. Other priorities are left in their previous state. Initially, all priorities are allowed.

PARAMETERS

filter_allow	Each bit set to '1' allows the corresponding priority messages to be sent
filter_prevent	Each bit set to '1' prevents the corresponding priority messages from being sent. If the same bit is set in both parameters, then the message is prevented.

RETURN VALUE

The previous "allow" mask.

LIBRARY

`dm_log.lib`

devmate_log_status

```
int devmate_log_status(void)
```

DESCRIPTION

Test communication link to the DeviceMate. The link status indicates whether logging functions can be accessed on the DeviceMate.

RETURN VALUE

0: The link is currently being established

1: The link is successfully established

-1: An error has occurred. Call **devmate_log_init()** to re-establish the link. An error typically occurs because there is either no DeviceMate, or because the DeviceMate is not configured for logging support.

LIBRARY

dm_log.lib

SEE ALSO

devmate_log_init

5.6 Watchdog Subsystem

devmate_wd_init

```
int devmate_wd_init(int block_ms);
```

DESCRIPTION

This function initializes the watchdogs. It will clear any watchdogs on the DeviceMate.

PARAMETERS

block_ms	Number of milliseconds to wait before timing out.

RETURN VALUE

0: Success.

-1: Error (eg, timeout occurred).

LIBRARY

dm_wd.lib (Rabbit-based targets)
dm_wd.c (nonRabbit-based targets)

devmate_wd_add

```
int devmate_wd_add(char* name, long updatewith, int block_ms);
```

DESCRIPTION

This adds a watchdog on the DeviceMate and the countdown begins. If **updatewith** amount of time passes before a **devmate_wd_hit()** call is received from the target, the DeviceMate resets the target. If this function is called on an existing watchdog, the amount of time to wait for a **devmate_wd_hit()** is changed to the new **update-with** value.

PARAMETERS

name	Text identifier for watchdog instance
updatewith	Maximum number of milliseconds that can pass without the target hitting the watchdog. Exceeding this time causes a reset of the target.
block_ms	Number of milliseconds function to wait before timing out

RETURN VALUE

0: Success.

-1: Error (eg, timeout occurred).

LIBRARY

dm_wd.lib (Rabbit-based targets)
dm_wd.c (nonRabbit-based targets)

devmate_wd_hit

```
int devmate_wd_hit(char* name);
```

DESCRIPTION

Reset the watchdog. This function returns immediately after attempting to send the hit-wd message. If there are no buffers available or if the transmission gets corrupted, the message will not reach the DeviceMate. To guard against this, this function should be called a number of times within the **updatewith** time period. If the occasional message gets lost, the system will continue to operate.

PARAMETERS

name	The text identifier for the watchdog— it must match the one passed to **devmate_wd_add()**

RETURN VALUE

0: Success.
-1: Error.

LIBRARY

dm_wd.lib (Rabbit-based targets)
dm_wd.c (nonRabbit-based targets)

devmate_wd_rmv

```
int devmate_wd_rmv(char* name, int block_ms);
```

DESCRIPTION

This function removes the watchdog from the list on the DeviceMate.

PARAMETERS

name	The text identifier for the watchdog— it must match the one passed to **devmate_wd_add()**
block_ms	Number of milliseconds to wait before timing out

RETURN VALUE

0: Success.
-1: Error (eg, timeout occurred).

LIBRARY

dm_wd.lib (Rabbit-based targets)
dm_wd.c (nonRabbit-based targets)

Chapter 6. Porting Guidelines for NonRabbit-Based Targets

Target Communications may be implemented on any target processor that has an asynchronous serial peripheral. This section describes how to link the serial peripheral to the Target Communications Library (TCL), expressed as a programming interface, so that the target may communicate with a DeviceMate unit.

6.1 Overview

The TCL is a framework of ANSI C code that implements the majority of the Target Communications (TC) Protocol. Since the serial peripheral interface is highly specific to each target architecture, a certain amount of code will need to be created to bind the serial peripheral to the TCL framework. This is called the adapter.

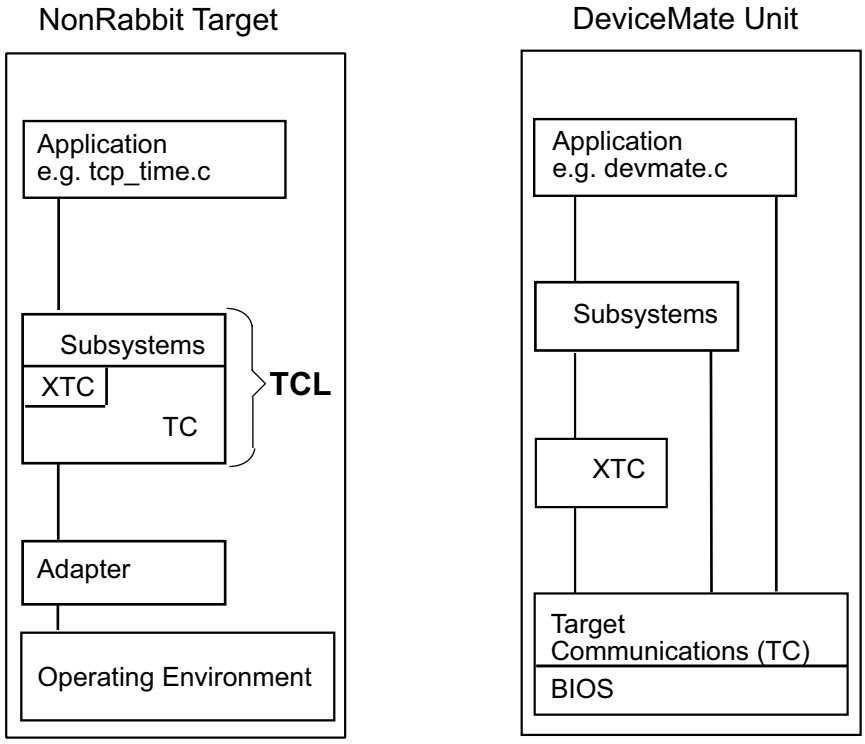


Figure 1. Block Diagram of Software Components for a DeviceMate System with a NonRabbit-Based Target

6.1.1 Steps for Porting to a Non Supported Target

When porting the TCL to a target that is not currently supported by Z-World, it may be necessary to modify or re-implement some of the source code that is provided. The following list summarizes the steps to take.

- Choose the supported source code base which is written for an architecture most similar to the intended target. For example, if targeting the MC68000 line of processors, then the Sparc version would be the best starting point.
- Determine the style of interaction between the asynchronous serial port and the software. This may be either a polling or an interrupt-driven style. Interrupt-driven is preferable, however polling may be preferable in the case that direct control of the serial port hardware is made difficult by the operating environment.
- Edit **tc_conf.h**. This header file contains many macros that describe the target architecture.
- Implement the serial port interface. For an interrupt-driven style, use **tc_386ex.c** (from the 80386EX examples) as a starting point. For polling style, try **tc_sysv.c** from the Sparc or Linux examples.
- Implement any other routines that are required by the definitions in **tc_conf.h**. Target processors with a non-linear address space will require the most work.

6.1.2 Sample Architectures

Examples of porting the TCL to several different architectures are provided with Dynamic C. The sample architectures are:

- **Intel 80386EX**: This processor is used in many embedded systems. The sample programs assume that an embedded DOS environment is available, however there is minimal dependence on DOS. This sample will also work on a desktop PC running DOS. This sample was tested using the Borland C++ 4.52 command-line compiler.
- **Sparc**: This processor is unlikely to be used in embedded systems, however it demonstrates use of the TCL on a big-endian RISC processor. This code could be used almost unaltered on Motorola-based processors. This sample was tested on a Sun Ultra Sparc, using the gcc compiler for Solaris 8.
- **Linux on 80x86**: Embedded Linux is becoming a popular choice for high-end embedded systems. Naturally, the sample will also work on a desktop Linux system with any reasonably recent kernel and gcc.

The Sparc and Linux samples assume a Unix-like operating system is available. Ideally, TCL would be implemented as a device driver, however the samples run as normal user-mode processes.

> **NOTE:** The samples will require you to provide the appropriate operating environment and compiler.

6.2 NonRabbit-Based Target Properties

A nonRabbit-based target is assumed to have the following properties:

- Sends and receives 8-bit characters with no parity and 1 stop bit.
- Can operate at one of the "standard" rates of 2400, 4800, 9600, 19200, 38400, 57600 or 115200 bits per second.
- Configuration of the serial device is under control of the application and/or OS; TCL does not provide any support for setting the communications parameters.
- Only requires connection of TxData, RxData and signal ground.
- Operates in "raw" mode i.e. the operating system, if any, does not perform any insertion or deletion of characters such as XON/XOFF software flow control.
- When a character is received, the application may be asynchronously notified via an interrupt service routine, or something which acts like or simulates an ISR. The ISR is considered to be part of the adapter.
- When a character has been transmitted, the application may be asynchronously notified that a new character may be transferred to the serial peripheral for transmission.
- The serial peripheral may or may not have any buffering of its own (e.g. a FIFO), or buffering may or may not be provided by the operating system (e.g. a small circular buffer).

It is assumed that there is no operating system as such, however the existence of an operating system may be advantageous if it can handle some of the details of serial communication.

6.3 TCL Interface

The interface works in two directions: the serial peripheral ISR calls TCL functions to process received data (or obtain data to transmit), and the TCL calls some functions provided by the adapter in order to notify the adapter of any significant change of state in the TCL. Since some of the routines may be invoked asynchronously, the adapter may also provide serialization routines.

6.3.1 TCL Data-Handling API

The data-handling API comprises functions that either generate data for transmission or process received data. These functions are called by the ISR part of the adapter. Since these functions are not called from TCL itself, it is possible for them to be in-lined directly in the ISR, saving one level of function call overhead. These functions perform a minimal amount of processing that is roughly constant for each transmitted or received byte. This makes direct call from an ISR practicable, since interrupt service latency is minimized.

These functions depend on state information saved in a single instance of a **`TCState`** structure. This structure is defined in **`tc.c`**. It is assumed that static variables can be accessed from the ISR.

6.3.1.1 Data Types

The TCL relies heavily on data types with definite sizes. The following definitions are assumed:

```
typedef ??? uint8;      // Unsigned 8-bit byte
typedef ??? uint16;     // Unsigned 16-bit word
typedef ??? int16;      // Signed 16-bit word
typedef ??? uint32;     // Unsigned 32-bit longword
typedef ??? faraddr_t;  // Far pointer to uint8
```

??? denotes the appropriate basic type for the target architecture. The default definitions, in **`tcconfig.h`**, are appropriate for the majority of architectures: a **`short`** is assumed to be 16 bits, and a **`long`** is assumed to be 32 bits.

6.3.1.1.1 Far Pointers

`faraddr_t` is defined for architectures which need to make a distinction between near and far pointers. Architectures with a single, linear, address space can define this as

```
typedef uint8 * faraddr_t;
```

Other architectures should define this using the appropriate qualifier, or as a “long” type. Some macros that define how to access memory using far pointers must be defined—please see “Configuring the TCL Framework” on page 107.

It is assumed that if a **`faraddr_t`** is incremented (i.e. add 1) then the next byte of data space storage, considered as a linear array of bytes, will thereby be addressed. This implicitly limits TCL to run on byte-addressable architectures. This assumption holds true if **`faraddr_t`** is defined as either a **`uint8*`** or as an integer type.

6.3.1.2 Received Data Handler

Each character received by the serial peripheral is passed to the received data handler. This may be performed on a character-by-character basis, or a sequence of more than one character at a time. Single character processing may be used if the peripheral does not buffer more than one character, and it is desired to process each character as it arrives. Peripherals that support a FIFO buffer, or operating systems which provide a small amount of buffering, may pass multiple characters in one call. This may improve efficiency.

Since the receive handler may be called from an ISR, it is important that the data be handled as quickly as possible. If necessary, the receive data handler in the TCL may be recoded in the target CPU’s assembler language.

The prototype for the receive data handler is:

```
void _tc_rxdata(uint8 * data, uint16 len)
```

The ISR calls this routine whenever a new character arrives. The address of the character is provided in **`data`**, and the number of characters at that address in **`len`**. **`len`** will be 1 for unbuffered (character-by-character) processing. **`_tc_rxdata()`** must be called exactly once for any and all received characters.

6.3.1.3 Transmit Data Handlers

Two functions are provided for the purpose of obtaining the next character to transmit. The first function is simpler to use since it performs all necessary processing, but the second function may allow greater efficiency. If the second function is used, then the first function may be deleted from the TCL code base since it will not otherwise be used.

The first function's prototype is:

```
uint16 _tc_txdata(uint8 * data, uint16 max_len)
```

This function is called whenever the serial peripheral is able to accept the next character (or several characters) for transmission. This is typically performed from the "transmit buffer ready" ISR. The ISR provides a pointer to a memory area to store the data, and specifies the length of that area in **max_len**. The specified length must be at least 1. The return value is the number of characters that were actually stored in the given data area. This may be less than **max_len**. Specifically, zero may be returned when there is nothing to transmit. In this case, the ISR should let the serial peripheral become idle i.e. not transmit any further data. If a non-zero length is returned, then the ISR must queue the returned character(s) for transmission.

The second function returns a complete Target Communications packet for transmission. This is more complex to use, since a data structure is returned which must be scanned to determine where and how much data is to be transmitted. In addition, the caller is responsible for inserting HDLC frame start, escape sequences and a final checksum.

The second function's prototype is

```
_TCGatherSeg * _tc_txpacket(void)
```

The definition of the returned structure is

```
typedef struct {
   uint16 len;
   faraddr_t addr;
} _TCGatherSeg;
```

The return value from **_tc_txpacket()** actually points to an array of the above structures. The last element in the array is determined by examining the **len** field: this is zero for the terminating entry. The terminating entry does not actually specify any data to transmit. The adapter must scan the array of **_TCGatherSeg** structures, transmitting each segment in sequence. **addr** is the far address of the data to transmit, and **len** is the length (in characters) of the data starting at that address. When the data from one segment is completely transmitted, the next array element is examined and its data transmitted. The process halts when the element contains a zero **len** field. At this point, the next packet may be obtained by another call to **_tc_txpacket()**.

The last segment will have a length of 2 and its data will be initialized to zeros. This is the final checksum for the entire frame. It must be computed by the caller based on the checksum of all data previously transmitted in the packet, not counting the frame start character or any escaping transformations (described below). When all segments except for the last (checksum) have been transmitted, then the checksum is sent with appropriate HDLC escapes but without further updating of the checksum itself. The checksum algorithm is described below.

If **NULL** is returned then there is no data to transmit. The serial peripheral transmitter should be allowed to become idle in this case.

The data locations returned are guaranteed to be stable until the next call to this function. Calling the function again implicitly releases the old data buffers to be re-used in the new return. This

implies that the adapter must not try to “remember” to position of any data from a previous call to this function, since that data may be overwritten by the current call.

The data returned is raw data. The adapter must insert a frame start character before transmitting data from the first returned segment. The frame start character is 0x7E. Each data byte must be examined before transmission. If the data is either 0x7E or 0x7D, then two characters must be transmitted:

- 0x7E must be transmitted as 0x7D 0x5E
- 0x7D must be transmitted as 0x7D 0x5D
- All other characters are transmitted without change.

This is standard HDLC frame start and escape sequencing. Note that **`_tc_txdata()`** already performs all HDLC processing; only this function requires it to be performed by the adapter.

If the adapter makes calls to this function, it must not make calls to **`_tc_txdata()`**, and vice-versa.

6.3.1.4 Checksum Algorithm

The checksum used by the TCL is a 16-bit Fletcher checksum. This has the advantage of being computable using 8-bit arithmetic on machines that have an instruction to add the carry flag to the accumulator. The following C code shows how the checksum is updated given the next byte in the stream, **`c`**. **`a`** and **`b`**, the 1st and 2nd bytes of the checksum, are initialized to zero.

```
uint8  c;    // next character
uint16 a;    // 1st byte of checksum (in LSB)
uint16 b;    // 2nd byte of checksum (in LSB)

a += c;
if (a > 255) a -= 255;
b += a;
if (b > 255) b -= 255;
```

Checksums are used to protect the TC header, as well as the entire TC packet. The TC header checksum is automatically computed.

6.3.2 Adapter Notification API

TCL notifies the adapter of significant events by calling functions which must be implemented by the adapter.

6.3.2.1 Transmission Start

Whenever a new packet is available for transmission, the transmission start function is called:

```
void _tc_start(void)
```

This is provided as a wake-up call to the serial peripheral transmitter, since it may have become idle since the last packet was transmitted. Typically, this function starts the ball rolling by determining the first new character to transmit, and using that character to prime the transmit shift register. Thereafter, the serial peripheral will generate transmit-ready interrupts in order to retrieve subsequent characters.

6.3.2.2 Event Signal

Whenever a new packet has completed arrival, or has completed transmission, this function is called by the TCL to notify the adapter. The adapter may then notify the application. This will allow the application to terminate a blocking (wait) state.

```
void _tc_signal(void)
```

The occurrence of a call to this function means that the internal state of the TCL has changed in a way that will be meaningful to the application. If the application happens to be waiting for an event, this is an appropriate time for the application to make further calls to the TCL API.

This function is primarily intended for multitasking OS support, where it may indeed issue a “signal” or post to a semaphore. If there is no OS, then typically there will be a main loop which continually polls TCL for activity (via the **devmate_tick()** function). In this case, there is no need for this function to do anything; however it should still be coded.

6.3.3 Configuring the TCL Framework

TCL is provided as a portable framework of routines coded in ANSI C. Because of the wide range of target architectures, it will often be necessary to make some modifications to the TCL code. This process is made easier by defining a set of macros which tell TCL how to contend with the various quirks of the target architecture. Some miscellaneous support routines are also required, which are described in this section.

The main issues to contend with are

- Structure packing conventions.
- Byte ordering.
- Memory model (linear, segmented etc.).
- Interaction with operating system, if any.
- Resource serialization.

The macros are defined in **tcconfig.h**. A description of each macro is included in the header file, with some additional information below.

6.3.3.1 Byte Swapping and Packing

Since TC uses a little-endian format, big-endian target processors will need to perform byte swapping as well as packing. Little-endian targets will only need to perform packing. Targets which are both little-endian and have no alignment restrictions (such as the Rabbit processor and other Intel-derived 8-bit processors) do not need to define these functions.

If the preprocessor macro **TC_BIG_ENDIAN** is defined in **tc_config.h**, then the appropriate byte-swapping code will be automatically included. A function defined in **tc.c**, **_tc_reorder()**, is used to perform table-driven byte swapping for an entire data structure.

If **TC_STRICT_ALIGN** is defined, then TCL assumes that the compiler may introduce padding into structure definitions, in order to align elements on the most efficient boundary for memory access purposes. In this case, TCL automatically includes pack/unpack code to convert from internal structure format into transmission format (which is fully packed).

If, in addition to **TC_STRICT_ALIGN**, the macro **TC_BUS_ALIGN** is defined, then the architecture is assumed to require that multi-byte elements (uint16, uint32 etc.) be stored to and loaded from addresses which are a multiple of their size, as opposed to architectures such as the 80x86 which do not strictly require address alignment. (Address alignment is usually more efficient, which is why compilers add padding to structures; however, some architectures actually require alignment.)

6.3.3.2 Memory Model

tcconfig.h contains some configuration items to specify the memory model. TCL assumes that there are two types of pointers: near and far. This is the case on the Rabbit processor, as well as many other architectures with a segmented memory model. TCL can be configured to ignore this distinction, if the memory address space is linear and all pointers are the same size. Some architectures' C compiler can automatically model a limited linear address space, even on a segmented architecture. This is the case on the 80x86 processor using the smaller memory models, where the compiler simply makes all pointers near.

Since TCL is a frugal resource consumer, it is usually sufficient to use a small memory model. Otherwise, if there is a distinction between near and far pointers, then TCL uses far memory to hold data buffers, and near memory for everything else.

If **TC_LINEAR_MEMORY** is defined, then it is assumed that the address space is linear. Many of the internal conversion calls are replaced with in-line code or a call to the **memcpy()** function.

If **TC_LINEAR_MEMORY** is not defined, then a number of routines must be defined for dealing with data movement between near and far locations. All far pointers are referred to by the **faraddr_t** typedef, which is basically a far pointer to **uint8** (an unsigned char).

6.3.3.2.1 Adapter Routines

The routines which must be implemented by the adapter are listed below. The default implementation is used if **TC_LINEAR_MEMORY** is defined.

faraddr_t xalloc(long sz)

Allocate **sz** bytes of storage, returning the base address (or **FARADDR_NULL** if error). Default implementation is **(uint8 *)malloc(sz)**.

faraddr_t paddr(void * virt)

Convert the given near address to a far address. TCL assumes that this is always possible. Default implementation is a pointer cast (i.e. no code generated).

void xmem2root(void *dest, faraddr_t src, unsigned len)

Copy memory from a far to a near address. Default implementation uses **memcpy()**.

void root2xmem(faraddr_t dest, void *src, unsigned len)

Copy memory from a near to a far address. Default implementation uses **memcpy()**.

void xmem2xmem(faraddr_t dest, faraddr_t src, unsigned len)

Copy memory from a far to a different far address. Default implementation uses **memcpy()**.

void _tc_setnext(faraddr_t buf, faraddr_t next)

Assuming **buf** points to a **faraddr_t** (not a **uint8**), set the pointer at ***buf** to next. Default implementation is a macro to the effect of ***(faraddr_t *)buf = next**.

faraddr_t _tc_getnext(faraddr_t buf)

Assuming **buf** points to a **faraddr_t** (not a **uint8**), retrieve the pointer at ***buf**. Default implementation is a macro to the effect of ***(faraddr_t *)buf**.

uint8 _tc_getbyte(faraddr_t buf)

Return the **uint8** (unsigned char) pointed to by **buf**. Default implementation is a direct dereference of **buf**.

6.3.3.3 Interaction with Operating System

TCL assumes that very minimal facilities are available from an operating system. The only interaction is between TCL and the serial peripheral, the application, and a simple timer. Only the timer is described here. The other interactions are described by the rest of this chapter.

TCL assumes that a simple real-time timer is available in the form of a function that returns the number of milliseconds since some arbitrary epoch. This function should take no parameters and return a uint16 value which is a snapshot of the current millisecond timer.

```
uint16 _tc_getMilliseconds(void)
```

If the OS maintains a more precise timer, or a timer with greater range than 65536 milliseconds, then the value returned should reflect the millisecond bit in the LSB. For example, if a hypothetical OS maintained a 32-bit counter (**OS_TIMER**) which incremented every microsecond, then an appropriate function definition would be

```
uint16 _tc_getMilliseconds(void)
{
   return (uint16)(OS_TIMER >> 10);
}
```

which is shifting by 10 to divide by approximately 1000. The absolute accuracy of the timer does not need to be any more precise than about +/-20%, and the resolution does not need to be any better than about 50ms. For example, if the counter increments by 50 every 50ms, then this is sufficient resolution.

The counter should wrap around modulo 65536.

Another simple real-time timer is available, also in the form of a function, this one returning the number of seconds (as a long) since some arbitrary epoch.

```
uint32 _tc_getSeconds(void)
```

If this is not available, some of the demos may not link properly, however the seconds timer is not required by TCL itself.

6.3.3.4 Serialization

TCL uses buffer queues to perform all resource serialization. Two queue manipulation functions must be defined. If the preprocessor symbol **TC_CALLED_FROM_ISR** is defined, then these functions must be implemented by the adapter. Otherwise, it is assumed that TCL is single-threaded and there is no serialization requirement. **TC_CALLED_FROM_ISR** indicates that another execution thread (typically a serial peripheral ISR) calls TCL routines. In this case, it is important to define the queue management routines so that they operate atomically with respect to the ISR and the main application.

The two functions **_tc_queue_buffer()** and **_tc_get_buffer()** may be coded simply to wrap interrupt-disabling code around calls to the non-protected queue management functions. The non-protected functions are defined with the same prototype and name, except that an additional underscore precedes the name. For example, a protected **_tc_get_buffer()** function may be coded as:

```
faraddr_t _tc_get_buffer(faraddr_t *chain)
{
   faraddr_t rc;
   int was_enabled = currently_enabled();

   if (was_enabled)
      disable();                 // Disable interrupts
   rc = __tc_get_buffer(chain);  // Call non-protected version
   if (was_enabled)
      enable();                  // Enable if previously enabled
   return rc;
}
```

where **enable()** and **disable()** are architecture-specific functions. Note that the enabling and disabling of interrupts should nest correctly. If desired, one level of function calls may be eliminated by modifying **__tc_get_buffer()** and **__tc_queue_buffer()** directly, and *not* defining **TC_CALLED_FROM_ISR**.

6.4 Multitasking Environment

Subsystems must be safe in a preemptive multitasking environment. The implementation of binary semaphores that map to µC/OS-II functions can be ported for use with some other preemptive multitasking environment. The following macros must be redefined by the user.

6.4.1 Locking Macros

The following macros are in **`dm_app.lib`** :

TC_LOCKING

Locking is enabled if this macro is defined in the subsystem code. Locking is automatically defined when µC/OS-II is being used.

TC_LOCKTYPE

This macro specifies the data type of the lock. If it is not defined in the subsystem and µC/OS-II is being used, **`TC_LOCKTYPE`** will be defined as **`OS_EVENT*`**: a pointer able to access an **`OS_EVENT`** data structure that will identify a binary semaphore.

TC_CREATELOCK()

This macro takes no parameters. It returns a lock handle of type **`TC_LOCKTYPE`**. For µC/OS-II, this will use the function **`OSSemCreate()`**, with 1 as the parameter (the initial count of the semaphore).

TC_LOCK(lock)

This macro takes a lock handle of type **`TC_LOCKTYPE`** as a parameter. It will block until the given semaphore can be acquired (unless locking is not being used, in which case it will do nothing). Under µC/OS-II, this will wrap the function **`OSSemPend()`**, with the time-out as 0 (infinity) and the error status being read into a dummy global variable.

By default this macro will expand to nothing.

TC_UNLOCK(lock)

This macro takes a lock handle of type **`TC_LOCKTYPE`** as a parameter. It will release the given semaphore (unless locking is not being used, in which case it will do nothing). Under µC/OS-II, this will wrap the function **`OSSemPost()`**.

By default this macro will expand to nothing.

6.4.2 Critical Sections

A critical section of code is a part of the program that accesses shared data.

The critical section macros are: **`TC_ENTER_CRITICAL()`** and **`TC_EXIT_CRITICAL()`**. They take no parameters. Under µC/OS-II they map to **`OS_ENTER_CRITICAL()`** and **`OS_EXIT_CRITICAL()`**.

They should be used with caution as they disable interrupts.

Appendix A. Guidelines for Writing Custom Subsystems

Customized subsystems may be added to the DeviceMate software. This appendix discusses the general strategy used in writing the subsystems provided by Z-World.

A.1 Software Overview

The DeviceMate communicates with the target processor using thin layers of service calls. Each feature in the DeviceMate feature set is identified by a unique packet type and communicates by sending and receiving buffers to the same packet type on the other processor. The low-level Target Communications (TC) makes a best attempt to deliver the buffers intact, but the buffer is only passed to the destination subsystem if the checksum and packet format are correct.

A.1.1 Packet Type

Packet types 0 through 15 are reserved and should not be used. The following packet type values are pre-defined:

```
#define TC_TYPE_SYSTEM 0x00   // System messages
#define TC_TYPE_DEBUG  0x01   // Debugging messages
#define TC_TYPE_WD     0x02   // Watchdogs
#define TC_TYPE_TCPIP  0x03   // TCP/IP tunneling over TC/XTC
#define TC_TYPE_FS     0x04   // File system
#define TC_TYPE_VAR    0x05   // Web-page variable
#define TC_TYPE_LOG    0x06   // Logging
```

A.1.1.1 Configuration Macro

The macro **TC_MAX_APPLICATIONS** is the maximum number of subsystems (read: callbacks) that are supported. Therefore, the largest packet type value supported is **TC_MAX_APPLICATIONS - 1**. This macro defaults to 10.

A.2 Multitasking Environment

Subsystems must be safe in a preemptive multitasking environment such as µC/OS-II. This is done with a generic implementation of binary semaphores to serialize access, or the disabling of interrupts to enter critical sections of code.

These locking macros can be redefined by the user to match the target environment.

A.2.1 Locking Macros

In **dm_app.lib**, are the following macros:

TC_LOCKING

Locking is enabled if this macro is defined in the subsystem code. Locking is automatically defined when µC/OS-II is being used.

TC_LOCKTYPE

This macro specifies the data type of the lock. If it is not defined in the subsystem and µC/OS-II is being used, **TC_LOCKTYPE** will be defined as **OS_EVENT***: a pointer able to access an **OS_EVENT** data structure that will identify a binary semaphore.

TC_CREATELOCK()

This macro takes no parameters. It returns a lock handle of type **TC_LOCKTYPE**. For µC/OS-II, this will use the function **OSSemCreate()**, with 1 as the parameter (the initial count of the semaphore).

TC_LOCK(lock)

This macro takes a lock handle of type **TC_LOCKTYPE** as a parameter. It will block until the given semaphore can be acquired (unless locking is not being used, in which case it will do nothing). Under µC/OS-II, this will wrap the function **OSSemPend()**, with the time-out as zero (infinity) and the error status being read into a dummy global variable.

TC_UNLOCK(lock)

This macro takes a lock handle of type **TC_LOCKTYPE** as a parameter. It will release the given semaphore (unless locking is not being used, in which case it will do nothing). Under µC/OS-II, this will wrap the function **OSSemPost()**.

A.2.1.1 Initialization

The initialization function of a subsystem should have something like the following:

```
#ifdef TC_LOCKING
TC_LOCKTYPE _myapp_lock;
#endif

_myapp_init() {
// ...

#ifdef TC_LOCKING
_myapp_lock = TC_CREATELOCK();
#endif

// ...
}
```

A.2.1.2 µC/OS-II Support

The maximum number of events supported by µC/OS-II is defined by the macro **OS_MAX_EVENTS**. This macro includes the number of semaphores, message mailboxes and any message queues being used by µC/OS-II. Two must be added to **OS_MAX_EVENTS** for each of the DeviceMate subsystems that are used; also, one must be added if any of the included subsystems use XTC.

In the subsystem code, follow the define of **OS_MAX_EVENTS** with a **#use "ucos2.lib"** statement. This ensures that the definition supplied outside of the library is used, rather than the default in the library. Also the inclusion of **ucos2.lib** must happen before the inclusion of any other libraries. For more information on the Dynamic C implementation of µC/OS-II please see the *Dynamic C User's Manual*.

The µC/OS-II lock function, **OSSemPend()**, takes as a parameter a pointer to an error status integer. This is a global dummy variable in **dm_app.lib**.

A.2.1.3 Locking Use

In general, the easiest way to make a subsystem safe for preemptive multitasking environments is to lock the tick and API functions on entry, and unlock them on exit. This way will be safe unless:

- the tick function is accessible from something other than the tick function chain (such as from one of the API functions)
- one of the API functions calls another of the API functions
- some other obscure way that is probably really, really bad

By default, the lock and unlock macros will expand to nothing.

If **MCOS** is defined (which it is if you are using µC/OS-II in Dynamic C), then the lock and unlock macros will expand to a default set of macros that use the native µC/OS-II semaphores. Note that these macros will provide binary semaphores only! If you expect to be able to lock multiple times from the same task and have it succeed, then you will have unpleasant surprises!

A.2.2 Critical Sections

A critical section of code is a part of the program that accesses shared data.

The critical section macros are: **`TC_ENTER_CRITICAL()`** and **`TC_EXIT_CRITICAL()`**. They take no parameters. Under µC/OS-II they map to **`OS_ENTER_CRITICAL()`** and **`OS_EXIT_CRITICAL()`**.

They should be used with caution as they disable interrupts.

A.2.3 Data Flow

The following diagram shows normal data flow between DeviceMate and target subsystems.

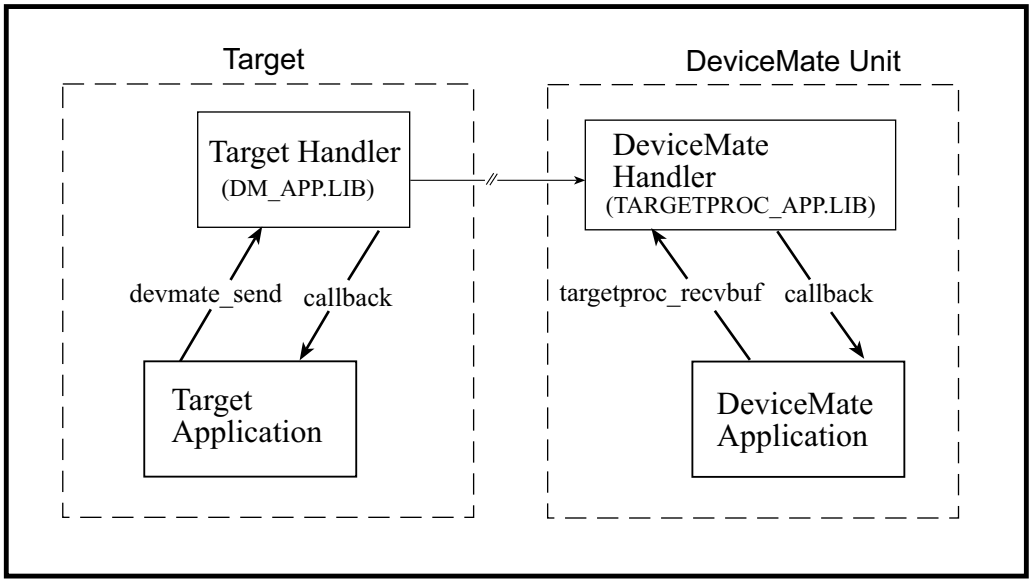


Figure 1. Data Flow Initiated by Rabbit Target

Typically, the target subsystem kicks things off by queuing a buffer to be sent to the DeviceMate. After the buffer has been transmitted, it is returned to the target subsystem by the subsystem-supplied callback. On the other side, the DeviceMate begins receiving the data into a DeviceMate-supplied buffer that was previously registered by a call to **`targetproc_recvbuf()`**. After the data has been completely received and verified, the callback supplied by the DeviceMate subsystem is called on the buffer.

This also works in the opposite direction, with the subsystem on the DeviceMate queuing a buffer to send to the target by calling **`targetproc_send()`** and the target receiving the data via the callback using the buffer that was previously registered by a call to **`devmate_recvbuf()`**.

A.3 The Callback Function

Each of the subsystems, on both the target and the DeviceMate, will have a callback function associated with it that must be defined as follows:

```
int callback (uint16 flags, uint8 type, unint8 subtype, uint16
   length, uint32 buffer, uint32 userdata);
```

PARAMETERS

`flags` A bitfield with the following values:

```
#define TC_RECEIVE       0x0001
#define TC_TXDONE        0x0002
#define TC_UNSUPPORTED   0x0004
#define TC_NOBUFFER      0x0008
#define TC_RESET         0x0010
#define TC_SYSBUF        0x8000
```

`TC_RECEIVE` and **`TC_TXDONE`** inform the subsystem what type of callback was made. If **`TC_RECEIVE`** is set, it is a receive callback: meaning that the packet in **`buffer`** is valid and should be handled quickly. It **`TC_TXDONE`** is set, it is a transmit callback and the packet in **`buffer`** was sent successfully.

If **`TC_UNSUPPORTED`** is set, the other side has not registered a callback for the requested type.

If **`TC_NOBUFFER`** is set, the other side has a registered callback, but has no free receive buffers at the moment. This implies that the packet referenced in the last **`TC_TXDONE`** callback was not received successfully.

If **`TC_RESET`** is set, it is requested that the subsystem reset itself to an initial state.

If **`TC_SYSBUF`** is set, the buffer given in the callback is a system buffer and will be returned automatically once the callback is finished.

`type` Identifies the subsystem with a unique value.

`subtype` On a receive callback, this value is the subtype of the packet. On txdone, this parameter is undefined.

`length` On a receive callback, this is the length of the received data in the buffer. On txdone, this is undefined.

`buffer` The physical address of the beginning of the buffer's header. Note that on a receive callback, the user data starts **`sizeof(_TCHeader)`** bytes after the location pointed to by **`buffer`**.

userdata The data that the subsystem specified in **devmate_send()** or **devmate_recvbuf()**.

RETURN VALUE

The callback function should return 0 for now—other return codes may be defined later.

A.3.1 Callback Registration

At startup, each subsystem must register their callback function with the following API call:

```
int devmate_registercallback ( uint8 type, uint16
   (*call_back)());
```

devmate_registercallback() will return **TC_SUCCESS** or **TC_ERROR**. Passing a **NULL** value as the function pointer will un-register the callback and shut down handling of the subsystem identified by **type**.

Subsystems that use XTC do not call **devmate_registercallback()** since XTC does it for them.

A.4 Buffer Management

There are two general classes of buffers. One set is owned by the system. These system buffers are used to notify a subsystem (running on either the target or DeviceMate) that there is a problem transmitting the data. The subsystem should copy out any important values and then return from the callback function as quickly as possible.

A.4.1 Subsystem Buffers

The second set of buffers is owned by the subsystems. When a subsystem on DeviceMate wants to send data, it places the data in a buffer and queues it to be sent by calling **targetproc_send()**. A subsystem running on the target would call **devmate_send()**.

Before a subsystem can receive data from the target handler it must register the buffer by calling **devmate_recvbuf()**. This must be done for each receive buffer that will be used.

Subsystems running on DeviceMate register buffers with the DeviceMate handler.

A.4.2 Queue and Buffer Routines

These routines will be used by subsystems that do not use XTC services.

_tc_get_buffer

```
extern faraddr _tc_get_buffer(faraddr_t *chain);
```

DESCRIPTION

Removes a buffer from the specified chain, and returns the long pointer to it, or 0 if there was no buffer available on that chain.

PARAMETERS

chain Identifies the buffer queue.

RETURN VALUE

0: The queue is empty

long *: Address of the buffer that was removed from the queue

LIBRARY

tc.lib

_tc_queue_buffer

```
extern int _tc_queue_buffer(faraddr_t *chain, faraddr_t
   buffer);
```

DESCRIPTION

This function adds a buffer to the end of a queue.

PARAMETERS

chain A pointer to the queue where you want to add the buffer

buffer Address of the buffer to add to the queue

RETURN VALUE

0: For all cases

LIBRARY

tc.lib

_tc_create_queue

```
int _tc_create_queue(faraddr_t* chain, faraddr_t buffer, long
   bufsize, int number);
```

DESCRIPTION

This function creates a new queue and may add buffers to it.

PARAMETERS

chain	The pointer to the queue
buffer	The address of the first buffer to add to the newly created queue.
bufsize	The number of bytes in the buffers that will be added to the queue
number	The number of buffers to add to the queue

RETURN VALUE

0: The queue is empty
!0: Pointer to last buffer in the queue

LIBRARY

tc.lib

_tc_empty

```
int _tc_queue_empty(faraddr_t* chain);
```

DESCRIPTION

This function checks the queue to see if anything is there.

PARAMETERS

chain	Pointer to a queue

RETURN VALUE

0: The queue is empty
!0: The queue is not empty

LIBRARY

tc.lib

A.5 Transmitting Packets API

A valid callback function, that can deal with **TC_TXDONE** callbacks, must be registered for the subsystem before calling **devmate_send()**.

devmate_send

```
int devmate_send(char type, char subtype, int length, long
   buffer, long userdata);
```

DESCRIPTION

This function will queue a packet for transmission. If the function returns success, the specified buffer should not be used until the **TC_TXDONE** callback is issued for it.

PARAMETERS

type	The subsystem type of this packet. This type must have a valid callback function registered or an error will result.
subtype	The subtype of this packet
length	The length of the user data in the packet. Note that this does not include any headers in the buffer.
buffer	The physical address of the buffer to send. This is the beginning of the buffer, not the user data area. The user data should start **sizeof(_TCHeader)** bytes after this physical address.
userdata	This is an area where a subsystem can store any useful data. This area is stored in the buffer's header when **devmate_send()** is called, and is returned in the callback.

RETURN VALUE

TC_SUCCESS: Packet was successfully queued
TC_ERROR: General error
TC_NOCALLBACK: No callback function was registered for this subsystem

LIBRARY

dm_app.lib

A.6 Receiving Packets API

A valid callback function, that can deal with **TC_RECEIVE** callbacks, must be registered for the subsystem before calling **devmate_recvbuf()**.

devmate_recvbuf

```
int devmate_recvbuf(char type, int length, long buffer, long
   userdata);
```

DESCRIPTION

This function adds the given buffer to the end of **type**'s receive queue. Note that when a **TC_RECEIVE** callback gives a packet to the subsystem, that buffer has been removed from the receive queue. To add the buffer back to the receive queue so that another packet can be received into it, call **devmate_recvbuf()**.

PARAMETERS

type	The subsystem type of this packet.
length	The length of the user area in the buffer. This does NOT include the **sizeof(_TCHeader**) bytes at the beginning of the buffer. If zero is specified as the length, no data will be saved on the incoming packets, and the callback will only specify the subtype and such. No matther
buffer	The physical address of the buffer to receive into. This is the beginning of the buffer, not the user data area. The user data should start **sizeof(_TCHeader)** bytes after this physical address.
userdata	This is an area where a subsystem can store any useful data. This area is stored in the buffer's header when **devmate_recvbuf()** is called, and is returned in the callback.

RETURN VALUE

TC_SUCCESS: Packet was successfully queued
TC_ERROR: General error
TC_NOCALLBACK: No callback function was registered for this subsystem

LIBRARY

dm_app.lib

Appendix B. Using XTC

Subsystems that desire a reliable data stream or datagram service will want to use XTC services: A transport layer over the target communications (TC) link layer.

XTC is based on some of the ideas of TCP, with some features deleted because of the point-point nature of the TC. Either the target or the DeviceMate can actively open a connection or listen for a connection. The actual XTC library is almost identical on both ends. Since XTC is mainly symmetric, the rest of this document applies equally to the target or the DeviceMate unit, unless otherwise noted. The word “peer” is used to denote the target processor (TP) or the DeviceMate unit (DMU) when the distinction is not important.

An established connection is referred to as a “channel,” in order to distinguish it from a TCP “socket.” Any subsystem (out of the 255 allowed) can make use of XTC services. The subsystem “subtypes” are used as selectors for a particular channel instance. One subtype, 0xFF, is reserved as a “broadcast” to all channels belonging to the given subsystem. 0x00 through 0x7F are reserved for direct subsystem use. This leaves a maximum of 127 channels per subsystem, with channel numbers ranging from 0x80 through 0xFE. This is a potentially large number of resources which may not be able to be supported simultaneously on the peer. Available resources will be limited to a configured number of channels. Each channel endpoint (analogous to TCP sockets) will specify channel state, buffers, RTT estimates and any other information associated with the channel. Each subsystem will have its own pool of channels so that each subsystem has a minimum guaranteed number of endpoints.

When an subsystem initializes, it will send a “broadcast” NEG (negotiate) flag to reset any resources on the peer. All NEG packets contain a reason code and other negotiated information. The initial NEG broadcast specifies a subsystem initialization code. The peer will respond with a NEG (with initialize-acknowledge code) so that the originator knows that the peer is available and synchronized.

NEG can also be sent to individual channels, in which case it embodies some of the properties of the TCP RST (reset) flag, but can also specify a reason code and other diagnostic information. Any protocol error will result in a NEG flag being sent in order to renegotiate the channel.

The application and subtype fields from the TC header are part of the logical XTC header. The data portion of each TC packet contains a 5-byte header which contains flags, receive window, sequence number and acknowledgment number. SEQ/ACK numbers are 16 bits each. Flags and window are compressed into a single byte. There are 4 bits for the possible flags (SYN, ACK, FIN, NEG) and the available receive window is encoded in the remaining 4 bits which is interpreted as **`floor(log2(windowsize))`**. If the window size is zero, then a zero code is used. The peer must always interpret a window field of zero as meaning a closed window (not a window of 1). This logarithmic coding of available window provides fine resolution when the window is nearly closed, but very coarse resolution when it is half- to fully open. To compensate for this, each peer keeps track of the right edge of the window, using the fact that no peer is allowed to contract the window edge to the left.

B.1 Library Support

XTC clients (i.e.,subsystems that use XTC) must include in their code:

```
#use "dm_xtc.lib" (for Rabbit-based targets)
#include "dm_xtc.h"   (for nonRabbit-based targets)
```

and in the companion subsystem on the DeviceMate unit there must appear the statement:

```
#use "targetproc_xtc.lib"
```

B.2 Data Structures

Each XTC client (both on the target and on the DeviceMate) must declare one instance of a **`XTCApp`** structure and at least one instance of a **`XTCChan`** structure. The latter structure must be initialized to zero by the subsystem. The **`XTCApp`** struct must have some of its fields initialized to appropriate values before the data structure is registered. Examples of doing this are in the subsystems that use XTC: **`dm_var.lib`**, **`targetproc_var.lib`**, etc.

B.2.1 Registration

On the DeviceMate unit, for each subsystem that uses XTC, an **`XTCApp`** structure must be registered using **`targetproc_xtc_register()`** before calling **`targetproc_init()`**.

On the target, for each subsystem that uses XTC, an **`XTCApp`** structure must be registered using **`devmate_xtc_register()`** before calling **`devmate_init()`**.

B.2.2 XTCApp Structure

This data structure is defined in **`tc_xtc.lib`**. It contains fields that must be filled in by any subsystem wishing to use XTC services.

```
typedef struct _XTCApp{

// Fields to set up before calling the register function
   uint8 appno;          // Application code
   uint8 numchans;       // Number of channels in following array
   uint16 txbufsize;     // Transmit buffer size w/out TC header
   uint16 rxbufsize;     // Receive buffer size w/out TC header
   XTCChan * chans;      // Channel array

// Fields initialized by the subsystem's _xxxx_init().
   uint8 numtxbufs;      // # of transmit buffers (usually 1,
                         // maybe 2 for high speed)
   uint8 numrxbufs;      // Number of receive buffers
   uint16 reqpacing;     // Our requested pacing (sent to peer).
   uint8 aflags;         // Subsystem flags as follows:
#define XTC_AF_READY 0x01      // Subsystem is ready to establish
                               // channels with peer
#define XTC_AF_BCASTNEG 0x02   // Broadcast NEG, waiting for
                               // a response
#define XTC_AF_SERVER 0x80    // This is a DeviceMate app struct
                               // (else TP)
#define XTC_AF_CONTROL 0x40  // DeviceMate unit: automatically
                             //   listen on channel 0 (control).
                             // Target processor: automatically
                             //   active open channel 0.
                             // If this flag is set in both, then
                             // the control channel will be opened
                             // whenever devmate_xtc_ready()
                             // or targetproc_xtc_ready()
                             // returns true.
#ifdef FUNCPTR_PROTOTYPES
   void (*tc_handler)(_xtcappptr app, _TCHeader * hdr, faraddr_t
    data);
#else
   void (*tc_handler)();     // Handler for app subtypes that
                             // are not XTC
#endif

// the rest of the fields in this structure are internal or
// initialized by devmate_xtc_init() or targetproc_xtc_init().
          .
          .
          .
} XTCApp;
```

B.2.3 XTC Configuration Macro

TC_MAX_XTC_APPS
Defaults to 4, which allows for the currently defined set of subsystems that use XTC services, namely TCP/IP, file system, web page variables and e-mail. This macro may be defined before the inclusion of **dm_xtc.lib**.

B.3 XTC API

_devmate_xtc_init

```
void _devmate_xtc_init(void);
```

DESCRIPTION

Initialize (or re-initialize) use of XTC layer.

LIBRARY

```
dm_xtc.lib (Rabbit-based targets)
dm_xtc.c (nonRabbit-based targets)
```

devmate_xtc_ready

```
int devmate_xtc_ready(XTCApp * app);
```

DESCRIPTION

Initiate a connection at the subsystem level, and test the connectivity state of the XTC subsystem. Returns positive when ready, 0 when trying to establish connectivity, or -1 if connectivity cannot be established.

This function initiates the process of establishing connectivity, by sending a broadcast NEG packet to the peer. If the peer responds with a broadcast NEG then the subsystem is considered to be connectable and the function returns positive. If the peer responds with an “app not registered” system message, then it is assumed that the subsystem is not defined on the peer, and -1 is returned. (This may be because the DeviceMate unit has not registered the same subsystem number.) Otherwise zero is returned and the subsystem should keep trying.

This function does not call **`devmate_tick(),`** so the subsystem should spin on this function using e.g.,

```
while (!devmate_xtc_ready(&MyApp)) devmate_tick();
```

PARAMETERS

`app`	This is a pointer to the subsystem’s state structure.

RETURN VALUE

≥**`1`**: Connection is established
`0`: Trying to establish connectivity
`-1`: Connectivity could not be established

LIBRARY

```
dm_xtc.lib (Rabbit-based targets)
dm_xtc.c (nonRabbit-based targets)
```

devmate_xtc_register

```
int devmate_xtc_register(XTCApp * app, uint16 rxlen, uint16
   txlen);
```

DESCRIPTION

This function must be called before **devmate_init()**, once only for each subsystem that uses XTC. The caller should initialize the necessary fields in the **XTCApp** struct, including the channel array as follows: either memset the entire channel struct to zero (in which case the parameters will be used to allocate suitable buffers etc.) or memset the channel to zero *except* for the rx and/or tx xbuf. If either rx or tx xbuf has a non-zero **blen** field, then it is assumed to be already initialized. rxlen/txlen must be at least 2 greater than the MSS.

PARAMETERS

app	This is a pointer to the subsystem's state structure.
rxlen	Default buffer size for receive buffer
txlen	Default buffer size for transmit buffer

RETURN VALUE

0: Success
-1: Error, too many subsystems
-2/**-3**: Buffer size error

LIBRARY

```
dm_xtc.lib (Rabbit-based targets)
dm_xtc.c (nonRabbit-based targets)
```

xtc_abort

```
int xtc_abort(XTCApp * app, uint8 chan);
```

DESCRIPTION

Force a channel to close, informing the peer if there is an active session. This generally causes a NEG packet to be sent, and forces the channel to the closed state. Both sides of the session will receive a "reset" reason code if they test the channel using **xtc_error()**.

If a session is established, but is immediately aborted with no transfer of data, then **xtc_estab()** will return -1, **xtc_error()** will return **XTC_NEGCODE_RESET**, and there will be no readable data.

PARAMETERS

app	This is a pointer to the subsystem's state structure.
chan	The particular connection that will be forcibly closed

RETURN VALUE

0

LIBRARY

tc_xtc.lib (Rabbit-based targets)
tc_xtc.c (nonRabbit-based targets)

xtc_aread

```
int xtc_aread(XTCApp * app, uint8 chan, char * data, uint16
   len);
```

DESCRIPTION

Read queued data, but only if the specified length of data is immediately available in the receive buffer.

PARAMETERS

app	Pointer to the subsystem's state structure.
chan	The XTC connection
data	Location of data to read
len	Number of bytes of data to read

RETURN VALUE

0: Not **len** amount of data available to read
-1: Receive buffer not large enough to hold **len** amount of data
-2: Channel is not readable
len: Data was successfully removed from the receive queue

LIBRARY

tc_xtc.lib (Rabbit-based targets)
tc_xtc.c (nonRabbit-based targets)

xtc_areadp

```
int xtc_areadp(XTCApp * app, uint8 chan, faraddr_t data, uint16
   len);
```

Same as **xtc_aread()**, except uses xmem address for data.

xtc_awrite

```
int xtc_awrite(XTCApp * app, uint8 chan, char * data, uint16
   len);
```

DESCRIPTION

Queue data for transmission, but only if there is sufficient space in the transmit buffer. The ‘a’ stands for “atomic.”

PARAMETERS

app	Pointer to the subsystem’s state structure.
chan	The XTC connection
data	Location of data to write
len	Number of bytes of data to write

RETURN VALUE

0: Data cannot be queued
-1: Transmit buffer not large enough
-2: Channel is not writable
len: Data successfully queued

LIBRARY

tc_xtc.lib (Rabbit-based targets)
tc_xtc.c (nonRabbit-based targets)

xtc_awritep

```
int xtc_awritep(XTCApp * app, uint8 chan, faraddr_t data, uint16
   len);
```

Same as **xtc_awrite()**, except uses xmem address for data.

xtc_close

```
int xtc_close(XTCApp * app, uint8 chan);
```

DESCRIPTION

Indicate that the channel is closed for writing. The channel will generally still be readable until the peer also issues **xtc_close()**. If the other side never closes its side of the channel, then it will be necessary to use **xtc_abort()** to force the channel closed. If the channel is passively opened, with no session established, then it will be quietly closed.

The return code may be safely ignored: it returns non-zero only if **xtc_close()** has already been called for this channel, or the channel does not exist.

Note that the channel may be closed yet readable, i.e. the subsystem did not finish reading data. **xtc_readable()** is a better test for a closed channel when it is important that all data be read

PARAMETERS

app	Pointer to the subsystem's state structure.
chan	The XTC connection

RETURN VALUE

!0: **xtc_close()** was already called or the channel does not exist

LIBRARY

tc_xtc.lib (Rabbit-based targets)
tc_xtc.c (nonRabbit-based targets)

xtc_closed

```
int xtc_closed(XTCApp * app, uint8 chan);
```

DESCRIPTION

Returns non-zero only if the channel is not defined, or fully closed. After calling **xtc_close()**, the subsystem can spin on this function to wait for complete termination of the session.

PARAMETERS

app Pointer to the subsystem's state structure.

chan The XTC connection

RETURN VALUE

0: Channel is fully closed
!0: Channel is not defined, or it is not fully closed

LIBRARY

tc_xtc.lib (Rabbit-based targets)
tc_xtc.c (nonRabbit-based targets)

xtc_error

```
int xtc_error(XTCApp * app, uint8 chan);
```

DESCRIPTION

Returns the most recent error code from the channel. The code will be zero if the channel has successfully closed after a previous session, or is currently open. Otherwise, a non-zero error code may be returned in the case that the channel was reset (NEG) by the peer. The actual error code will be one of a documented set of "official" error codes, or, if not one of these, will be a diagnostic code to assist developers.

PARAMETERS

app Pointer to the subsystem's state structure.

chan The XTC connection

RETURN VALUE

See "Description"

LIBRARY

tc_xtc.lib (Rabbit-based targets)
tc_xtc.c (nonRabbit-based targets)

xtc_estab

```
int xtc_estab(XTCApp * app, uint8 chan);
```

DESCRIPTION

This function checks the status of the connection. After an active or passive open, the subsystem should spin on **xtc_estab()** waiting for a non-zero return value. After **xtc_estab()** returns non-zero, the channel state should be examined using the other status functions. It is possible for **xtc_estab()** to return non-zero for a closed channel. This means that the channel became active, but was immediately reset by the peer. In this case, the return code will be negative, or this can be subsequently be tested using **xtc_error()**.

PARAMETERS

app	Pointer to the subsystem's state structure.
chan	The XTC connection

RETURN VALUE

0: Channel has been opened (active or passive) but not yet ready to transfer data

-1: Channel closed and an error occurred in the previous session (**xtc_error()** will return a non-zero error code in this case)

>0: Channel is readable, or successfully closed

LIBRARY

tc_xtc.lib (Rabbit-based targets)
tc_xtc.c (nonRabbit-based targets)

xtc_flush

```
int xtc_flush(XTCApp * app, uint8 chan);
```

DESCRIPTION

Forces XTC to send the queued transmit data as soon as possible. Normally, there is no need to call this function, as **xtc_write()** will send data at the optimum point in time considering link efficiency. However, sometimes the subsystem needs the timeliest response, in which case this function can be used. Note that XTC uses the Nagle algorithm by default, however this can be overridden by a channel flag or by use of **xtc_flush()**.

PARAMETERS

app	Pointer to the subsystem's state structure.
chan	The XTC connection

RETURN VALUE

0: Queued data was transmitted
-1: Channel not writable

LIBRARY

tc_xtc.lib (Rabbit-based targets)
tc_xtc.c (nonRabbit-based targets)

xtc_listen

```
int xtc_listen(XTCApp * app, uint8 chan, int (*handler)());
```

DESCRIPTION

Passively open the specified channel. After passively opening a channel, the subsystem must spin on **xtc_estab()**. When **xtc_estab()** returns non-zero, a session is either currently open, or a transaction was received, or the channel was reset by the peer. At this point, any results should be garnered (or the channel may be used in streaming mode) then the channel may be closed and re-used with a new **xtc_listen()** call after **xtc_closed()** returns true.

The handler function that may be passed as the third parameter has the following prototype:

```
int handler(XTCApp * app, XTCChan * c, xbuf * tx, long
   buf, uint16 len);
```

app and **c** are pointers to the subsystem and channel contexts (which may not need to be accessed if the handler is devoted to a single channel). **tx** is a pointer to the transmit buffer, in which a response is to be placed. **buf** and **len** specify the physical address and length of the request data.

A response is simple to add: the function **xbuf_append()** is called with **tx**, and the response data:

```
int xbuf_append(xbuf * cb, long data, uint16 dlen);
```

The handler function should return the result from **xbuf_append()**, i.e. the number of bytes successfully queued, or -1 if an error occurred. If -1 is returned, the session is aborted so that the peer can tell (by examining the result of **xtc_error()**) that the transaction failed.

The response should not be longer than **app->mss**. If it is longer, then the response will not be sent in a single segment. This should be transparent to the receiving peer apart from the longer time necessary, however, the intention of transaction mode is to allow quick request/response processing.

If no handler function is specified (i.e. **NULL** is passed as the third parameter to **xtc_listen()**), a default response of zero data is generated. The request data can still be obtained in the mainline **xtc_listen()** loop, after **xtc_estab()** returns non-zero. If not actually read, the request data is discarded when **xtc_listen()** is next called.

PARAMETERS

app	Pointer to the subsystem's state structure.
chan	The XTC connection
handler	Pointer to function that will handle transaction mode requests. If this pointer is **NULL**, requests will still be accepted, but responses will contain no data.

RETURN VALUE

0: Channel was opened passively
-1: Error, e.g., channel was already open or the channel number was not configured

LIBRARY

tc_xtc.lib (Rabbit-based targets)
tc_xtc.c (nonRabbit-based targets)

xtc_open

```
int xtc_open(XTCApp * app, uint8 chan);
```

DESCRIPTION

Open a new channel.

PARAMETERS

app	Pointer to the subsystem's state structure
chan	The XTC connection

RETURN VALUE

0: Success
-1: Channel already open or doesn't exist

LIBRARY

tc_xtc.lib (Rabbit-based targets)
tc_xtc.c (nonRabbit-based targets)

xtc_opts

```
int xtc_opts(XTCApp * app, uint8 chan, uint8 optnum, uint16
   value)
```

DESCRIPTION

Specify channel or subsystem options. **optnum** specifies the option number (**XTC_OPT_***) and **value** is a value appropriate for the specified option. Most options apply for the life of the subsystem, but some may only apply to the next or current session. Some options may apply to the subsystem as a whole, others to the specific channel number.

PARAMETERS

app	Pointer to the subsystem's state structure.
chan	The XTC connection
optnum	The option number. Actual options are: **XTC_OPT_SERVER**: subsystem to act as the DeviceMate **XTC_OPT_CONTROL**: subsystem automatically listens on control channel 0, if server; otherwise it actively opens the control channel **XTC_OPT_NONAG** : channel to bypass Nagle algorithm All options currently take a boolean parameter value: 0 = turn setting off, else turn it on.
value	An appropriate value for the specified option

RETURN VALUE

0: Option set successfully
-1: Error

LIBRARY

tc_xtc.lib (Rabbit-based targets)
tc_xtc.c (nonRabbit-based targets)

xtc_preread

```
int xtc_preread(XTCApp * app, uint8 chan, char * data, uint16
   len);
```

Same as **xtc_read()**, except that the data is not removed from the queue. This is a macro which invokes **xtc_prereadp()**.

xtc_prereadp

```
int xtc_prereadp(XTCApp * app, uint8 chan, faraddr_t data,
   uint16 len);
```

Same as **xtc_readp()**, except that the data is not removed from the queue.

xtc_read

```
int xtc_read(XTCApp * app, uint8 chan, char * data, uint16 len);
```

DESCRIPTION

Read queued data, to a maximum of **len** bytes.

Returns from 0 to len, or -1 if channel not in a readable state. A channel may be readable when closed, and indeed should not be re-opened until all queued data have been read, however if opened before that then the queued data will be discarded.

If data is NULL, then the data is merely removed from the queue.

This is a macro which invokes **xtc_readp()**.

PARAMETERS

app	Pointer to the subsystem's state structure.
chan	The XTC connection
data	Pointer to data to read
len	Number of bytes

RETURN VALUE

≥**0**: Number of bytes read
-1: Channel not readable

LIBRARY

```
tc_xtc.lib (Rabbit-based targets)
tc_xtc.c (nonRabbit-based targets)
```

xtc_readable

```
int xtc_readable(XTCApp * app, uint8 chan);
```

DESCRIPTION

This function returns positive if the channel is currently readable. This means that there is either unread data in the receive buffer, or that the channel is still open and able to receive more data from the peer. A channel can be readable even after it is closed. Any unread data in the receive buffer is only discarded when the channel is re-opened.

The return value also indicates the amount of data in the receive buffer. The number of bytes that may be read immediately (by calling **xtc_read()**) is obtained by subtracting 1 from the return value of this function.

PARAMETERS

app	Pointer to the subsystem's state structure.
chan	The XTC connection

RETURN VALUE

0: Channel is not in a readable state.

>0: Channel is currently readable, subtracting 1 from this return value is the number of bytes that may be read immediately (by calling **xtc_read()**).

LIBRARY

tc_xtc.lib (Rabbit-based targets)
tc_xtc.c (nonRabbit-based targets)

xtc_readp

```
int xtc_readp(XTCApp * app, uint8 chan, faraddr_t data, uint16
   len);
```

Same as **xtc_read()**, except uses xmem address for data.

xtc_writable

```
int xtc_writable(XTCApp * app, uint8 chan);
```

DESCRIPTION

This function returns positive if the channel is currently writable. This means that the channel is open (but not necessarily established), and **xtc_close()** has not yet been called.

The return value also indicates the number of bytes that may be immediately written to the transmit buffer (by calling **xtc_write()**). The amount of free space in the buffer is calculated by subtracting 1 from the return value.

Note that it is possible to write to a channel that is not established. There is a limit to the amount of data which may be buffered, however the buffered data will be sent immediately when a connection is established.

PARAMETERS

app	Pointer to the subsystem's state structure.
chan	The XTC connection

RETURN VALUE

0: Channel is not in a writable state.

>0: Channel is currently writable, this return value is the number of bytes that may be immediately written to the transmit buffer (by calling **xtc_write()**).

LIBRARY

tc_xtc.lib (Rabbit-based targets)
tc_xtc.c (nonRabbit-based targets)

xtc_write

```
int xtc_write(XTCApp * app, uint8 chan, char * data, uint16 len)
```

DESCRIPTION

Queue data for transmission. The length parameter is defined as unsigned, however in practice the specified length should be less than or equal to **INT_MAX** (32767) since the return value, which reflects the amount actually written, is signed.

If **xtc_writable()**-1 is positive, then that amount of data is guaranteed to be accepted by **xtc_write()** if there are no intervening XTC calls.

xtc_write() may be called immediately after **xtc_open()**, providing the amount of data is less than or equal to **app->mss.**

This is a macro which invokes **xtc_writep()**.

PARAMETERS

app	Pointer to the subsystem's state structure.
chan	The XTC connection
data	Pointer to data to read
len	Number of bytes to queue

RETURN VALUE

≥**0**: Number of bytes queued (maximum number of bytes is **len**)
-1: Channel not writable

LIBRARY

tc_xtc.lib (Rabbit-based targets)
tc_xtc.c (nonRabbit-based targets)

xtc_writep

```
int xtc_writep(XTCApp * app, uint8 chan, faraddr_t data, uint16
   len);
```

Same as **xtc_write()**, except uses xmem address for data.

Index

Symbols

A

B

C

D

E

F

H

I

L

M

N

O

P

R

S

T

U

W

X

Z