APPLICATION NOTE

AN421

Remote Program Update

Updating deployed firmware[1] without having physical access to the device running it is a very useful feature. Remote updating saves time, money and resources. The Remote Program Update Library may be used to add this feature to any application running on a supported Rabbit-based device.

1.0 Hardware and Software Requirements

Remote Program Update is supported on Rabbit-based devices running Dynamic C 10.54 or later and meeting the following requirements:

- Rabbit 4000 or newer processor
- Firmware runs from fast SRAM
- Device has mass storage: NAND, serial flash, mini SD card

1.1 Hardware Requirements

The following Rabbit core modules and boards may be used with the Remote Program Update library:

- RCM4200
- RCM4300 Series
- RCM4400W
- RCM5400W Series
- RCM5600W Series
- RCM5750/60[2]
- BL4S100 Series
- BL4S200
- BL5S220
- Rabbit 6000 Boards

1.2 Software Requirements

The Remote Program Update library and samples that illustrate its use are automatically installed with the installation of Dynamic C 10.54 or later.

1. The term “firmware” is used in this document and the Remote Program Update library to refer to the code running on the Rabbit-based target. In the suite of documentation available with Dynamic C, this code is also called: the software, an application, a program, a sample, a sample program and various other synonyms.
2. Even though its firmware runs from parallel flash, RCM5750/60 uses a custom loader included with Dynamic C 10.56 to perform firmware updates.

An initial program must be loaded onto the hardware via the programming cable (using either Dynamic C or the Rabbit Field Utility (RFU)) before the remote update feature can be used to then install a firmware .bin file. The remote update library will only accept firmware compiled with Dynamic C 10.54 or later.

2.0 Power-Fail Safe Updates

Updating firmware in Dynamic C versions 10.54 to 10.60 has a failure window of around 10 seconds, the time it takes to write the firmware to the boot flash. Dynamic C 10.62 introduces a power-fail safe option for all serial boot boards, including:

- RCM4300 Series
- BL4S100 Series
- BL4S200
- RCM5600W (firmware limited to 510KB)
- RCM5650W
- Rabbit 6000 Boards (firmware limited to 510KB when equipped with 1MB serial boot flash)

Power-fail safe updates reserve space on the serial boot flash for two copies of firmware and ensures that during an update the original firmware remains bootable until the updated firmware is completely written.

3.0 Support Information

All programs compiled with Dynamic C 10.54 and later will contain hooks for implementing Remote Program Update. This includes programs compiled as .bin files and also .c files compiled and downloaded directly to the Rabbit-based target via the programming cable.

Any program compiled with Dynamic C 10.54 may use the application programming interface (API) supplied by the program update library (`/Lib/.../RemoteProgramUpdate/board_update.lib`) to perform remote, on-board firmware updates. Partial firmware updates are not supported.

To run the Remote Program Update sample programs or to use its functionality in your existing application, there are several things to consider, including creating a bin file, storage selection, and upload and download methods. The rest of this section discusses these topics.

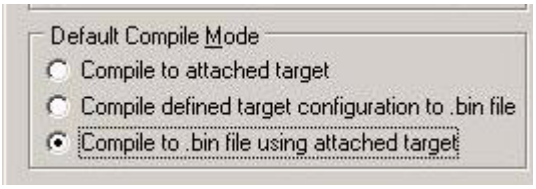
3.1 Creating Remote Program Update-Enabled .Bin Files

The firmware is both stored and installed as a .bin file. The required .bin file is created within Dynamic C in one of two ways:

1. The Compile Menu:

2. The Compiler Tab of the Options | Project Options Menu:

The Compiler tab contains the setting for the "Default Compile Mode":



This setting controls the behavior of the "F5" compile option. Notice that the "Default Compile Mode" has two options for compiling to a .bin file. See the *Dynamic C User's Manual* for more information on the defined target configuration option.

3.2 Storage of .Bin Files

The board update library provides a single, standard API (buTempCreate/Write/Close) that can be configured at compile time to store a temporary copy of the firmware .bin file in one of the following locations:

- FAT Filesystem on Serial Data Flash, Serial Boot Flash, NAND or mini SD card
- Serial Data Flash (direct storage without FAT filesystem)
- Serial Boot Flash
- Secondary firmware location for power-fail safe updates

The selection of a temporary storage location, if one is desired, must be made at compile time using configuration macros. See Section 5.2.1.2 for a list of these macros. Not all memory options are available on all supported hardware. See Table 1 to determine the temporary storage options available for specific hardware

Table 1. Options for Temporary Storage of Firmware

Rabbit-Based Hardware	Temporary Storage Locations
RCM4200	FAT filesystem, serial data flash
RCM4300 Series*	FAT filesystem, serial boot flash, secondary location
RCM4400W Series	FAT filesystem, serial data flash
RCM5400W Series	FAT filesystem, serial data flash
RCM5600W	Direct write to boot section of serial boot flash, secondary location (limited to 510KB firmware size)
RCM5650W	FAT filesystem, serial boot flash, secondary image
RCM5750/60	Serial data flash
BL4S100 Series	FAT filesystem, serial boot flash, secondary location
BL4S200 (RCM4310)	FAT filesystem, serial boot flash, secondary location
BL5S220 (RCM5400W)	FAT filesystem, serial data flash

* The RCM4300 series uses an SD card that supports FAT. Sample program bootchk.c demonstrates this configuration with Remote Program Update functionality.

Some helper functions are provided to put the firmware in the temporary storage location. See Section 5.2.3 for more information on the helper functions.

As listed in Table 1, the boot section of the serial boot flash is the only “temporary” storage location available on an RCM5600W for firmware larger than 510KB. Although the boot section is not really a temporary storage location, it is used as one because there is no mass storage on this module and the non-bootable portion of the boot flash is too small to hold the firmware image. If your firmware is less than 510KB, it is safer to use the secondary location and power-fail safe updates available in Dynamic C 10.62.

It is not mandatory to use one of the temporary storage locations (except for power-fail safe updates); there are other local storage options. For example, the firmware may reside in a RAM buffer. It is mandatory to “open” the firmware using one of the buOpenFirmwareXYZ functions in order to install. See Section 5.2.4 and Appendix A for more information on the open firmware functions.

3.3 Storage of .Bin Files for Power-Fail Safe Updates

On boards with a serial boot flash, the "secondary location" is the preferred option for storing new firmware images. It is virtually identical to the old "serial boot flash" option used for non-power-fail-safe updates but always writes to the secondary (non-boot) area of flash. After verifying the new firmware, installation is accomplished by a fast update to the appropriate valid marker. If the update fails at any point due to loss of power, the board will still boot from the previous version (left intact during the update).

3.4 Upload / Download Methods

The method used for transferring the firmware to the target is application specific. Remote Program Update comes with sample programs that illustrate the following methods:

- HTTP Server (using RabbitWeb enhancements) - `upload_firmware.c`
- HTTP Client - `download_firmware.c`
- FTP Client - `download_firmware.c`
- TFTP Client - `tftp_get_firmware.c`

In addition to the above communication methods, the sample program `bootchk.c` demonstrates how to check the FAT filesystem on an SD card for a firmware update. No network connection is needed since the firmware .bin file is locally accessible to the running program.

There are many ways to transmit the firmware to the Rabbit. The examples provided with Remote Program Update demonstrate some common ones, but you are not limited to these.

You may decide to provide some other communication protocol (for example Xmodem over a serial port). The decision will likely be based on existing infrastructure and the current functionality of your application. For example, if the deployed software is already running a web server with file upload capability, it makes sense to use it for firmware updates. See the TFTP sample (`tftp_get_firmware.c`) to use as a template for adding an additional communication protocol; e.g., porting an Xmodem implementation.

3.5 Real-World Use of Remote Program Updating

The features of the Remote Program Update library are not simple to demonstrate without additional infrastructure. This section describes how to integrate the Remote Program Update functionality into an existing, shipping product.

3.5.1 Check SD Card for Updates

The `bootchk.c` sample demonstrates checking an SD card for a firmware update. The check is done at boot time for a given filename (`bootchk.bin` in the sample), and the firmware is installed if it is newer than what is currently running.

To perform an update, a technician (or even a customer) would power off the device, insert the SD card with new firmware, boot the device and wait for the update to complete, then power off again and remove the card.

3.5.2 Check Web/FTP Server for Updates

The `ftp2fat.c` and `http2fat.c` samples show how to download a file from an FTP or web server to the FAT filesystem.

An update infrastructure based on these samples might be to have your Internet-enabled Rabbit application connect to a CGI script on a web server to ask if there is a firmware update available. The Rabbit would send its current version number and serial number, and the CGI script would reply with an URL for the new firmware if that particular device should update itself. If the Rabbit gets a reply indicating new firmware is available, it would download the new image, verify it using the Remote Program Update library, and then install and reboot.

The API functions `buDownloadInit()` and `buDownloadTick()` provide an alternate method for downloading from a web or FTP server using the configured temporary storage location.

3.5.3 Allow Uploading of New Firmware Via Web Browser

A Rabbit application with an existing web server (with or without RabbitWeb) can integrate the code from the `upload_firmware.c` sample. That sample uses the buTempCreate/Write/Close API to store the firmware on an unused portion of the serial boot flash, or on the FAT filesystem.

3.5.4 Updates Over a Serial Port

If you have a serial console, you could add an Xmodem upload feature to send new firmware serially, and store it using the buTempCreate/Write/Close API, or the FAT API to save it in the FAT filesystem. After the upload completes, you would want to verify the firmware, and then prompt the user to initiate an update.

The current release of Dynamic C does not include Xmodem receive code. You would have to port an existing Xmodem implementation to the Rabbit. See the Wikipedia entry on the Xmodem protocol for links to public domain source code.

4.0 Running Sample Programs

Before running any Remote Program Update specific sample programs, run the sample program `pong.c`, located in the `Samples/` directory relative to the Dynamic C installation in order to verify that your board is connected properly and communicating with Dynamic C. After running `pong.c`, run one of sample programs listed in Section 4.1. These samples demonstrate transmitting firmware from a remote location to the Rabbit, as well as verifying and then installing the new firmware. Sample programs listed in Section 4.2 demonstrate FTP or HTTP to transmit a .bin file and store it in the FAT filesystem; these samples do not make use of the Remote Program Update API.

Instructions are listed at the top of each sample program file. Read these instructions, as they will detail any infrastructure requirements. Also, read the configuration section of the program so you can customize the code to fit your hardware/software situation.

4.1 Remote Program Update Sample Programs

The sample programs that demonstrate the functionality of the board update library are located in the `Samples/RemoteProgramUpdate/` directory relative to the Dynamic C installation. Each of the samples demonstrate a different communication method for transmitting firmware to a Rabbit-based target. Most of the samples in the bulleted list below require a #define of the macro that controls selection of the temporary storage location. See Section 5.2.1.2 for details on the storage location macros.

- `bootchk.c` - This sample is designed for the RCM43xx series. It demonstrates how an application can check for a firmware update on an SD card and install it if it is newer than what is currently running. The sample requires the use of the FAT filesystem on the SD card. The easiest way to write firmware to the SD card is to use a card reader (some laptops have them built-in), but the sample programs listed in 3.2 may also be used.
- `download_firmware.c` - demonstrates running an HTTP client on the Rabbit. It can be easily modified to run an FTP client instead by changing the macro `FIRMWARE_URL` to point to an FTP server. This sample requires a server of the appropriate type that provides access to the named firmware .bin file.
- `firmware_info.c` - retrieves information about the currently running firmware and displays it to the Stdio window. The information retrieved is listed in Section 5.3.3. This program applies to all Rabbit-based boards, thus can be found in the top-level `Samples/` directory.
- `firmware_report.c` - when compiled to RAM, reports on the firmware installed on the boot flash. If BU_ENABLE_SECONDARY is defined in the Project Options, reports on when the bootloader was installed, and displays information on the boot and secondary firmware images on the flash.
- `tftp_get_firmware.c` - demonstrates running a TFTP client on the Rabbit. This sample requires a TFTP server that provides access to the named firmware .bin file.
- `upload_firmware.c` - demonstrates running an HTTP server on the Rabbit that will display a password-protected, form-based web page allowing clients to upload a .bin file. To view the results of running this program you will need a web browser on the same network as the Rabbit.
- `verify_firmware.c` - shows how to check the CRC32 of firmware stored on the boot flash and the firmware currently running from RAM. Useful to detect an overwritten flash, or a program error that has corrupted code or constants in the running program. This program works on all boards.

4.2 TCP/IP Sample Programs

There are sample programs demonstrating FTP and HTTP clients. They are located relative to the Dynamic C installation directory at: `/Samples/tcpip/ftp/` and `/Samples/tcpip/http/`.

- `ftp2fat.c` - demonstrates use of the FTP client library and ftp2fat helper library to copy files from a remote FTP server and save them to the FAT filesystem on the Rabbit.
- `http2fat.c` - demonstrates use of the HTTP client library and http2fat helper library to copy files from a remote web server and save them to the FAT filesystem on the Rabbit.
- `http_client.c` - demonstrates use of the HTTP client library to copy files from a remote web server and display them on the Stdio window.
- `http_upld.c` / `httpupld2.c` - These sample programs demonstrate HTTP file upload.

4.3 FAT and Serial Flash Sample Programs

There are several sample programs listed here that may be of use during the development/debug process.

- `FAT_shell.c` - presents a DOS/UNIX-like shell to access the FAT filesystem on a memory device. The FAT device may be partitioned and/or formatted by this program. Successfully running this program can establish and/or verify a functioning FAT partition.
- `sflash_inspect.c` - utility for inspecting a serial flash chip in raw mode.
- `sdflash_inspect.c` - utility for inspecting an SD card in raw mode.

5.0 Adding Remote Program Update Functionality to Existing Code

This section identifies the code needed to add Remote Program Update functionality to existing applications.

5.1 Code Overview

As mentioned previously, all programs compiled with Dynamic C 10.54 or later contain information that allows for the use of Remote Program Update functionality. It is up to the software programmer whether or not to use that functionality. In order to use it, a number of steps will typically occur in the program code. The following pseudo-code lays out what may be added to existing programs to produce firmware that allows remote updating.

The bold text in the pseudo-code identifies the topics that are specific to the Remote Program Update functionality.

The rest of the pseudo-code (non-bold text) identifies tasks that support the Remote Program Update functionality in terms of handling the firmware image before it is selected for use by the remote update library.

```
Configuration
   If firmware is remote, select temporary storage location if desired:
      secondary firmware image (for power-fail safe updates), FAT,
      serial data flash, serial boot flash or direct write
   If firmware is remote, select communication protocol
      http, ftp, tftp, or user-supplied other (e.g., Xmodem)
   Provide Remote Location of Firmware
      (for download only)
   Include appropriate libraries
      board_update.lib plus others based on above choices
Initialization
   Call initialization function(s)
      specific to communication protocol & storage location
      selected (e.g., tftp_init())
Transmit Firmware
   Activate communication method
      e.g., http_handler(), tftp_tick(), etc.
   Store firmware in selected local location
Select Firmware
   Firmware is in temp storage location or some other local storage
Verify Firmware
```

Power-Fail Safe Updates (DC 10.62+)	Non-PFS Updates (DC 10.54+)
Install Firmware **Update marker to promote secondary image to new boot image (about 10-50ms)**	Activate Safeguard Measures Optional: recommended tasks, such as disabling power button **Install Firmware** **Copy firmware image from local location to boot flash (about 10-15 seconds)**

```
Reboot System
   Run the new firmware
```

The exception to the above sequence of tasks involves the RCM5600W. Power-fail safe updates are an option for firmware smaller than 510KB. For larger firmware (up to 1MB), the only temporary storage location option is the boot portion of the serial boot flash (BU_TEMP_USE_DIRECT_WRITE). After the firmware has been written to this location, it should be verified and then it can be run by rebooting the system. The firmware does not need to be "Installed" because it is already in the boot flash, but for future compatibility, it is advisable to call the install function anyway. If the "Verify Firmware" task fails, the function `buRestoreFirmware()` must be called to restore the firmware that is running in fast RAM back to the boot portion of the serial boot flash. Otherwise, the Rabbit device would fail to boot after reset and be unreachable remotely.

5.2 Code Details

This section explains and identifies the Dynamic C code that implements the above pseudo-code.

5.2.1 Configuration

If the new firmware is located remotely, there are two main configuration options that must be selected at the beginning of the firmware code, before the inclusion of the Remote Program Update library:

- Communication Method
- Temporary Storage Location

Along with these two configuration options, if the application will be performing a file download, the name and location of the firmware .bin file might need to be known by the communication method during compile-time configuration. The exception to this would be run-time knowledge gained through something like a CGI script.

5.2.1.1 Communication Method

The communication method for transmitting firmware to your Rabbit-based target must be selected. There are many methods to choose from. Sample programs are provided to illustrate several common ones:

- HTTP Server - `upload_firmware.c`
- HTTP Client - `download_firmware.c`
- FTP Client - `download_firmware.c`
- TFTP Client - `tftp_get_firmware.c`

If your application does not already use a TCP-based network interface and you want to use a communication method that requires one, the following library must be included in your firmware code:

```
#use "dcrtcp.lib"
```

In addition, some important configuration macros exist for the network protocols and communication methods provided with Dynamic C. More information on the available options and requirements is found in the *Dynamic C TCP/IP User's Manual, Vols. 1 and 2*.

5.2.1.2 Temporary Storage Location

For boards that support power-fail safe updates, define BU_ENABLE_SECONDARY in the Project Options. There's no need to define a BU_TEMP_USE_XXX macro -- the Remote Program Update API will store the new firmware in the secondary location on the serial boot flash.

For non-PFS updates, firmware that has been transmitted to the Rabbit-based target may be temporarily stored and verified before it is installed in the boot section of memory. The Remote Program Update API recognizes four temporary locations to store the firmware. The location is determined at compile time by a #define of one, and only one, of the following macros:

- `BU_TEMP_USE_DIRECT_WRITE` - boot portion of the serial boot flash. This is the only method supported for firmware larger than 510KB on the RCM5600W.
- `BU_TEMP_USE_FAT` - FAT filesystem
- `BU_TEMP_USE_SBF` - unused portion of the serial boot flash located between the boot firmware and the UserBlock and System ID Block
- `BU_TEMP_USE_SFLASH` - serial data flash. The starting page for storage of the firmware is defined by `BU_TEMP_PAGE_OFFSET`.
- `BU_TEMP_USE_SECONDARY` - secondary location reserved on the serial boot flash for power-fail safe updates

It is not necessary to use one of the temporary storage locations; for other storage options see Section 5.2.4. If the firmware .bin file will not be placed in one of the temporary storage locations, the file's location is specified by a call to one of the buOpenFirmwareXYZ functions. See the function descriptions for `buOpenFirmwareBoot`, `buOpenFirmwareRAM` and `buOpenFirmwareSFlash` for details on these other firmware storage locations.

5.2.1.3 Provide Remote Location of Firmware

To download firmware, the communication method needs to know where to find it. In the following code snippet from `tftp_get_firmware.c`, the remote location of the firmware is defined as macros to pass to the initialization/setup function of the communication method selected, in this case TFTP. (These macros are not used by the remote update library.)

```
// Running TFTP client on Rabbit
#define TFTP_SERVER  "10.10.6.100"
#define TFTP_FILE "firmware.bin"
```

If you are running an HTTP or FTP client on the Rabbit, you can call `buDownLoadInit()` and `buDownLoadTick()` in place of the buTempCreate/Write/Close functions. The buDownLoadInit/Tick functions require the server address and name of the .bin file to be passed as one parameter, such as:

```
// Running an HTTP client
#define FIRMWARE_URL "http://example.com/firmware.bin”
```

```
// Running an FTP client
#define FIRMWARE_URL\
   "ftp://username:password@example.com/path/firmware.bin”
```

5.2.1.4 Include Libraries

Using FAT or the serial data flash without FAT require libraries specific to those locations. (The use of FAT offers some additional configuration options.)

```
#ifdef BU_TEMP_USE_SFLASH
   #use "sflash.lib"
#endif
#ifdef BU_TEMP_USE_FAT
   #use "fat.lib"
#endif
```

Libraries that provide the API for Remote Program Update and the selected communication method must be included. The remote update library must come after the HTTP and FTP client libraries.

```
// HTTP server running on Rabbit
#use “http.lib”

// HTTP client running on Rabbit
#use “http_client.lib”

// FTP client running on Rabbit
#use “ftp_client.lib”

// TFTP client running on Rabbit
#use “tftp.lib”

// Compile in remote update API. Must come after http_client.lib and ftp_client.lib
#use "board_update.lib"
```

5.2.2 Initialization

There are software components that must be initialized before use.

5.2.2.1 Network

If your application does not already contain a network interface, you need to initialize the stack in order to communicate on a network. During the development/debug cycle, you would typically call `sock_init_or_exit()` and replace it with a call to `sock_init()` for firmware that is ready to deploy.

5.2.2.2 Communication Protocol

Network communication protocols require that the network be initialized first (see Section 5.2.2.1).

The Remote Program Update sample programs illustrate the initialization of protocols for both uploading and downloading the firmware:

- HTTP Server - `http_init()`
- HTTP, FTP and TFTP Clients - `httpc_init()`, `ftp_client_setup()`, and `tftp_init()`, respectively.

5.2.2.3 Temporary Storage Location

Only the FAT filesystem requires initialization. To initialize the FAT filesystem, call `fat_Automount()`.

The other two storage locations (serial data flash, serial boot flash) do not require a call to an initialization function but are handled in the program update library.

5.2.3 Transmit Firmware to Local Storage

In order for a firmware update to occur, the new firmware image must be stored locally, i.e., a memory device directly accessible by the application. The local memory may be one of the temporary storage locations provided.

Storing the firmware in a temporary storage location has three basic software components:

- Open/create temporary storage location
- Write firmware to the location
- Close temporary storage location

These three things are accomplished with the storage-independent API functions: `buTempCreate`(), `buTempWrite`() and `buTempClose`().

Two additional API functions may be used instead if you are downloading from a web or FTP server: `buDownLoadInit()` and `buDownLoadTick()`. These functions call the buTempCreate/Write/Close functions.

The function `buTempWrite()` is used in conjunction with the tick function of the selected communication method to transfer the firmware .bin file and write it to the temporary storage location.

The following code illustrates this code sequence/relationship. The code has been stripped of the error checking that exists in the program file in order to focus on how the firmware .bin file is transferred using TFTP and then written to the temporary storage location selected earlier in the program with one of the BU_TEMP_USE_* macros.

Program Name: tftp_get_firmware.c

```
...
tftp_init(&ts);
while (buTempCreate() == -EBUSY);
while ((result = tftp_tick(&ts)) >= 0) {
   if (ts.buf_used){
      offset = 0;
      while (offset < ts.buf_used){
         result = buTempWrite( &buffer[offset], ts.buf_used - offset);
         offset += result;
      }
      ts.buf_used = 0;
   }
   if (!result)          // this was the last block of data
      break;             // exit
}
if (!result){
   printf("Download completed\n");
   while (buTempClose() == -EBUSY);
}
```

A state structure is initialized prior to the call to tftp_init(), as detailed in the function description for tftp_init() and illustrated in tftp_get_firmware.c. The field "buf_used" in the TFTP state structure is the number of bytes transmitted to or received from the TFTP server. As you can infer from the above code, this field is updated in the tick function and then used to determine the amount to write to the temporary storage location.

After the above code is executed, the firmware will be in a staging area where it can be verified before it is installed in the boot area of memory.

5.2.4 Select Firmware

After using buTempCreate/Write/Close to store the .bin file in a temporary storage location, the firmware image must be selected by calling the non-blocking function buOpenFirmwareTemp() before it can be verified and installed.

```
i = 0;
do {
   result = buOpenFirmwareTemp(BU_FLAG_NONE);
} while ( (result == -EBUSY) && (++i < 20) );
```

If a temporary storage location is not being used, select the firmware image to verify and install by calling one of the other buOpenFirmwareXYZ functions: buOpenFirmwareBoot(), buOpenFirmwareFAT(), buOpenFirmwareRAM(), buOpenFirmwareRunning() and buOpenFirmwareSFlash(). These functions are necessary when the firmware is not located in the staging area created by calling buTempCreate/Write/Close.

The sample program bootchk.c demonstrates using buOpenFirmwareFAT().

5.2.5 Verify Firmware

The firmware should be verified before it is installed. The verification process consists of:

- Checking the CRC-32 on the firmware image to confirm that the file is not corrupted.
- Confirming that the board type the firmware was compiled for matches the target hardware.
- Confirming that the firmware image was compiled for flash.
- For power-fail safe updates, confirming that firmware is compatible with the bootloader.

If the install function determines that verification has not taken place, a call will be made to `buVerifyFirmwareBlocking()` before the firmware is installed. If you do not want to verify the firmware, the verification requirement can be overridden by setting the correct bit (i.e., `BU_FLAG_NOVERIFY`) in the flags parameter passed to the buOpenFirmwareXYZ function. Note that it is dangerous to circumvent verification. If the firmware is corrupted, it could lead to an unbootable target.

Since the verification process may take a significant amount of time, a non-blocking verification process is also available: `buVerifyFirmware()`. This is the function demonstrated in the provided program update sample programs.

5.2.6 Activate Safeguard Measures (Non-PFS)

Prior to starting the firmware install process, there are several preparations to consider. Because a non-power-fail safe install can result in an unreachable target if the process is interrupted, you should enact as many safeguards as possible. For example if you have a display attached to the Rabbit-based target, you could show an update-in-progress message.

Listed here are some other safeguard measures to consider:

- Disable the power button
- Use LEDs or display screen to give install status
- Notify remote server that firmware install attempt about to begin
- If running µC/OS-II, halt other tasks

5.2.7 Install Firmware (Power-Fail Safe)

Once firmware stored in the secondary location has been verified, the install process goes quickly. A typical install is complete in just 20 milliseconds -- the time it takes to update the marker for the new firmware image.

5.2.8 Install Firmware (Non-PFS)

The amount of time it takes to complete the install process depends on firmware size and processor speed. It may take only a few seconds, but for a very large firmware image, it will take longer.

The install function, `buInstallFirmware()`, will install the firmware image to the boot flash. It may first attempt to verify the firmware image as described in Section 5.2.5.

5.2.9 Restart System

After the new firmware has been successfully installed, it can be run by forcing a watchdog timeout to reset the board. This is done by calling `forceWatchdogTimeout()`.

5.3 Configuration Macros, Flags and Data Structures

This section lists all of the new configuration macros used in board_update.lib, as well as some additional configuration macros that may be useful for applications using the Remote Program Update functionality.

5.3.1 Remote Program Update Configuration Macros

One and only one of the BU_TEMP_USE_* macro may be #defined in the firmware to specify a temporary storage location:

- BU_ENABLE_SECONDARY - define this macro in the Project Options to enable power-fail safe firmware updates. Uses a "secondary" bootable location on the serial flash as the staging area for verifying new firmware.
- BU_ENABLE_MINILOADER - define this macro in the Project Options to include the custom loader used to install firmware on the RCM5750/RCM5760
- BU_TEMP_USE_FAT - selects the FAT filesystem as a staging area for verifying and installing the firmware.
- BU_FAT_TIMEOUT (10 ms) - sets the amount of time the Remote Program Update library functions will block while waiting for the FAT library to complete a call. This macro defaults to 10 ms. The valid range is from 1 to 32,000 ms.
- BU_TEMP_FILE ("a:firmware.bin") - sets the name for the firmware .bin file that will be stored in the FAT filesystem when the FAT is used as the temporary storage location. It defaults to “a:firmware.bin”.
- BU_TEMP_USE_SBF - selects an unused portion of the serial boot flash as a staging area for verifying and installing the firmware. Consider using BU_ENABLE_SECONDARY (power-fail safe firmware updates) instead, if you can dedicate space to the second firmware image.
- BU_TEMP_USE_SFLASH - selects the serial flash as a staging area for verifying and installing the firmware.
- BU_TEMP_USE_DIRECT_WRITE - selects the boot portion of the serial boot flash as a staging area for verifying and installing the firmware.
- BU_TEMP_USE_SECONDARY - selects secondary location reserved on the serial boot flash for verifying and installing the firmware in power-fail safe updates.
- BU_TEMP_PAGE_OFFSET - designates the starting page number on the serial flash. This macro defaults to 0 if BU_TEMP_USE_SFLASH is defined. To override the default include a #define of BU_TEMP_PAGE_OFFSET in your application.
- BU_TEMP_USE_DIRECT_WRITE - write new firmware directly to the boot image on the serial boot flash instead of a temporary staging area. Not recommended since a loss of power at any point during the download/verify process will leave the board unbootable. Consider using BU_ENABLE_SECONDARY (completely power-fail safe) instead, if you have room for two firmware images on the serial boot flash.
- BOARD_UPDATE_DEBUG - If defined, functions will be debuggable (e.g., you can set breakpoints and single-step into them).
- BOARD_UPDATE_VERBOSE - If defined, causes status and debug information to be displayed in the Stdio window.
- MAX_FIRMWARE_BINSIZE - For serial boot flash boards only, sets the maximum allowable size of the firmware (BIN) image. If not set by the user then a default maximum firmware size is set based on

the size of the serial boot flash. If a user-set value is too large for the serial flash and enabled RPU options then a warning is issued and the macro is redefined to the default maximum firmware size.

If used, the optional _FIRMWARE_* macros must be defined in the Defines tab of the Options | Project Options menu. Set these macros to have the information embedded into the firmware, and accessible to the Remote Program Update API.

- _FIRMWARE_NAME_ - This macro is a string, up to 19 printable characters, null-terminated.
- _FIRMWARE_VERSION_ - This macro is a 16-bit word (default = 0x0000). Its primary use is to allow a program to determine if an update is needed. This is demonstrated in the sample program Samples/RemoteProgramUpdate/bootchk.c.

  In bootchk.c, _FIRMWARE_VERSION_ is treated as a BCD (binary-coded decimal) value and printed as:

  ```
  printf("%u.%02x", _FIRMWARE_VERSION_ >> 8, _FIRMWARE_VERSION_
  & 0xFF)
  ```

- _FIRMWARE_TIMESTAMP_ - This macro is a 32-bit value signifying the number of seconds since 1/1/1980. Defaults to using the system time at compile time (_DC_GMT_TIMESTAMP_), can override to get repeatable firmware images (same .bin built from project/source files regardless of compile date/time).

5.3.2 Flags Parameter

The “flags” parameter is passed to the buOpenFirmwareXYZ functions. It is a bitmask for user-settable flags. The currently supported flags are:

- BU_FLAG_NONE - no flags set
- BU_FLAG_NOVERIFY - do not perform the pre-install verification. This is not recommended because it could result in an unreachable target if the firmware was corrupted.

5.3.3 Data Structures

There are two data structures of interest to programmers using board_update.lib.

- firmware_info_t - This structure holds board-specific and compile time data about firmware opened with one of the buOpenFirmwareXYZ functions: which includes the currently-executing program, firmware stored in RAM, on the boot flash, in a FAT file, on the serial data flash, or firmware stored in one of the temporary locations.
- bu_download_t - This structure holds state and status information on the currently downloading file when using the buDownloadInit/Tick API.

firmware_info_t

A structure of this type is embedded in the first 1024 bytes of each program. Call `fiProgramInfo()` to get a copy of the structure from the currently running firmware. Call the function `buGetInfo()` to get a copy of the structure from the open firmware image.

```
typedef struct{
   unsigned long magic;                  //  set to _FIRMINFO_MAGIC_NUMBER
   char struct_ver;                      //  version of this structure
   word board_type;                      //  set to _BOARD_TYPE_ at compile time
   unsigned long length                  //  bytes of uncompressed firmware, w/CRC-32
   word version;                         //  user-settable version, (0x0C21 = 12.21)
   word compiler_ver;                    //  set to CC_VER at compile time
   word flags;                           //  bitmask of settings related to build
      _FIRMINFO_FLAG_SEP_INST_DATA       //  Separate I&D enabled
      _FIRMINFO_FLAG_RST28               //  RST28 compiled in
      _FIRMINFO_FLAG_RAM_COMPILE         //  Compile-to-RAM mode
      _FIRMINFO_FLAG_CAN_BE_SECONDARY    //  Can run as secondary firmware
      _FIRMINFO_FLAG_WRAPPED_BIN         //  Secondary firmware installed by RFU
   unsigned long build_timestamp;        //  build-time as seconds since Jan. 1, 1980
   unsigned long mb_type;                //  set to _DC_MB_TYPE_ at compile time
   char reserved[11];                    //  space for future use, set to 0x00
   char user_defined[8];                 //  space reserved for user-defined variables
   char program_name[20];                //  null-terminated, user-defined name
   unsigned long header_crc32;           //  CRC-32 of this structure
} firmware_info_t;
```

The element “program_name” is set by the project macro `_FIRMWARE_NAME_`.

The element “version” is set by the project macro `_FIRMWARE_VERSION_`.

The element “build_timestamp” is set to the time and date in Dynamic C when the program was compiled or it can be overridden by the project macro `_FIRMWARE_TIMESTAMP_`.

`_FIRMINFO_FLAG_CAN_BE_SECONDARY` is set for an image that can be loaded from a non-zero offset on the serial boot flash. Set if `BU_ENABLE_SECONDARY` is defined in the project options.

`_FIRMINFO_FLAG_WRAPPED_BIN` is set for a special firmware image created by the Rabbit Field Utility (RFU) that allows for power-fail safe updates on serial boot flash. This image has a boot loader, firmware markers and embedded firmware inside.

The following information and validation functions make use of firmware_info_t:

- `buGetInfo`
- `fiDump`
- `fiProgramInfo`
- `fiProgramSize`
- `fiValidate`

bu_download_t

This structure is used to store state information for `buDownloadTick()`. The structure is initialized by `buDownloadInit()`. Private elements of this structure may change in future releases, but the caller of buDownloadTick() can make use of the public elements, filesize and bytesread.

```
typedef struct {
   ...
   unsigned long filesize;      // size of downloading file, 0 if size unknown
   unsigned long bytesread;     // bytes of the file read so far
   ...
} bu_download_t;
```

Useful elements the caller of `buDownloadTick()` can use are "filesize" and "bytesread."

5.3.4 Other Useful Configuration Macros

All of the remote update sample programs contain configuration macros that are not specific to Remote Program Update. If you examine the code, you will notice these macros:

```
#define STDIO_DEBUG_SERIAL SADR
#define STDIO_DEBUG_BAUD 115200
#define STDIO_DEBUG_ADDCR
```

They allow you to run a terminal emulation program and use the "DIAG" connector of the programming cable to display output from the Rabbit. This is useful while in the development/debug cycle since the firmware update and reboot process breaks the connection with the Dynamic C debugger. By redirecting STDIO to serial port A, you can keep communication open between your host PC and the Rabbit while the firmware upload/update process runs without Dynamic C.

If a communication method is used to transfer the firmware to a temporary storage location, the configuration macros needed depend on the method used. The most involved method demonstrated by the sample programs is the HTTP server, which includes HTTP authentication and a form-based web page that offers file upload.

5.4 For More Information...

For the FAT filesystem, more information is available in the *Dynamic C User's Manual*. Most of the sample programs listed in Section 4.0 contain code that demonstrate using the FAT filesystem, as well as code comments to explain what is happening.

Additional documentation on the following configuration macros can be found in the *Dynamic C TCP/IP User's Manual, Vols. 1 and 2*.

- `USE_RABBITWEB`
- `TCPCONFIG`
- `USE_HTTP_UPLOAD`
- `MAX_UDP_SOCKET_BUFFERS`
- `SSPEC_FLASHRULES`
- `USE_HTTP_DIGEST_AUTHENTICATION`

6.0 Summary

The remote update feature is a powerful addition to the Dynamic C suite of libraries. Its versatility allows you to select from different storage locations and communication methods. Its interface allows you to quickly and easily give your application remote updating capabilities, letting you make bug fixes or add new features and get them deployed much more efficiently than would otherwise be possible.

Appendix A: API Function Descriptions

This section documents the application programming interface implemented by the Remote Program Update Library, `board_update.lib`, as well as some helper functions in `firmware_info.lib`, a library that is automatically included by the remote update library.

`buCloseFirmware`
`buCopyToSecondary`
`buDownloadInit`
`buDownloadTick`
`buGetInfo`
`buInstallFirmware`
`buMarkerBuild`
`buMarkerChecksum`
`buMarkerDump`
`buMarkerRead`
`buMarkerReadBoot`
`buMarkerVerify`
`buOpenFirmwareBoot`
`buOpenFirmwareFAT`
`buOpenFirmwareRAM`
`buOpenFirmwareRunning`
`buOpenFirmwareSFlash`
`buOpenFirmwareSecondary`
`buOpenFirmwareTemp`
`buReadFirmware`
`buRestoreFirmware`
`buRewindFirmware`
`buTempClose`
`buTempCreate`
`buTempWrite`
`buVerifyFirmware`
`buVerifyFirmwareBlocking`
`fiDump`
`fiProgramInfo`
`fiProgramSize`
`fiValidate`

buCloseFirmware

```
int buCloseFirmware();
```

DESCRIPTION

Close the firmware source stream, previously opened with a buOpenFirmwareXYZ call.

If temporary memory was allocated to cache a copy of the firmware during the verification process, `buCloseFirmware()` will also release that memory.

RETURN VALUE

0: Closed firmware source stream.

`-EPERM`: Source already closed.

`-EBUSY`: Timeout waiting for FAT filesystem. Continue to call `buCloseFirmware()` until it returns something other than `-EBUSY`.

LIBRARY

`board_update.lib`

SEE ALSO

`buOpenFirmwareRunning`, `buOpenFirmwareRAM`, `buOpenFirmwareBoot`, `buOpenFirmwareFAT`, `buOpenFirmwareSFlash`, `buOpenFirmwareTemp`, `buReadFirmware`, `buVerifyFirmware`, `buVerifyFirmwareBlocking`, `buRewindFirmware`, `buInstallFirmware`, `buRestoreFirmware`

buCopyToSecondary

```
int buCopyToSecondary( int far *progress);
```

DESCRIPTION

Simple wrapper for buVerifyToSecondary(progress, 1). See that function's help for additional details.

LIBRARY

board_update.lib

SEE ALSO

buVerifyFirmware, buVerifyToSecondary

buDownloadInit

```
int buDownloadInit (bu_download_t *bu_dl, tcp_Socket *sock,
   const char *url);
```

DESCRIPTION

Initiate FTP or HTTP connection and initialize status structure to pass to `buDownloadTick()`, in order to download a file from a server and save it to the temporary location used by `buOpenFirmwareTemp()`.

PARAMETERS

`bu_dl` Pointer to status structure.

`sock` Pointer to TCP socket to use for making HTTP connections. For FTP connections, the `ftp_client.lib` library uses its own sockets and this parameter is ignored (and can be set to NULL).

`url` URL of file to download, in one of the following formats (items in [] are optional):

- `http://[user:pass@]hostname[:port]/filename`
- `ftp://[user:pass@]hostname[:port]/filename`
- `www.hostname[:port]/filename (assumes http://)`
- `ftp.hostname[:port]/filename (assumes ftp://)`

HTTP defaults to port 80 and no credentials (username/password).
FTP defaults to port 21 and anonymous FTP.

RETURN VALUE

0: Success, connection established. Can pass <bu_dl> to `buDownloadTick()` to continue download.

`-EINVAL`: Error parsing <url> or <localfile>.

`-EBUSY`: Timeout opening connection, call `buDownloadTick()` to continue.

`-NETERR_DNSERROR`: Unable to resolve hostname from <url>.

`-NETERR_INACTIVE_TIMEOUT`: Timed out due to inactivity

`-NETERR_HOST_REFUSED`: Unable to connect to FTP server.

LIBRARY

`board_update.lib`

SEE ALSO

`buDownloadTick`, `buOpenFirmwareTemp`

buDownloadTick

```
int buDownloadTick( bu_download_t *bu_dl );
```

DESCRIPTION

Read more data from HTTP or FTP server, and write it out to the temporary location (see buTempCreate for details).

PARAMETERS

bu_dl Pointer to status structure set up by buDownloadInit().

RETURN VALUE

0: Success, file download complete.
-ENODATA: Server sent a 0-byte file.
-EBUSY: Download in progress.
-EBUSY: Download in progress.
-EINVAL: Invalid structure passed as parameter 1.
Any other negative value: I/O error when updating the directory entry.

Errors for HTTP connections:
-ENOTCONN: Connection closed, cannot read from socket.

Errors for FTP connections:
FTPC_ERROR: General error, call ftp_last_code() for details.
FTPC_NOHOST: Could not connect to server.
FTPC_NOBUF: No buffer or data handler.
FTPC_TIMEOUT: Timed out on close: data may or may not be OK.
FTPC_DHERROR: Data handler error in FTPDH_END operation.
FTPC_CANCELLED: FTP control socket was aborted (reset) by the server.

NOTE: Monitor download progress via bu_dl->filesize and bu_dl->bytesread (both unsigned long).

LIBRARY

board_update.lib

SEE ALSO

buDownloadInit, buOpenFirmwareTemp

buGetInfo

```
int buGetInfo( far firmware_info_t *fi );
```

DESCRIPTION

Get a copy of the firmware information from the last firmware image opened with one of the following functions:

- `buOpenFirmwareRunning()`: Image of the currently-executing program.
- `buOpenFirmwareRAM()`: Firmware stored at an arbitrary location in RAM.
- `buOpenFirmwareBoot()`: Firmware stored on the boot flash (serial or parallel).
- `buOpenFirmwareFAT()`: Firmware stored in a FAT file.
- `buOpenFirmwareSFlash()`: Firmware stored on serial flash (read with `sflash.lib`).
- `buOpenFirmwareTemp()`: Firmware stored in temporary location.

PARAMETERS

fi	Pointer to buffer to receive copy of firmware information retrieved from last opened firmware image.

RETURN VALUE

Error codes shared with fiValidate:

0: Information is valid.
`-EINVAL`: <fi> is NULL
`-EILSEQ`: Not a valid `firmware_info_t` structure (bad marker bytes or unsupported version of structure).
`-EBADMSG`: Bad CRC (structure has been corrupted).

Additional error codes:
`-EPERM`: Source not open, need to call buOpenFirmwareXYZ first.
`-ENODATA`: Firmware info not found in source.
`-EBUSY`: Still reading first 1KB from source, call again.

Firmware opened with `buOpenFirmwareFAT()`:
`-EEOF`: File length smaller than firmware length read from header (only for uncompressed files).

LIBRARY

`board_update.lib`

SEE ALSO

`firmware_info_t`, `fiValidate`, `fiDump`, `fiProgramInfo`, `buOpenFirmwareRunning`, `buOpenFirmwareRAM`, `buOpenFirmwareBoot`, `buOpenFirmwareFAT`, `buOpenFirmwareSFlash`, `buOpenFirmwareTemp`

buInstallFirmware

```
int buInstallFirmware();
```

DESCRIPTION

Copy the previously opened and (possibly) verified firmware to the boot flash. If there is a problem copying the firmware, this function calls buRestoreFirmware() to copy a copy of the running firmware back to the boot flash.

If the firmware has not already been verified with `buVerifyFirmware()` (or `buVerifyFirmwareBlocking()`), this function will verify the firmware before installing it.

To skip the verification process (something that could result in installing non-bootable firmware on the device), use the `BU_FLAG_NOVERIFY` option when opening the firmware source.

If the installation was successful, the caller will probably want to reboot using the `forceWatchdogTimeout()` function.

`buInstallFirmware()` will always close the firmware image (if one was open) before returning.

RETURN VALUE

0: Firmware installed (running in debugger, so reboot skipped).
`-ENODATA`: Source not open, or firmware info not found in source.
`-EPERM`: Attempting to install boot firmware on top of itself, or firmware too large for boot flash, or non-relocatable firmware to a non-zero flash address, or some other failure
`-EIO`: Can't rewind source.
`-EBADMSG`: CRC-32 mismatch after installing.
`-ENOMEM`: Couldn't allocate buffer to copy firmware.

LIBRARY

`board_update.lib`

SEE ALSO

`buOpenFirmwareRunning`, `buOpenFirmwareRAM`, `buOpenFirmwareBoot`, `buOpenFirmwareFAT`, `buOpenFirmwareSFlash`, `buOpenFirmwareTemp`, `buReadFirmware`, `buVerifyFirmware`, `buVerifyFirmwareBlocking`, `buRewindFirmware`, `buCloseFirmware`, `buRestoreFirmware`

buMarkerBuild

```
int buMarkerBuild( firmware_marker_t far *marker,
   word sequence, long flash_offset);
```

DESCRIPTION

Build a firmware_marker structure for the firmware image stored at a given flash offset.

PARAMETERS

marker	Buffer for storing the marker.
sequence	Sequence number to embed in marker (usually the previous boot image's sequence number, plus 1). Note that the loader and board_update.lib handle sequence number rollover correctly (i.e., 0x0000 is considered newer than 0xFFFF).
flash_offset	Flash offset of firmware to build a marker for.

RETURN VALUE

-EINVAL: <marker> is NULL or offset is negative

buMarkerChecksum

```
word buMarkerChecksum( const firmware_marker_t far *marker);
```

DESCRIPTION

Calculate the proper checksum for a given firmware_marker structure.

PARAMETERS

marker Marker to calculate checksum for.

RETURN VALUE

Checksum for given marker. If marker is NULL, checksum is 0.

buMarkerDump

```
void buMarkerDump( const firmware_marker_t far *marker);
```

DESCRIPTION

Debugging tool that dumps the contents of a firmware marker to STDOUT.

PARAMETERS

marker Marker to display on STDOUT.

RETURN VALUE

None.

LIBRARY

board_update.lib

buMarkerRead

```
int buMarkerRead( firmware_marker_t far *dest, word flags);
```

DESCRIPTION

Read the firmware marker for a given firmware image.

PARAMETERS

dest	Buffer for storing the marker.
flags	Either BU_FLAG_IMAGE_Z, BU_FLAG_IMAGE_A or BU_FLAG_IMAGE_B.

RETURN VALUE

0: Read marker
-EINVAL: <dest> is NULL or <flags> is not a valid setting
-EBUSY: timeout trying to read serial boot flash

LIBRARY

board_update.lib

SEE ALSO

buMarkerVerify

NOTE: If BU_FLAG_IMAGE_Z is passed as parameter 2, the firmware_marker_t structure is manually assembled by reading image Z's firmware_info_t structure from flash and fleshing out the rest of the structure.

buMarkerReadBoot

```
buMarkerReadBoot( firmware_marker_t far *dest);
```

DESCRIPTION

Read the firmware_marker_t structure for the firmware image that will boot on next reset. This isn't necessarily the same as the firmware that is currently running.

PARAMETERS

dest Buffer for storing the marker or NULL if the caller just needs to know which image will boot.

RETURN VALUE

BU_FLAG_IMAGE_Z, BU_FLAG_IMAGE_A, BU_FLAG_IMAGE_B: marker saved to <dest> is for image Z, A or B.

-EBUSY: timeout trying to read serial boot flash

NOTE: If this function returns BU_FLAG_IMAGE_Z, the firmware_marker_t structure is manually assembled by reading image Z's firmware_info_t structure from flash and fleshing out the rest of the structure.

LIBRARY

board_update.lib

buMarkerVerify

```
int buMarkerVerify( const firmware_marker_t far *marker);
```

DESCRIPTION

Verify the marker read from serial flash. Checks the version field, checksum and embedded firmware_info_t structure (which has its own CRC-32).

PARAMETERS

marker Marker to verify.

RETURN VALUE

0: Information is valid.

-EINVAL: <marker> is NULL.

-EILSEQ: Not a valid marker.

-EBADMSG: Bad firmware_info_t CRC-32 or firmware_marker_t checksum (structure has been corrupted).

LIBRARY

board_update.lib

SEE ALSO

buMarkerRead

buOpenFirmwareBoot

```
int buOpenFirmwareBoot( word firmflags);
```

DESCRIPTION

Access the boot firmware image. This is the image that the board will boot from on reset, and isn't necessarily the image used when the currently running program booted.

PARAMETERS

firmflags Bitmask combination of the following flags:

- BU_FLAG_NONE: Not compressed or encrypted.
- BU_FLAG_NOVERIFY: Skip pre-install verify (dangerous).

Note that since the boot firmware can't possibly be encrypted or compressed, the compression and encryption flags are not valid.

After opening a firmware image, use the buGetInfo function to examine the info structure in its header; buVerifyFirmware to validate its CRC-32.

RETURN VALUE

0: Successfully opened firmware image.
-EINVAL: Compression or encryption flag passed in <firmflags>.

LIBRARY

board_update.lib

SEE ALSO

buOpenFirmwareRunning, buOpenFirmwareRAM,buOpenFirmwareFAT, buOpenFirmwareSFlash, buOpenFirmwareTemp, buGetInfo, buVerifyFirmware, buVerifyFirmwareBlocking, buInstallFirmware

buOpenFirmwareFAT

```
int buOpenFirmwareFAT( const char *filepath, word firmflags );
```

DESCRIPTION

Access a firmware image stored on the FAT filesystem. To use this function, the statement #use "FAT.LIB" must come before the statement #use "board_update.lib" in your program.

After opening a firmware image, call `buGetInfo()` to examine the information structure in its header (`firmware_info_t`); call `buVerifyFirmware()` to validate its CRC-32 and then `buInstallFirmware()` to install it to the boot flash.

PARAMETERS

`filepath` Full filepath, in one of the following formats:

- `a:/path/firmware.bin`
- `a:firmware.bin`
- `/a/path/firmware.bin`
- `firmware.bin (defaults to partition A)`

`firmflags` Bitmask combination of the following flags:

- `BU_FLAG_NONE`: Not compressed or encrypted.
- `BU_FLAG_NOVERIFY`: Skip pre-install verify (dangerous).

Compression Options (currently unsupported):

- `BU_FLAG_LZ77`: zcompress format
- `BU_FLAG_DEFLATE`: zlib/deflate format (RFC1950/RFC1951)
- `BU_FLAG_GZIP`: gzip format (RFC1952)

Encryption Options (currently unsupported):

- `BU_FLAG_3DES`: 3DES (Triple-DES)
- `BU_FLAG_AES`: AES (Advanced Encryption Standard)

RETURN VALUE

0: Successfully opened firmware image.
`-EINVAL`: Could not parse <filepath>.
`-ENOENT`: File <filepath> does not exist.
`-EMFILE`: Too many open files.

LIBRARY

`board_update.lib`

SEE ALSO

`buOpenFirmwareRunning`, `buOpenFirmwareRAM`, `buOpenFirmwareBoot`, `buOpenFirmwareSFlash`, `buOpenFirmwareTemp`, `buGetInfo`, `buVerifyFirmware`, `buVerifyFirmwareBlocking`, `buInstallFirmware`

buOpenFirmwareRAM

```
int buOpenFirmwareRAM( const byte far *address, unsigned long length,
   word firmflags );
```

DESCRIPTION

Access a firmware image stored at a given memory location (either in RAM or memory-mapped parallel flash).

After opening a firmware image, call `buGetInfo()` to examine the info structure in its header; call `buVerifyFirmware()` to validate its CRC-32; then `buInstallFirmware()` to install it to the boot flash.

PARAMETERS

`address` Start address of image.

`length` Total bytes in image. Set to zero if length is unknown.

`firmflags` Bitmask combination of the following flags:

- `BU_FLAG_NONE`: Not compressed or encrypted.
- `BU_FLAG_NOVERIFY`: Skip pre-install verify (dangerous).

Compression Options (currently unsupported):

- `BU_FLAG_LZ77`: zcompress format
- `BU_FLAG_DEFLATE`: zlib/deflate format (RFC1950/RFC1951)
- `BU_FLAG_GZIP`: gzip format (RFC1952)

Encryption Options (currently unsupported):

- `BU_FLAG_3DES`: 3DES (Triple-DES)
- `BU_FLAG_AES`: AES (Advanced Encryption Standard)

RETURN VALUE

0: Successfully opened firmware image.
`-EINVAL`: Invalid parameters passed to function.

LIBRARY

`board_update.lib`

SEE ALSO

`buOpenFirmwareRunning`, `buOpenFirmwareFAT`, `buOpenFirmwareBoot`, `buOpenFirmwareSFlash`, `buOpenFirmwareTemp`, `buGetInfo`, `buVerifyFirmware`, `buVerifyFirmwareBlocking`, `buInstallFirmware`

buOpenFirmwareRunning

```
int buOpenFirmwareRunning( word firmflags );
```

DESCRIPTION

Access the currently-running firmware image in RAM. On boards without fast SRAM for program execution, this is the same as calling buOpenFirmwareBoot (and will read the firmware from the parallel boot flash).

PARAMETERS

firmflags Bitmask combination of the following flags:

- BU_FLAG_NONE: Not compressed or encrypted.
- BU_FLAG_NOVERIFY: Skip pre-install verify (dangerous).

Note that since the boot firmware cannot be encrypted or compressed, the compression and encryption flags are not valid.

After opening a firmware image, call buGetInfo() to examine the information structure in its header; call buVerifyFirmware() to validate its CRC-32 and then call buInstallFirmware() to install it to the boot flash.

RETURN VALUE

0: Successfully opened firmware image.
-EINVAL: Compression or encryption flag passed in <firmflags>

LIBRARY

board_update.lib

SEE ALSO

buOpenFirmwareRAM, buOpenFirmwareFAT, buOpenFirmwareBoot, buOpenFirmwareSFlash, buOpenFirmwareTemp, buGetInfo, buVerifyFirmware, buVerifyFirmwareBlocking, buInstallFirmware

buOpenFirmwareSFlash

```
int buOpenFirmwareSFlash( const sf_device *dev, int bank, long page,
   unsigned long bytesinfile, word firmflags );
```

DESCRIPTION

Access a firmware image stored on the serial flash. If you are going to use this function, you need to have the statement `#use "SFLASH.LIB"` in your program before the statement `#use "board_update.lib"`.

After opening a firmware image, call `buGetInfo()` to examine the information structure in its header; call `buVerifyFirmware()` to validate its CRC-32, then call `buInstallFirmware()` to install it to the boot flash.

PARAMETERS

dev — Pointer to `sf_device` structure for the flash chip, populated by `sf_initDevice()`.

bank — RAM bank to use when reading the data (set to 1 or 2).

page — Serial flash page with first byte of firmware image.

bytesinfile — Number of bytes used on serial flash for firmware image. Set to zero if length is unknown.

firmflags — Bitmask combination of the following flags:

- `BU_FLAG_NONE`: Not compressed or encrypted.
- `BU_FLAG_NOVERIFY`: Skip pre-install verify (dangerous).

Compression Options (currently unsupported):

- `BU_FLAG_LZ77`: zcompress format
- `BU_FLAG_DEFLATE`: zlib/deflate format (RFC1950/RFC1951)
- `BU_FLAG_GZIP`: gzip format (RFC1952)

Encryption Options (currently unsupported):

- `BU_FLAG_3DES`: 3DES (Triple-DES)
- `BU_FLAG_AES`: AES (Advanced Encryption Standard)

RETURN VALUE

0: Successfully opened firmware image.

`-EINVAL`: Invalid parameter passed in.

LIBRARY

`board_update.lib`

SEE ALSO

`buOpenFirmwareRunning`, `buOpenFirmwareRAM`, `buOpenFirmwareFAT`, `buOpenFirmwareBoot`, `buOpenFirmwareTemp`, `sf_initDevice`, `buGetInfo`, `buVerifyFirmware`, `buVerifyFirmwareBlocking`, `buInstallFirmware`

buOpenFirmwareSecondary

```
int buOpenFirmwareSecondary( word firmflags);
```

DESCRIPTION

Access the secondary firmware image stored on the boot flash as the A-image or B-image.

PARAMETERS

firmflags Bitmask combination of the following flags:

- BU_FLAG_NONE: Not compressed or encrypted.
- BU_FLAG_NOVERIFY: Skip pre-install verify (dangerous).

And one of the following flags:

- BU_FLAG_IMAGE_A: Open the A-image.
- BU_FLAG_IMAGE_B: Open the B-image.

NOTE: Since bootable firmware can't possibly be encrypted or compressed, the compression and encryption flags are not valid.

After opening a firmware image, use the buGetInfo function to examine the info structure in its header; buVerifyFirmware to validate its CRC-32.

RETURN VALUE

0: Successfully opened firmware image.

-EBUSY: Timeout trying to read marker.

-ENOENT: Flash does not have a valid marker for the requested firmware image (A-image or B-image).

-EINVAL: Invalid flag set in **firmflags**.

LIBRARY

board_update.lib

SEE ALSO

buOpenFirmwareRunning, buOpenFirmwareRAM, buOpenFirmwareBoot, buOpenFirmwareSFlash, buOpenFirmwareTemp, buGetInfo, buVerifyFirmware, buVerifyFirmwareBlocking, buInstallFirmware

buOpenFirmwareTemp

```
int buOpenFirmwareTemp( word firmflags );
```

DESCRIPTION

Read from a firmware image in temporary storage. Use the buTempCreate/Write/Close API to write to the temporary firmware image.

View the function help for `buTempCreate()` for further information on temporary firmware images (such as storage location).

After opening a firmware image, call `buGetInfo()` to examine the information structure in its header; call `buVerifyFirmware()` to validate its CRC-32 and then call `buInstallFirmware()` to install it to the boot flash.

PARAMETERS

`firmflags` Bitmask combination of the following flags:

- `BU_FLAG_NONE`: Not compressed or encrypted.
- `BU_FLAG_NOVERIFY`: Skip pre-install verify (dangerous).

Compression Options (currently unsupported):

- `BU_FLAG_LZ77`: zcompress format
- `BU_FLAG_DEFLATE`: zlib/deflate format (RFC1950/RFC1951)
- `BU_FLAG_GZIP`: gzip format (RFC1952)

Encryption Options (currently unsupported):

- `BU_FLAG_3DES`: 3DES (Triple-DES)
- `BU_FLAG_AES`: AES (Advanced Encryption Standard)

RETURN VALUE

0: Successfully opened firmware image.

`-ENODATA`: Firmware information not found in source.

`-EBUSY`: Timeout trying to open temp firmware.

Error codes when using a FAT file for temporary storage:

`-EINVAL`: Couldn't parse `BU_TEMP_FILE`.

`-ENOENT`: File `BU_TEMP_FILE` does not exist.

`-EMFILE`: Too many open files.

Error codes when using the serial flash for temporary storage:

`-ENODEV`: Cannot find/read the serial flash.

Error codes when storing direct to boot firmware (RCM5600W) or a secondary boot image:

`-EINVAL`: Compressed and encrypted options not supported.

LIBRARY

`board_update.lib`

SEE ALSO

`buOpenFirmwareRunning`, `buOpenFirmwareRAM`, `buOpenFirmwareFAT`, `buOpenFirmwareBoot`, `buOpenFirmwareSFlash`, `buGetInfo`, `buVerifyFirmware`, `buVerifyFirmwareBlocking`, `buInstallFirmware`,

buTempCreate, buTempWrite, buTempClose

buReadFirmware

```
int buReadFirmware( byte far *dest, int bytesrequested );
```

DESCRIPTION

Read the next <bytesrequested> of the unencrypted, uncompressed firmware into the buffer <dest>. The firmware must be opened first using one of the buOpenFirmwareXYZ functions listed in SEE ALSO below.

PARAMETERS

dest Pointer to destination buffer. If NULL and this is our first read, loads a buffer with first 1024 bytes of the image and populates _bu_firmfile.info.

If <dest> is NULL for all reads, this function is being called by buVerifyFirmware() and it just needs to calculate the CRC-32 of the decrypted, uncompressed firmware.

bytesrequested Decrypted, uncompressed firmware bytes to read.

RETURN VALUE

0 to <bytesrequested>: Number of bytes read.
-EPERM: Source not open, need to call buOpenFirmwareXYZ first.
-ENODATA: Firmware info not found in source.
-EINVAL: Must specify a non-NULL <dest> if <bytesrequested> > 0.
-EEOF: On the first read, stream is not large enough to contain entire firmware image. On subsequent reads, we have already read the entire firmware image

LIBRARY

board_update.lib

SEE ALSO

buOpenFirmwareRunning, buOpenFirmwareRAM, buOpenFirmwareFAT, buOpenFirmwareBoot, buOpenFirmwareSFlash, buOpenFirmwareTemp, buVerifyFirmware, buVerifyFirmwareBlocking, buRewindFirmware, buInstallFirmware, buCloseFirmware, buRestoreFirmware

buRestoreFirmware

```
int buRestoreFirmware( long bytestoerase );
```

DESCRIPTION

Copy the running firmware image back to the boot flash. This is typically only done when a firmware update fails for some reason, and it's necessary to get back to a bootable state.

PARAMETERS

bytestoerase Bytes of boot flash device to erase before copying. Pass 0 to use the length of the running firmware image.

RETURN VALUE

0: Successfully copied running firmware back to boot flash.

`-EIO`: I/O error trying to read running firmware.

`-ENOMEM`: Unable to allocate buffer for copying data.

LIBRARY

`board_update.lib`

SEE ALSO

`buOpenFirmwareRunning`, `buOpenFirmwareRAM`, `buOpenFirmwareFAT`, `buOpenFirmwareBoot`, `buOpenFirmwareSFlash`, `buOpenFirmwareTemp`, `buReadFirmware`, `buVerifyFirmware`, `buVerifyFirmwareBlocking`, `buRewindFirmware`, `buCloseFirmware`, `buRestoreFirmware`

buRewindFirmware

```
int buRewindFirmware();
```

DESCRIPTION

Rewind the firmware source back to the beginning. This is necessary when using `buVerifyFirmware()` before calling `buInstallFirmware()`.

RETURN VALUE

0: Firmware source already rewound, or successfully rewound.

`-EPERM`: Source not open, need to call buOpenFirmwareXYZ first.

`-EBUSY`: Timeout waiting for FAT filesystem.

`-EIO`: Can't rewind source.

LIBRARY

`board_update.lib`

SEE ALSO

`buOpenFirmwareRunning`, `buOpenFirmwareRAM`, `buOpenFirmwareBoot`, `buOpenFirmwareBoot`, `buOpenFirmwareSFlash`, `buOpenFirmwareTemp`, `buReadFirmware`, `buVerifyFirmware`, `buVerifyFirmwareBlocking`, `buInstallFirmware`, `buCloseFirmware`, `buRestoreFirmware`

buTempClose

```
int buTempClose();
```

DESCRIPTION

Close temporary firmware image.

View the function help for `buTempCreate()` for details on where the temporary firmware image is stored on various hardware types.

RETURN VALUE

0: Successfully opened temp firmware image for writing.

`-EPERM`:Temporary firmware image is not open.

`-EBUSY`: Operation took longer than `BU_FAT_TIMEOUT` milliseconds. To complete the operation, call `buTempClose()` again.

`-EIO`: Error trying to truncate or close FAT file. Temporary file deleted.

LIBRARY

`board_update.lib`

SEE ALSO

`buTempCreate`, `buTempWrite`

buTempCreate

```
int buTempCreate();
```

DESCRIPTION

Prepare to write to the temporary firmware image.

On boards with a serial boot flash, the temporary image can be stored on the flash between the boot image and the userblock.

On boards with a serial data flash, the temporary image can be stored directly on the flash pages, or on a FAT filesystem hosted on the flash.

On the RCM4400W, the temporary image can be stored on a portion of the 1MB serial data flash shared with the FPGA firmware for the Wi-Fi interface.

To set the storage location, use one of the following macros:

```
// use FAT filesystem for temporary firmware image and override defilename
#define BU_TEMP_USE_FAT

// override filename used for temporary image (default = "a:firmware.bin")
#define BU_TEMP_FILE "a:firmware.bin"

// use serial boot flash for temporary firmware image
#define BU_TEMP_USE_SBF

// write temporary firmware image directly to serial flash
#define BU_TEMP_USE_SFLASH

// override default starting page number on serial flash (default = 0)
#define BU_TEMP_PAGE_OFFSET 0

// write image directly to boot flash (dangerous, serial only)
#define BU_TEMP_USE_DIRECT_WRITE

// store two copies of firmware on serial boot flash, for powerfail-safe firmware updates
#define BU_TEMP_USE_SECONDARY
```

RETURN VALUE

0: Successfully opened temp firmware image for writing.
`-EPERM`: Not supported on this hardware.
`-ENODEV`: Couldn't read from serial flash.
`-EBUSY`: Timeout waiting for FAT filesystem.
<0: Error opening FAT file, see `fat_Open()`[1] for full list of error codes and their meanings.

LIBRARY

`board_update.lib`

SEE ALSO

`buTempWrite`, `buTempClose`, `fat_Open`

1. The function description for fat_Open() is found in the Library Lookup feature of the Dynamic C Help menu and the *Dynamic C Function Reference Manual*.

buTempWrite

```
int buTempWrite( const char far *buffer, int writebytes );
```

DESCRIPTION

Write data to temporary firmware image that was previously opened with `buTempCreate()`.

View the function help for `buTempCreate()` for details on where the temporary firmware image is stored on various hardware types.

PARAMETERS

`buffer` Pointer to source buffer for write

`writebytes` Number of bytes to write.

RETURN VALUE

0 to <writebytes>: Number of bytes written. Less than <writebytes> may be written.
`-EINVAL`: <buffer> is NULL or <writebytes> is less than 0.
`-EPERM`: Temporary firmware image is not open.
`-ENOSPC`: Out of space for image.
`-EFBIG`: Image is larger than maximum size for this hardware.
`-EIO`: Error trying to write to image.
`-EBUSY`: Timeout waiting for FAT or serial boot flash driver. Call `buTempWrite()` again with same parameters.
<0: Some other error trying to write to temporary firmware image.

If `buTempWrite()` returns a value less than zero, it will automatically close the temporary firmware image.

LIBRARY

`board_update.lib`

SEE ALSO

`buTempCreate`, `buTempClose`

buVerifyFirmware

```
int buVerifyFirmware( int far *progress );
```

DESCRIPTION

Verify that the currently selected firmware image is OK to install on this device. Verifies CRC-32 of firmware image (to detect corruption) and confirms that the firmware was compiled for this target hardware.

Because validating firmware can take a significant amount of time, especially with encrypted or compressed firmware, this is a non-blocking function.

Use `buVerifyFirmwareBlocking()` to block until verification is complete or an error is detected.

PARAMETERS

`progress` Address of an integer to store a progress indicator. On a return of the value `-EAGAIN`, `*progress` is set to a value from 0 to 10,000, representing the percent complete in .01% increments (1234 = 12.34%). On a return of 0, `*progress` is set to 10,000.

RETURN VALUE

0: Verification complete, firmware image is OK to install.
`-EAGAIN`: Verification partially complete, call function again.
`-ENODATA`: Source not open, or firmware info not found in source.
`-EEOF`: Stream is not large enough to contain the entire firmware image.
`-EPERM`: Firmware was compiled for a different target.
`-EBADDATA`: CRC-32 mismatch, firmware image corrupted.

LIBRARY

`board_update.lib`

SEE ALSO

`buOpenFirmwareRunning`, `buOpenFirmwareRAM`, `buOpenFirmwareBoot`, `buOpenFirmwareBoot`, `buOpenFirmwareSFlash`, `buOpenFirmwareTemp`, `buReadFirmware`, `buVerifyFirmwareBlocking`, `buRewindFirmware`, `buInstallFirmware`, `buCloseFirmware`, `buRestoreFirmware`

buVerifyFirmwareBlocking

```
int buVerifyFirmwareBlocking();
```

DESCRIPTION

Verify that the currently selected firmware image is OK to install on this device. This function verifies the CRC-32 of the firmware image (to detect corruption) and confirms that the firmware was compiled for this target hardware.

Because validating firmware can take a significant amount of time, especially with encrypted or compressed firmware, consider using `buVerifyFirmware()`, the non-blocking version of this function.

RETURN VALUE

0: Firmware is ready to install.
`-ENODATA`: Source not open, or firmware info not found in source.
`-EPERM`: Firmware was compiled for a different target.
`-EBADDATA`: CRC-32 mismatch, firmware image corrupted.

LIBRARY

`board_update.lib`

SEE ALSO

`buOpenFirmwareRunning`, `buOpenFirmwareRAM`, `buOpenFirmwareFAT`, `buOpenFirmwareBoot`, `buOpenFirmwareSFlash`, `buOpenFirmwareTemp`, `buReadFirmware`, `buVerifyFirmware`, `buRewindFirmware`, `buInstallFirmware`, `buCloseFirmware`, `buRestoreFirmware`

buVerifyToSecondary

```
int buVerifyToSecondary( int far *progress, int copy);
```

DESCRIPTION

Verify that the currently selected firmware image is OK to install on this device while optionally copying it to the secondary firmware location if available. Verifies CRC-32 of firmware image (to detect corruption) and confirms that the firmware was compiled for this target hardware.

Because validating firmware can take a significant amount of time, especially with encrypted or compressed firmware, this is a non-blocking function.

Use buVerifyFirmwareBlocking to block (without copying) until verification is complete or an error is detected.

PARAMETERS

Parameter 1 Address of an integer to store a progress indicator. On a return of -EAGAIN, *progress is set to a value from 0 to 10,000, representing the percent complete in .01% increments (1234 = 12.34%). On a return of 0, *progress is set to 10,000.

Parameter 2 Set to 0 to verify only, or 1 to copy to secondary location while verifying. Note that this parameter is ignored unless BU_ENABLE_SECONDARY is defined.

RETURN VALUE

0: Verification (and copy) complete, firmware image is ready to be installed.

-EAGAIN: Verification (and copy) partially complete, call function again.
-ENODATA: Source not open, or firmware info not found in source.
-EEOF: Stream isn't large enough to contain entire firmware image.
-EPERM: Firmware was compiled for a different target.
-EBADDATA: CRC-32 mismatch, firmware image corrupted.
-EIO: Error trying to write image to secondary location.

LIBRARY

board_update.lib

SEE ALSO

buOpenFirmwareRunning, buOpenFirmwareRAM, buOpenFirmwareFAT, buOpenFirmwareBoot, buOpenFirmwareSFlash, buOpenFirmwareTemp, buReadFirmware, buVerifyFirmwareBlocking, buRewindFirmware, buInstallFirmware, buCloseFirmware, buRestoreFirmware, buVerifyFirmware, buCopyToSecondary

fiDump

```
int fiDump( const far firmware_info_t *fi );
```

DESCRIPTION

Display information stored in the `firmware_info_t` structure to the Stdio window in human-readable form.

PARAMETERS

fi Pointer to firmware information retrieved with `buGetInfo()` or `fiProgramInfo()`.

RETURN VALUE

0: Information is valid.

`-EINVAL`: <fi> is NULL

`-EILSEQ`: Not a valid `firmware_info_t` structure (bad marker bytes or unsupported version of structure).

`-EBADMSG`: Bad CRC (structure has been corrupted).

LIBRARY

`firmware_info.lib`

SEE ALSO

`firmware_info_t`, `fiValidate`, `buGetInfo`, `fiProgramInfo`

fiProgramInfo

```
int fiProgramInfo( far firmware_info_t *fi );
```

DESCRIPTION

Get a copy of the firmware information from the currently-executing program.

PARAMETERS

fi Pointer to buffer to receive copy of firmware information.

RETURN VALUE

0: Information is valid.
-EINVAL: <fi> is NULL
-EILSEQ: Not a valid firmware_info_t structure (bad marker bytes or unsupported version of structure).
-EBADMSG: Bad CRC (structure has been corrupted).

LIBRARY

firmware_info.lib

SEE ALSO

firmware_info_t, fiProgramInfo, fiDump, buGetInfo

fiProgramSize

```
unsigned long fiProgramSize();
```

DESCRIPTION

Get the size of the currently executing program.

RETURN VALUE

Number of bytes in the firmware .BIN of the currently executing program.

LIBRARY

firmware_info.lib

SEE ALSO

firmware_info_t, fiValidate, fiDump, buGetInfo

fiValidate

```
int fiValidate( const far firmware_info_t *fi );
```

DESCRIPTION

Validate information stored in `firmware_info_t` structure.

PARAMETERS

fi	Pointer to firmware information retrieved with `buGetInfo()` or `fiProgramInfo()`.

RETURN VALUE

0: Information is valid.

`-EINVAL`: <fi> is NULL

`-EILSEQ`: Not a valid `firmware_info_t` structure (bad marker bytes or unsupported version of structure).

`-EBADMSG`: Bad CRC (structure has been corrupted).

LIBRARY

`firmware_info.lib`

SEE ALSO

`firmware_info_t`, `fiDump`, `buGetInfo`, `fiProgramInfo`