



RomPager[®] Intro Web Server
for ThreadX[®]
Programming Reference

ALLEGRO SOFTWARE DEVELOPMENT CORPORATION
1740 Massachusetts Avenue • Boxborough, MA 01719
Telephone: 978 264-6600 • Fax: 978 266 2839 • www.allegrosoft.com

This manual is © Copyright 1996-2010 by Allegro Software Development Corporation. All rights reserved. RomPager®, RomPlug®, RomXML® and RomXOAP® are trademarks registered in the U.S. Patent and Trademark Office to Allegro Software Development Corporation. RomMailer, RomPOP, RomWebClient, RomDNS, RomTime, RomCLI, RomSShell, and RomRadius are trademarks of Allegro Software Development Corporation. The RomPager and RomMailer products described in this manual are protected by U.S. Patent No. 6,782,427. UPnP™ is a certification mark of the UPnP Implementers Corporation. The names of other companies and products mentioned herein are the trademarks of their respective owners.

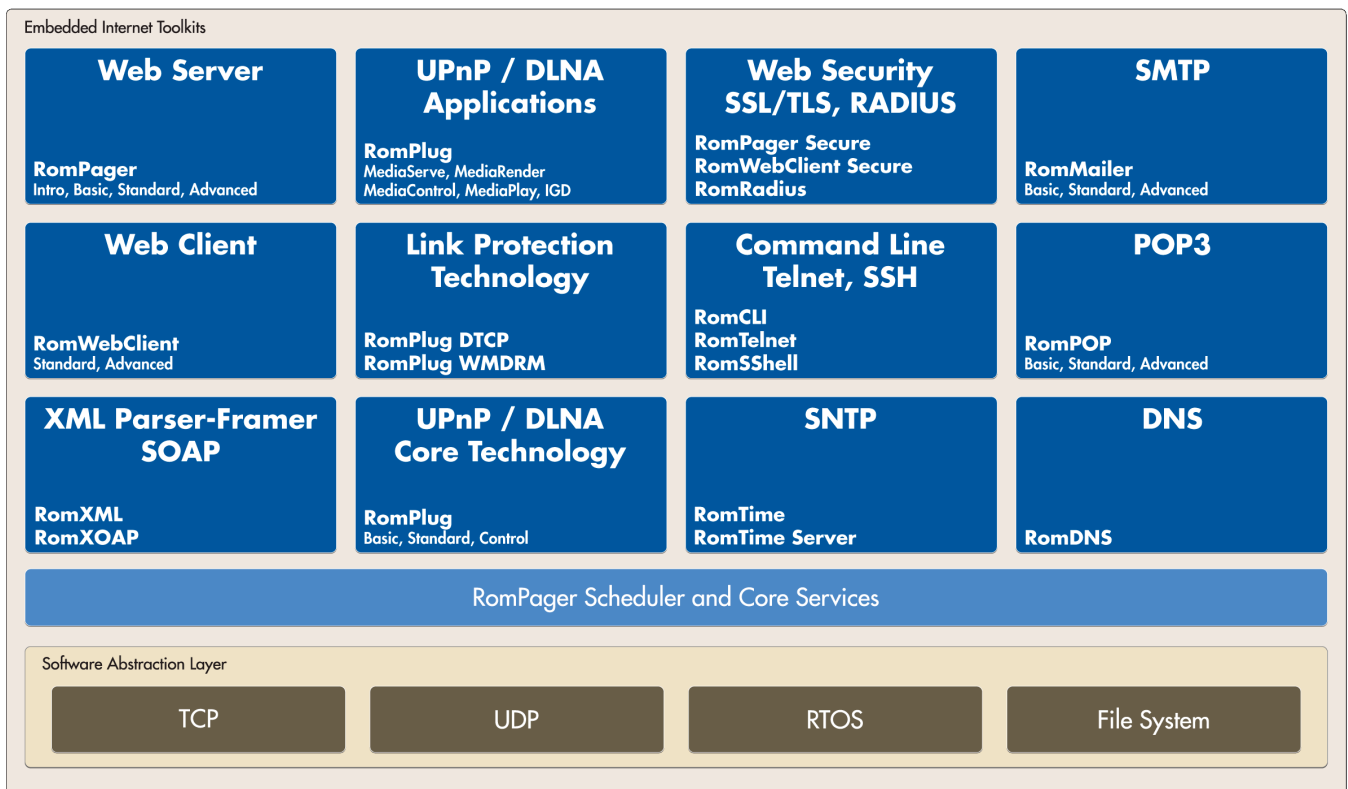
Table of Contents

Introduction	4
RomPager Intro - Web Server	8
RomPager Intro Task Flow	8
RomPager Engine	10
Delivered Source Code	10
The RomPager Engine	11
Operating System Scheduling Interface	11
Allegro and ThreadX Integration	12
Standard Library Routines	12
Operating System Time Services	13
TCP Interfaces	13
Configuring the RomPager Engine	13
Scheduling Algorithms	14
Date Handling	14
Debugging	14
TCP Connections	15
Installing RomPager Intro	16
Delivered Source Code	16
Installing with the RomPager engine	17
Configuring the RomPager Intro Options	18
Memory Allocation	18
HTTP Environment	18
Debugging Options	21
Device Name	21
Sample Code	21
RomPager Intro User Exit (CGI) Interface	22
Call Back Routines	27
RomPager Intro Sample User Exit Routine	28
AsTask.c for ThreadX	33

Introduction

The RomPager® family of products bring Web, email and other Internet services to embedded devices. The RomPager toolkits allow virtually any device to use standard Internet applications. A device can use Internet standard protocols to serve pages, images or applets; retrieve files from Web servers; and send or receive email with attachments; easily allowing Internet-based man-machine interaction. The RomPager products also allow embedded device designers to easily develop machine-to-machine systems using standard Internet techniques such as Web Services or SOAP.

The RomPager product family is provided in ANSI-C source code and has been ported to all major processor/OS platforms and is delivered with interface files for the leading OS environments. The RomPager products use a common Software Abstraction Layer to provide an interface with any RTOS, TCP/IP and file system environment. With a typical OS, the RomPager product family shares a single task/thread in the device environment using a common light-weight scheduler to support multiple simultaneous HTTP and other protocol requests. In fact, the Allegro products can run in devices without an OS, by running off the idle loop. All of the protocol products will run separately or in combination with the others. Sophisticated compiler option flags allow maximum code-sharing to provide the smallest possible code footprint.



There are four RomPager Web Server toolkits that provide a range of embedded Web server capabilities. The **RomPager Intro** toolkit is a demonstration HTTP 1.0/1.1 Web server with CGI-style user exit support. The **RomPager Basic** toolkit is an HTTP 1.0/1.1 Web server with full CGI-style user exit support and optional file system support. It uses from 7Kb to 12Kb of ROM code and provides a small, powerful server for low-end devices.

The **RomPager Standard** toolkit includes all the capabilities of the RomPager Intro toolkit and adds the PageLoader web object compiler. PageLoader imports Web pages (prepared with any Web page layout program), applets and graphics such as GIF, JPG and PNG and creates a compressed Web object library for the device.

The **RomPager Advanced** toolkit includes all the capabilities of the RomPager Standard toolkit and provides additional HTTP 1.0/1.1 features, an internal security database and the PageBuilder compiler. The PageBuilder compiler provides the features of PageLoader and a comprehensive Web Application development environment including full support for HTML (2.0, 3.2, and 4.0), XHTML, Javascript, object compression, application compression, and support for internationalization with dynamic phrase dictionaries.

Optional packages for the RomPager Advanced Web server include **SoftPages**, **Remote Host** and **Graphlets**. SoftPages adds a runtime HTML parser to RomPager and allows the device vendor to make runtime source changes to HTML pages. Remote Host provides integrated HTTP proxy services to support redirection of HTTP requests from the RomPager Web server to another Web server for retrieving objects too large to store in the embedded device. The Graphlets toolkit is a series of Java applets that provide graphic control indicators for the embedded device. The applets include line charts, bar charts, progress bars and dial indicators.

RomWebClient™ Standard is an HTTP 1.0/1.1 client that provides embedded devices the ability to retrieve and store objects from remoteWeb servers using the HTTP protocol. Objects can be in any format and can be used in a memory buffer or stored in the optional file system.

RomWebClient Advanced adds caching, cookies and pipelining capabilities. The Web clients interoperate with any standard Web server or with other embedded devices that have embedded Web servers.

The **RomPager Secure** and **RomWebClient Secure** toolkits provide SSL 3.0 and TLS 1.0, 1.1 secure server and client sessions. The encryption protocols interoperate with any secure browser or server and include RSA, RC4 DES, 3DES, SHA, and AES algorithms. The RomWebPager and RomWebClient secure toolkits offer standalone security capabilities or are available as integrated options for the RomPager server or RomWebClient toolkits.

The **RomXML™** toolkit is a small eXtensible Markup Language (XML) implementation that enables your embedded device to send (frame) and receive (parse) XML documents. Using XML in your embedded designs provides for free-format interchange of data and is widely accepted in the device management, remote sensing and enterprise IT communities. Allegro's RomXML has been designed from the ground up for use in embedded devices that often have limited resources. Written in ANSI-C, the toolkit offers built in capabilities to convert internal data between C language structures and XML documents.

The **RomXoap™** toolkit builds upon the capabilities of RomXML and offers design engineers a comprehensive solution for creating connectivity between embedded designs and enterprise IT environments utilizing standards based SOAP technology. Available as standalone toolkits or tightly integrated with the other RomPager family of products, RomXML and RomXoap provide the foundation for enabling embedded devices with XML, SOAP and Web Services capabilities.

RomPlug™ Basic is a toolkit for implementing the Device Discovery and Description sections of the UPnP™ architecture. The **RomPlug Advanced** Device toolkit adds support for the Control and Eventing sections of the UPnP architecture to build certified UPnP devices. The **RomPlug Control** toolkit provides the ability to build fully compliant UPnP architecture Control Points. Optional application toolkits provide support for **UPnP IGD**, **Media Renderer**, **Media Server**, **Media Player** and **Media Controller** devices. The **RomPlug DTCP** and **RomPlug WMDRM** toolkits allow engineering teams to easily integrate link protection into state of the art UPnP and DLNA enabled consumer electronics and mobile devices.

RomCLI™ is a toolkit that may be used to build Command Line Interfaces similar to Cisco IOS-based products. The RomCLI toolkit includes an offline compiler (CliBuilder) for preparing command definitions and also includes a built-in Telnet server as well as serial support. The **RomSShell™** toolkit provides a SSH Version 2 server that may be used by itself or as an integrated front end to RomCLI.

RomMailer™ Basic is a Simple Mail Transfer Protocol (SMTP) client that enables embedded devices to send Internet email text messages. **RomMailer Standard** adds support for attachment files using MIME and UUENCODED formats. **RomMailer Advanced** provides Delivery Status Notification and Message Delivery Notification support. When RomMailer is used with the RomPager Advanced Server, messages can also be HTML Mail with embedded graphics and dynamic insertion of variables into the message text.

RomPOP™ Basic is a toolkit for building a Post Office Protocol (POP3) client so that embedded devices can receive Internet email text messages. **RomPOP Standard** adds support for attachment files using MIME and UUENCODED formats. **RomPOP Advanced** provides Delivery Status Notification and Message Delivery Notification support.

RomDNS™ is a Domain Name Services client that provides embedded devices the ability to perform lookups of a variety of DNS records. It may be used to simplify configuration for RomMailer and RomPOP, or to provide server addresses for RomWebClient.

RomTime™ is a Network Time Protocol (NTP) client that provides embedded devices the ability to receive time services from a network time server, allowing devices to avoid requiring the user to enter the time manually. **RomTime Server** is an NTP server designed specifically for use in embedded applications.

RomTime Server is a Network Time Protocol (NTP) server that provides embedded devices the ability to provide time services to NTP client devices. With RomTime Server a device can provide synchronized time over the network so that client devices can all use the same network time and do not have to be set manually.

RomRadius is a client implementation of the RADIUS protocol that provides secure authentication services from RADIUS servers. With RomRadius, an embedded device can use the widely installed RADIUS protocol to provide additional access protection.

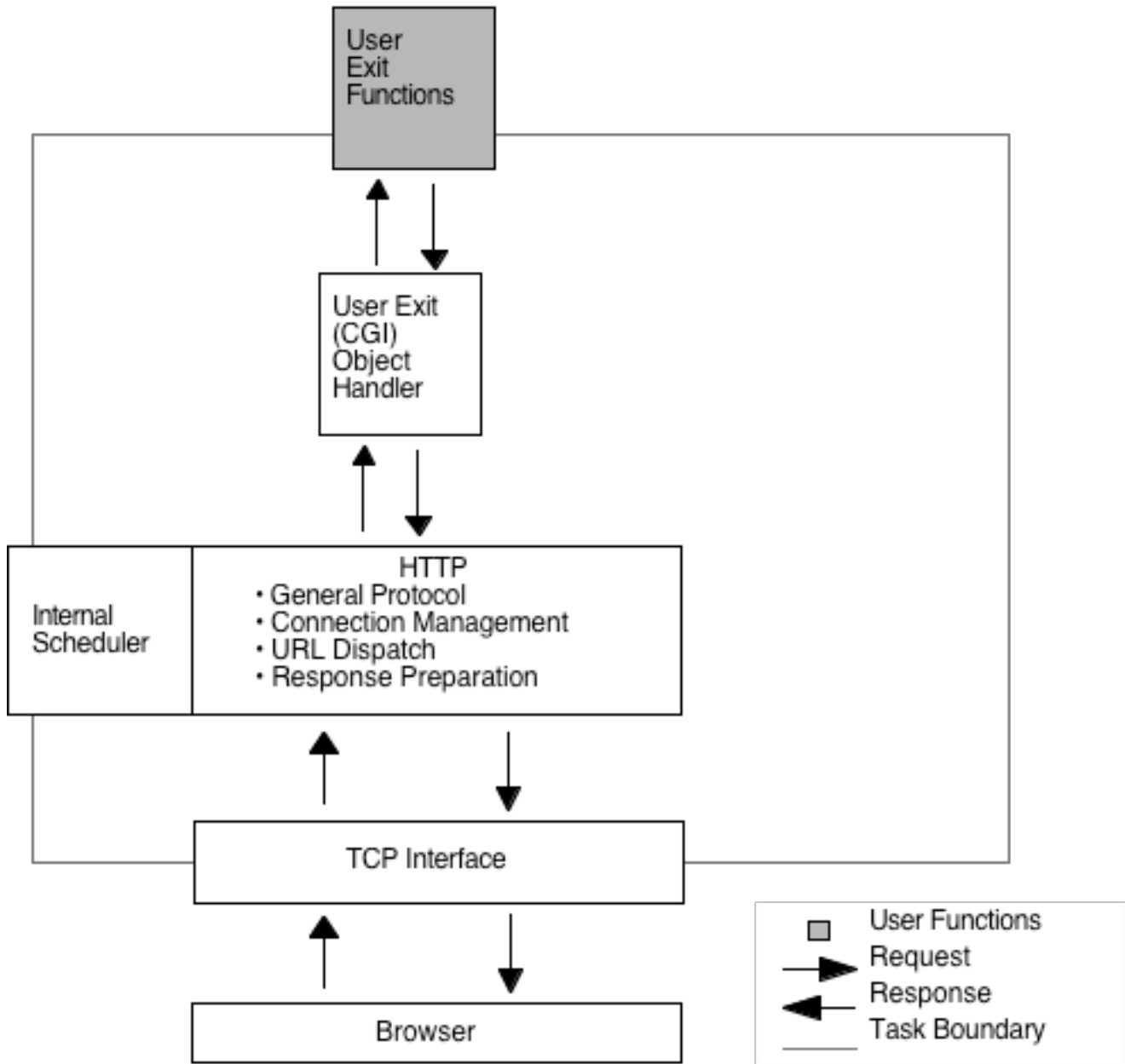
RomPager Intro - Web Server

The RomPager Intro Web Server is a basic Hyper Text Transfer Protocol (HTTP) server that serves objects to browsers or other Web clients. The RomPager Intro Web Server handles the HTTP protocol, passes URL requests to a user application, and sends formatted responses to the requesting browser or HTTP client. The interfaces between the RomPager Intro Web Server and input/output services such as TCP are described in the RomPager Product Family Porting Guide.

RomPager Intro Task Flow

In a RomPager Intro environment, the Web server runs as a single OS task, regardless of how many concurrent HTTP requests it is handling. When a URL request comes in from a browser, the Web server identifies the source of the object to be served, and either serves the object from the file system or passes the request to the user exit code for handling. The user exit code is responsible for preparing the buffers for the response. The user exit code may execute as a subroutine in the RomPager task or may optionally create other tasks to prepare the response buffers. A flow diagram is on the next page.

RomPager Intro Task Flow



The RomPager Engine

The RomPager engine is a collection of common code that is used to support all of the RomPager family of products. The engine contains a lightweight scheduler and support routines. It also provides network interface routines and various common functions such as MIME handling, header parsing, and various encoding/decoding routines.

Delivered Source Code

The source code is delivered in these directories:

Engine:

Sources:

- AsMain.c - the main entry points for the engine
- AsCommon.c - common subroutines used by the RomPager product line
- AsDate.c - the date handling routines
- AsMimes.c - the code for MIME type handling
- AsParse.c - the code for header parsing

Includes:

- AsCheck.h - consistency checks for configuration parameters
- AsChkDef.h - empty definitions required for some compilers
- AsConfig.h - the definitions used to configure the engine
- AsEngine.h - the main structure definitions used by the engine
- AsError.h - the interface error definitions
- AsExtern.h - the include file that includes the other definition files
- AsHttp.h - HTTP fragment definitions used by various toolkits
- AsMimes.h - the definitions of MIME types used by the engine
- AsParse.h - the definitions used by the header parsing code
- AsProtos.h - the interface calls to the engine
- AsStates.h - internal state definitions used by the engine
- AsStdLib.h - the macro definitions for standard library calls
- AsTarget.h - the file that defines the target RTOS environment
- AsTypes.h - the data type definitions used by the engine
- Stcp.h - the definitions used by the TCP interface

Interfaces:

ThreadX:

- AsTask.c - the Allegro engine interface for ThreadX
- Stcp.c - the TCP interface for NetX

The RomPager Engine

Operating System Scheduling Interface

The RomPager engine normally uses a single OS processing thread from ThreadX to perform its tasks. In devices with no RTOS, the RomPager engine may be run from the main idle loop. Multiple TCP connections are maintained in the TCP layer and multiple overlapping HTTP and other protocol requests are supported by the RomPager engine using internal request control blocks and a lightweight scheduler. The engine also maintains timers for connection management. The engine will use the single-task polling model to interface with the TCP/IP interface and timer services, and the engine should be called about every tenth of a second for optimum response.

The RomPager engine delivered with RomPager Intro is delivered with the interface files for ThreadX. The operating system task that calls the engine uses the calls described below.

```
void *   AllegroTaskInit(void);
void     AllegroTaskDeInit(void *theTaskDataPtr);
int      AllegroMainTask(void *theTaskDataPtr,
                        int *theHttpTasks,
                        int *theTcpTasks);
```

The RomPager engine is used in a polling model and is called from the device with three functions: `AllegroTaskInit`, `AllegroTaskDeInit`, and `AllegroMainTask`. The `AllegroTaskInit` function initializes the engine data structures and returns a pointer to the private data structure that the engine uses. The pointer to the data structure is passed in to all the other engine calls. The `AllegroTaskDeInit` function releases any dynamic memory acquired by the engine and cleans up the environment used by the engine. It should be called when the engine is shut down.

The `AllegroMainTask` function is called to perform the engine services. It returns a condition of `False` if it finds an internal error that it is unable to deal with. Each time it is called, the `AllegroMainTask` function also returns a couple of variables that provide information about the activity level of the engine. The variable `theHttpTasks` is the number of HTTP server tasks that were active during the call and the `theTcpTasks` variable is the number of TCP connections that were active for that call. The active number of TCP connections is likely to be higher than the number of HTTP server tasks if HTTP server requests are being queued or if other protocols such as the HTTP client or mail clients are active. This information can be used by the calling task to vary calling priorities or to determine levels of activity.

An example flow of the device calls to the engine is shown below. The file `AsProtos.h` contains the prototypes to include in your code.

```
/* Start of task */
theTaskIsUp = false;
theTaskData = AllegroTaskInit();
if (theTaskData != (void *) 0) {
    theTaskIsUp = true;
}
while (theTaskIsUp) {
    /*
     * Wait here for system time and
     * give up time to other threads.
     */
    theOkFlag = AllegroMainTask(theTaskData, &theHttpTasks, &theTcpTasks);
    if (!theOkFlag) {
        AllegroTaskDeInit(theTaskData);
        theTaskIsUp = false;
    }
}
/* End of task */
```

When the `AllegroMainTask` routine is called, each potential TCP connection is checked to see if there is something to do. If there is, the appropriate internal protocol task will be run to the next I/O breaking point (a packet read or write call, for example). After all the connections have been checked, the internal timer task will be run to check for various timeout conditions.

Allegro and ThreadX Integration

The interface routine between the Allegro engine and the ThreadX RTOS is `AsTask.c`. This routine initializes the appropriate threads for the NetX environment. A sample version of `AsTask.c` is at the end of this manual.

Standard Library Routines

The RomPager engine uses a number of C standard library functions such as `malloc`, `free`, `memset`, `strcpy` and `strcat`. The RomPager source code uses a series of macros such as `RP_MALLOC` and `RP_STRCAT`, so that if a device environment uses non-standard functions, they may be easily redefined. The `AsStdLib.h` file defines these macros and the underlying functions. The example routines provided with some of the Allegro products also use macros to access the C standard library functions. These macros are defined in the `ExStdLib.h` file.

Operating System Time Services

The RomPager engine is delivered with a variety of interface files for many of the popular operating environments. The `AsTarget.h` file is used to select which of the supported operating systems is being used. By selecting the ThreadX operating system, the configuration choices in `AsDate.c` will be made for you automatically.

The RomPager engine needs support from the operating system for timing/date services. The routines in `AsDate.c` may need to be customized for the services that your device operating system provides. Normally, the routines in `AsDate.c` use the standard C runtime library routines. If these routines are not available, then set the `RomPagerUseStandardTime` compilation flag to 0 and the Allegro provided routines will be used. The main time routine that the porter needs to be concerned with is the `RpGetSysTimeInSeconds` routine. The `RpGetSysTimeInSeconds` routine provides the engine either the calendar date/time in seconds or the number of seconds since the system was powered on depending on whether the `RomPagerCalendarTime` compilation flag is set.

The other time routines in `AsDate.c` are the `RpGetDateInSeconds`, and `RpBuildDateString` routines. These routines will use the standard C runtime library routines: `mktime`, `localtime`, and `strftime`. If the standard time library routines are not available, the Allegro provided routines will be compiled. If other equivalent library routines exist elsewhere in the system, the porter can save some memory by altering this module. Full documentation of the routines can be found in the `AsDate.c` source file.

TCP Interfaces

The RomPager engine needs to interface to TCP/IP services to be able to send and receive TCP packets for the HTTP application protocol. The RomPager engine is delivered with TCP interface files for the NetX TCP environment.

Configuring the RomPager Engine

The RomPager engine contains a number of definitions and compilation options that affect its memory size and/or performance. The RomPager Intro product is delivered with the engine pre-configured for standard ThreadX and NetX usage. You may want to look at the `AsConfig.h` file if you want to make changes to the default values.

Scheduling Algorithms

The Allegro scheduler uses a single task from the host OS environment and provides its own internal scheduler for handling simultaneous TCP connections. Each potential separate TCP connection maintains a state and other parameters in a connection control block. The number of connections to be managed is determined by `kStcpNumberOfConnections` which is defined in `Stcp.h`. The connection control block structure is the `rpConnection` structure which is defined in the `RomPager.h` file.

The entry point to the Allegro scheduler is the `AllegroMainTask` routine. When this routine is called, each connection will be checked to see if there is something to do. If there is, the internal task will be run to the next I/O breaking point (a packet read or write call, for example). After all the connections have been checked, the internal timer task will be run to check for various timeout conditions. The `AllegroMainTask` routine should be called at least once per second for timing functions and more often when TCP activity is occurring. A simple way to run the server without limiting performance on the rest of the system is to call `AllegroMainTask` once every 10th of a second.

Date Handling

The `kHttpRomMonth`, `kHttpRomDay` and `kHttpRomYear` variables are used to set up the internal date of the code image.

If `RomPagerUseStandardTime` is defined as 1, the date code will use the standard C library time calls. If your device is already using these libraries, you should set this flag. Otherwise, additional code (using about 500 bytes) equivalent to these libraries will be generated.

If `RomPagerCalendarTime` is defined as 1, date headers will be created using the device date. If a calendar date is not available, a header date will be created using the internal code image date plus the number of seconds since system boot.

Debugging

If `AsDebug` is set, various support routines and error checking code for tuning and debugging will be generated. In production versions, this flag should be turned off. This flag also allows various product specific debugging flags to be enabled in the individual product configuration files.

TCP Connections

The total number of TCP connections that the Allegro scheduler manages is defined by the value `kStcpNumberOfConnections` in `Stcp.h`. The Web server products use passive connections to wait for a browser to initiate an HTTP request on the connection. Make sure that the total pool is large enough to handle the expected number of simultaneous server connections.

Terminating Connections

There are a few cases where the engine may need to terminate a connection to free it up so that other requests can use it. Different implementations of TCP handle connection closings in different ways, and may need the RomPager engine to perform connection cleanup. The first case is that when a connection closes, it sometimes is left in the TCP `TIME_WAIT` state which depending on implementation can be a long time (4 minutes) after the connection has been closed. In the Web server environment each request is typically a different connection, so a lot of connections are opened and closed. If each connection has to wait 4 minutes to free up its resources, a lot of connections and internal resources will be used. Some browsers just abort the connection to avoid this problem. Persistent connection approaches such as “Keep-Alive” and HTTP 1.1 reduce this problem, but using connection timeouts reduces the problem for all browsers. By setting the `kConnectionCloseTimeout` variable to a low value, the connection will be aborted after spending a small amount of time in the `TIME_WAIT` state, so that fewer total connections are required.

In Web server environments, a second case can be caused when an impatient user hits the reload button in a browser, while a request is in process. The browser will close the connection that the receive has started on, but not all TCP implementations are able to signal the remote side close. This can leave a connection tied up waiting indefinitely for a receive that was reissued on another connection. The `kConnectionReceiveTimeout` value is used to abort connections where a receive was started and no data has been received.

The third case is when persistent Web server connections are being used with “Keep-Alive” or HTTP 1.1. A persistent connection can reduce the overhead for a single user, but can leave a connection tied up that another user might need. The `kConnectionMaxIdleTimeout` value is used to set the maximum idle time period. If another request is not made in this period, then the connection will be closed.

Installing RomPager Intro

Delivered Source Code

The source code is delivered in these directories:

RomPager Intro:

Sources:

- RiCallBk.c - the callback routines used with RomPager Intro
- RiHtml.c - the routines to handle error page display
- RiUrl.c - the routines for URL dispatching
- RpCgi.c - User Exit interface routines
- RpData.c - Internal error pages and system phrase dictionary
- RpFrmItm.c - Form item handling routines
- RpHttp.c - RomPager implementation of HTTP protocol
- RpHttpHd.c - Header response routines used for HTTP protocol
- RpHttpPs.c - parsing routines used for HTTP protocol
- RpHttpRq.c - HTTP request handling routines

Includes:

- RiConfig.h - the definitions used to configure RomPager Intro
- RiCallBk.h - the callback routine prototypes used with RomPager Intro
- RiCgi.h - the Web server User Exit definitions
- RiIntPrt.h - the internal prototype definitions
- RiPages.h - the definitions used by the internal error pages
- RpStates.h - the Web server state definitions
- RomPager.h - internal Embedded Web Server Definitions

Samples:

- RiCgiTst.c - Test code to exercise the RomPager Intro Web server
- RiCgiTs1.c - Alternate test code to exercise RomPager Intro

Installation with the RomPager Engine

The RomPager product family provides a central engine with its own scheduler and common routines for resource control. The configuration of the engine is done with the `AsConfig.h` file that is included with the engine. To install RomPager Intro, install the source files in the same directory as the RomPager engine. The `RiCgiTst.c` file includes some sample code for exercising the Web server. The `RiCgiTs1.c` file takes the same samples as the `RiCgiTst.c` file and adds some URL dispatching code.

Configuring the RomPager Intro Options

The RomPager Intro engine contains a number of definitions and compilation options that affect its memory size and/or performance. The file `RiConfig.h` needs to be examined and a number of choices need to be made.

Memory Allocation

The RomPager Intro server uses static memory allocation. Fixed global memory requirements are about 3Kb for the base engine and about 5Kb per HTTP request control block. Each simultaneous HTTP connection uses one HTTP request control block.

The HTTP request control block contains one buffer of `kHttpWorkSize` and two buffers of `kHtmlMemorySize`. Changing the size of these buffers will thus control the amount of memory required for a request control block. The buffer of `kHttpWorkSize` is used to prepare HTTP headers for transmission and to receive form data from the browser. If your system has large form entries you may need to increase `kHttpWorkSize` above the default size of 512. If only small forms are used, you may be able to reduce the size for increased memory savings. The two buffers of `kHtmlMemorySize` are used to prepare the HTML object responses for transmission to the browser. A double buffered scheme is used for maximum I/O overlap. The default size of 1450 fits well in an Ethernet TCP packet. If smaller sizes are used, you will be able to save on memory usage with a tradeoff that more packets will be sent for larger pages, images, and applets.

HTTP Environment

HTTP 1.0 Compliance

These flags control HTTP specification compliance. Strict conformance requires that they be set to 1. For a variety of reasons, most browsers (and all the popular ones) will work if they are set to 0, thus saving code space and processing time.

If `RomPagerMimeTypeChecking` is defined as 1, the code to check the HTTP Accept statements will be enabled. Since the popular browsers will accept any type and RomPager serves all objects with proper Mime Types, it isn't necessary to process the Accept statements.

If `RomPagerFullHostName` is defined as 1, the code to generate a host name from the IP address and port will be included. Since most browsers (Netscape 2.0 or later) pass in the host name with the request, this code is generally not necessary. The HTTP 1.1 specifications require browsers to send in the host name, so over time this code will be even less necessary.

HTTP 1.0 Options

Netscape created a “keep alive” technique for persistent connections in a HTTP 1.0 environment. Microsoft also supports this technique. The “keep alive” persistent connections became the basis for the initial definition of HTTP 1.1 persistent connections, but without the use of chunked encoding to signal object length, the technique provides less benefits for dynamically generated pages. If `RomPagerKeepAlive` is defined as 1, the code to maintain persistent connections using the Netscape “keep alive” technique will be generated.

HTTP Browser Headers

Browsers send various HTTP headers that may be useful to the management application. `RomPager` provides optional capabilities to examine these headers.

The `Accept-Language` HTTP header is sent by browsers and may provide some information about what languages a browser user is capable of displaying. This header is sent in different ways by different browsers and is not always a reliable indicator of the user’s desires. In general, if an application provides multiple language support, it is better to ask the user explicitly which language to display the pages in. If `RomPagerCaptureLanguage` is defined as 1, the code to capture the `Accept-Language` HTTP header that the browser sends in will be generated. Since this option adds processing overhead and 256 bytes of RAM for each HTTP request, this flag should only be turned on if the `RpGetAcceptLanguage` callback routine is used.

The `User-Agent` header is sent by browsers and indicates which browser is running. If your management application is sending different pages to different browsers, you will need to look at this header. If `RomPagerCaptureUserAgent` is defined as 1, the code to capture the `User-Agent` HTTP header will be generated. Since this option adds processing overhead and 256 bytes of RAM for each HTTP request, this flag should only be turned on if the `RpGetUserAgent` callback routine will be used.

HTTP 1.1

The HTTP 1.1 protocol is defined by RFC 2068 and additional drafts that will lead to additional RFCs. As of RomPager Release 3.00, HTTP 1.1 is supported by Microsoft IE 4.0 and IE 5.0 which are based on RFC 2068 with modifications. The RomPager HTTP 1.1 code has been tested and it interoperates with MSIE 4.0 and MSIE 5.0. Since this feature requires extra code space, and is not supported by all browsers, you may wish to turn this feature off for production builds.

If `RomPagerHttpOneDotOne` is defined as 1, the code to build an HTTP 1.1 server (RFC 2068 + draft modifications) will be generated. HTTP 1.1 can provide additional performance with the use of persistent connections and chunked encoding. These options add approximately 3000 bytes to the server code. HTTP 1.1 browsers know how to talk to HTTP 1.0 servers, so this capability may not be a requirement for all applications. If this option is enabled, the RomPager server will respond with an identification as an HTTP 1.1 server to all requests (1.0 and 1.1). The type of client request will be identified, and the appropriate headers generated. That is, HTTP 1.0 client requests will receive HTTP 1.0 response headers (which are a subset of HTTP 1.1 headers), and HTTP 1.1 client requests will receive HTTP 1.1 response headers.

The HTTP 1.1 specifications have compliance requirements and optional features that may not be required for interoperability. If memory is tight, these options can be turned off and the RomPager server will still interoperate with mainstream production HTTP 1.1 browsers.

HTTP 1.1 Compliance

To allow for transition to future versions of HTTP, Section 5.1.2 of the specifications requires compliant servers to support absolute URIs even though HTTP 1.1 clients will not generate these requests. Support for absolute URIs is enabled by the `RpAbsoluteUri` flag.

<code>RomPagerMimeTypeChecking</code>	225 bytes
<code>RomPagerFullHostName</code>	125 bytes
<code>RomPagerKeepAlive</code>	325 bytes
<code>RomPagerCaptureLanguage</code>	100 bytes
<code>RomPagerCaptureUserAgent</code>	100 bytes
<code>RomPagerHttpOneDotOne</code>	3000 bytes
<code>RpAbsoluteUri</code>	150 bytes

Debugging Options

If `RomPagerDebug` is set, various support routines and error checking code for tuning and debugging will be generated. In production versions, this flag should be turned off.

If `RpHttpFlowDebug` is set to 1, then a set of `printf` statements that track the flow of the HTTP processing will be generated in the engine. A `printf` statement will show the path requested, the response, and the object buffer being sent.

If `RpNoCache` is set to 1, then HTTP objects that are normally static such as help text pages, graphics and applets will be sent to the browser as dynamic objects so that they are not cached. This can be useful during the debugging/development process. The flag should be turned off for production builds, to support best browser performance.

Device Name

The name of the device that is used in error message pages defaults to "RomPager server" the value of `kRpBoxName`. If you want to change the device name, change the value of `kRpBoxName`.

Sample Code

If `RpDispatch` is defined as 0, the sample code (`RiCgiTst.c`) that uses simple URL dispatching will be enabled. If `RpDispatch` is defined as 1, the sample code (`RiCgiTs1.c`) that uses more advanced URL dispatching will be enabled.

RomPager Intro User Exit (CGI) Interface

The RomPager Intro Web server uses a user exit routine that provides CGI-like handling for most URL requests. If the URL request is not satisfied from the file system, the RomPager web server will pass control to the user exit routine by issuing the `RpExternalCgi` call. The call passes a condensed form of the information in the HTTP request to the external CGI routine using the `rpCgi` control block, and looks for responses from the CGI routine in the same structure. The interface definitions from the `RiCgi.h` file are at the end of this section.

The arguments passed to the CGI routine include the number of the TCP connection which is passed in `fConnectionId`. The HTTP request type (GET, HEAD, or POST) is passed as an enum in `fHttpRequest`. The URL is passed in `fPathPtr` and the contents of various HTTP headers are passed in `fHostPtr`, `fRefererPtr`, `fAgentPtr`, and `fLanguagePtr`. The contents of the Date header are passed in `fBrowserDate` after being converted to seconds since 1/1/1900. The `fArgumentBufferPtr` and `fArgumentBufferLength` fields point to any query arguments appended to a GET request, or the object body of a POST request.

The `fRequestState` field is used by the Web server to signal the state of the incoming request object buffer. The values for the `fRequestState` field are `eRpCgiPartialRequest`, and `eRpCgiRequestComplete`. HTTP requests that do not have a request object such as GET and HEAD requests will have the `fRequestState` field set to `eRpCgiRequestComplete`. Most POST requests with small amounts of forms data will also have the `fRequestState` field set to `eRpCgiRequestComplete`. The size of the request object buffer is controlled by the `kHttpRequestSize` variable in `RiConfig.h`. If there is more data for the HTTP request object than will fit in one buffer, the first buffer(s) of the object data will be sent with the `fRequestState` field set to `eRpCgiPartialRequest` and the last buffer will be sent to the User Exit routine with the `fRequestState` field set to `eRpCgiRequestComplete`.

The `fResponseState` field is used by the User Exit routine to signal its processing state. The values for the `fResponseState` field are `eRpCgiPending`, `eRpCgiPendAsynch`, `eRpCgiRequestAck`, `eRpCgiBufferComplete`, and `eRpCgiLastBuffer`. The values of `eRpCgiPending`, `eRpCgiPendAsynch` and `eRpCgiRequestAck` are used while handling the incoming HTTP request and the values of `eRpCgiPending`, `eRpCgiPendAsynch`, `eRpCgiBufferComplete`, and `eRpCgiLastBuffer` are used when preparing the response.

For CGI processes that need to run asynchronously, the user exit routine can signal this using the `eRpCgiPending` response or the `eRpCgiPendAsynch` response. If the `eRpCgiPending` response is returned, the RomPager Web server will issue another `RpExternalCgi` call at a later time to get the results of the CGI processing. If the `eRpCgiPendAsynch` response is returned, the RomPager Web server will not call the CGI process again. In order to resume Web server processing, the CGI process must wake up RomPager with the `RpCompleteCgi` call and pass in a pointer to the `rpCgi` control block that has the updated response information.

When a partial HTTP request is received with the `fRequestState` field set to `eRpCgiPartialRequest`, the User Exit application should signal that it is ready to receive the next HTTP object buffer by setting the `fResponseState` field to `eRpCgiRequestAck`.

After the User Exit application receives control with the `fRequestState` field set to `eRpCgiRequestComplete`, it can prepare the response to the HTTP request and set the appropriate value in the `fResponseState` field. If the `eRpCgiLastBuffer` state is returned, then the Web server knows the CGI process is complete and will send the response back to the browser client. If the `eRpCgiBufferComplete` state is returned, the Web server will send the response back to the browser client and issue another `RpExternalCgi` call to gather another buffer of the response.

For cases where the CGI process needs to free a data buffer after the web server has sent the response to the browser client, the CGI process should set the `eRpCgiBufferComplete` state for the last buffer sent to the web server. Then when the CGI process is called again by the web server, it should respond with a state of `eRpCgiLastBuffer` and the `fResponseBufferLength` field set to 0 after the data buffer has been released.

The `fUserDataPtr` field can be set to point to arbitrary user exit data. It is initialized by the Web server to a null value for each request, and then ignored. The user exit routine may use the field to store information that will be used for handling multi-buffer responses for a single HTTP request. The user application can insert a pointer to local information in the field, or a count, or anything else that it wants to get back the next time it receives the `RpExternalCgi` call.

The `fHttpResponse`, `fDataType` and `fObjectDate` fields are used to tell the engine which HTTP headers to prepare for the response.

The `fHttpResponse` field for a normal response will be `eRpCgiHttpOk` for a dynamically prepared object and `eRpCgiHttpOkStatic` for a static object such as a GIF or JPEG image. The `eRpCgiHttpRedirect` response is used after processing a form to tell the browser which page to retrieve next. The `eRpCgiHttpNotModified` response is used for requests for a static object that have been previously filled. This response can be used to save sending the object to the browser again. The `eRpCgiHttpNotFound` response is used to notify the browser that the CGI routine was not able to handle this request. The response of `eRpCgiHttpNoContent` is used to notify the browser that no data will be returned for this request.

The `fResponseBufferPtr` and `fResponseBufferLength` fields point to the HTML response buffer prepared by the external CGI routine. If the `fHttpResponse` field is `eRpCgiHttpRedirect`, then the external CGI routine needs to provide the URL to redirect to in the response buffer. If the `fHttpResponse` field is `eRpCgiHttpUnauthorized`, then the external CGI routine needs to provide the realm name in the response buffer.

The values for `fDataType` are the MIME types specified in `AsMimes.h`. If the MIME type is not in the list supported by `AsMimes.h`, the value of `eRpDataTypeOther` may be returned in `fDataType` with the actual string value in `fOtherMimeType`.

The value of `fObjectDate` is a date in internal RomPager format (number of seconds since 1/1/1900). If the object is a static object, the `fObjectDate` field is used to prepare the HTTP headers. If the object is a dynamic object, the current system date and time information will be used.

Request arguments from HTML forms are passed as a group of name/value pairs encoded in a format called 'Form URL Encoding'. The `RpGetFormItem` routine is used to decode and retrieve a name/value pair from the query buffer. The buffer pointer will be updated to point to the next name/value pair and the values of the current name/value pair will be copied into the buffers passed as input to the call.

The support for Web server User Exit routines includes an initialization routine and deinitialization routine. The initialization routine, `RpUserExitInit`, is called once when the main engine starts up and can be used by the application to initialize resources necessary for the User Exit routines. The deinitialization routine, `RpUserExitDeInit`, is called once when the main engine is shutting down and can be used by the application to free up resources.

The User Exit interface definitions from `RiCgi.h` are shown below, and a sample user exit routine that serves a few pages is shown in the next section.

```

/*
   Cgi HTTP requests
*/

typedef enum {
    eRpCgiHttpGet = 1,      /* Cgi request is HTTP GET          */
    eRpCgiHttpHead,        /* Cgi request is HTTP HEAD        */
    eRpCgiHttpPost,        /* Cgi request is HTTP POST        */
    eRpCgiHttpPut          /* Cgi request is HTTP PUT         */
} rpCgiHttpRequest;

/*
   Cgi request status
*/

typedef enum {
    eRpCgiPartialRequest,  /* Buffer ready - there are more buffers */
    eRpCgiRequestComplete /* Request complete - last request buffer */
} rpCgiRequest;

/*
   Cgi call responses
*/

typedef enum {
    eRpCgiPending,        /* Cgi processing incomplete          */
    eRpCgiPendAsynch,     /* Cgi processing incomplete - complete asynch */
    eRpCgiRequestAck,     /* Cgi acknowledge - ready for next buffer */
    eRpCgiBufferComplete, /* Cgi buffer ready - more buffers coming */
    eRpCgiLastBuffer     /* Cgi buffer ready - no more buffers */
} rpCgiResponse;

/*
   Cgi HTTP responses
*/

typedef enum {
    eRpCgiHttpOk,          /* Cgi returns HTTP 200 Ok          */
    eRpCgiHttpOkStatic,   /* Cgi returns HTTP 200 Ok - Static Object */
    eRpCgiHttpCreated,    /* Cgi returns HTTP 201 Created      */
    eRpCgiHttpNoContent,  /* Cgi returns HTTP 204 No Content   */
    eRpCgiHttpRedirect,   /* Cgi returns HTTP 302 Moved Temp  */
    eRpCgiHttpNotModified, /* Cgi returns HTTP 304 Not Modified */
    eRpCgiHttpBadRequest, /* Cgi returns HTTP 400 Bad Request  */
    eRpCgiHttpUnauthorized, /* Cgi returns HTTP 401 Unauthorized */
    eRpCgiHttpNotFound,   /* Cgi returns HTTP 404 Not Found    */
    eRpCgiHttpIntServerErr /* Cgi returns HTTP 500 Internal Server Err */
} rpCgiHttpResponse;

```

```

/*
    rpCgi structure
*/
typedef struct {
    /*
        Request fields
    */
    Unsigned8          fConnectionId;
    rpCgiRequest       fRequestState;
    rpCgiHttpRequest   fHttpRequest;
    char *             fPathPtr;          /* URL */
    char *             fHostPtr;         /* Host: */
    char *             fRefererPtr;     /* Referer: */
    char *             fAgentPtr;       /* User-Agent: */
    char *             fLanguagePtr;    /* Content-Language: */
    Unsigned32         fBrowserDate;    /* Date: (internal) */
    char *             fArgumentBufferPtr;
    Signed32           fArgumentBufferLength;
    /*
        Request/Response fields
    */
    void *             fUserData;        /* Arbitrary User Data */
    /*
        Response fields
    */
    rpCgiResponse      fResponseState;
    rpCgiHttpResponse  fHttpResponse;
    char *             fResponseBufferPtr;
    Signed32           fResponseBufferLength;
    rpDataType         fDataType;
    char               fOtherMimeType[kMaxMimeTypeLength];
    Unsigned32         fObjectDate;     /* Object Date (internal) */
} rpCgi, *rpCgiPtr;

/*
    The External User Interface call
*/
extern void RpExternalCgi(void *theTaskDataPtr, rpCgiPtr theRpCgiPtr);

/*
    Routines to initialize and de-initialize User Exit resources.
*/
extern RpErrorCode RpUserExitInit(void *theTaskDataPtr);
extern RpErrorCode RpUserExitDeInit(void);

/*
    Callback routine for asynch completion.
*/
extern void RpCompleteCgi(void *theTaskDataPtr, rpCgiPtr theCgiPtr);

```

Call back routines

There are a variety of call back routines that may be issued from the User Exit application to control internal states of the web server.

Form Item Handling

```
extern void RpGetFormItem(char ** theBufferPtr,  
                          char * theNamePtr,  
                          char * theValuePtr);
```

Request arguments from HTML forms are passed as a group of name/value pairs encoded in a format called 'Form URL Encoding'. The `RpGetFormItem` routine is used to decode and retrieve a name/value pair from the form buffer. The buffer pointer will be updated to point to the next name/value pair and the values of the current name/value pair will be copied into the buffers passed as input to the call.

Time Access Routines

```
extern Unsigned32 RpGetMonthDayYearInSeconds(void *theTaskDataPtr,  
                                              Unsigned32 theMonth,  
                                              Unsigned32 theDay,  
                                              Unsigned32 theYear);  
extern Unsigned32 RpGetSysTimeInSeconds(void *theTaskDataPtr);
```

The `RpGetMonthDayYearInSeconds` routine is used to translate an external date into internal format which is number of seconds since 1/1/1900. The values of `theMonth` are from 1 to 12. The values of `theDay` are from 1 to 31. The value of `theYear` is the full four digit year, for example, 2003. The `RpGetSysTimeInSeconds` routine is used internally by the RomPager engine and may also be used by the user application routines. It returns the current system time in internal format.

Connection Control Routine

```
extern void RpSetConnectionClose(void *theTaskDataPtr);
```

The `RpSetConnectionClose` routine is used to force the TCP connection being used for the current HTTP request to close after the current HTTP request is completed. Both Netscape "keep alive" support and HTTP 1.1 persistent connections attempt to keep the TCP connection open for multiple HTTP requests from a single browser. In some cases, it may be useful for the server to free up a TCP connection for other users or other TCP applications even if the browser has requested to keep the connection open.


```
/*
   This routine initializes any resources used by the
   User Exit feature.
   It is called once during RomPager initialization.
*/
RpErrorCode RpUserExitInit(void *theTaskDataPtr) {
    gUserExitSeconds = RpGetMonthDayYearInSeconds(theTaskDataPtr, 10, 26, 1998);
    return eRpNoError;
}

/*
   This routine de-initializes any resources used by the
   User Exit feature.
   It is called once during RomPager de-initialization.
*/
RpErrorCode RpUserExitDeInit() {
    return eRpNoError;
}
```

```

/*
   This routine is called for each URL that the RomPager Intro
   Web server processes. This routine is responsible for formatting
   the response that the Web server will send to the browser.
*/
extern void RpExternalCgi(void *theTaskDataPtr, rpCgiPtr theCgiPtr) {
    char *    theFormBufferPtr;
    Boolean   theFoundFlag;
    char  theName[25];
    char  theValue[25];

    /*
       All of the test pages are single buffer responses
       with no waiting.
    */
    theCgiPtr->fResponseState = eRpCgiLastBuffer;
    /*
       See what kind of command we got.
    */
    if (theCgiPtr->fHttpRequest == eRpCgiHttpGet) {
        /*
           The command was a GET, so see what page
           we got the GET request for.
        */
        if (RP_STRCMP(theCgiPtr->fPathPtr, "/Main") == 0 ||
            RP_STRCMP(theCgiPtr->fPathPtr, "/") == 0) {
            /*
               The request is for the root page which is a normal page.
            */
            theCgiPtr->fHttpResponse = eRpCgiHttpOk;
            theCgiPtr->fResponseBufferPtr = UserExitMainPage;
            theCgiPtr->fResponseBufferLength = RP_STRLEN(UserExitMainPage);
        }
        else if (RP_STRCMP(theCgiPtr->fPathPtr, "/Another") == 0) {
            /*
               This request is for another page. We check here
               to see whether the browser seen this page before.
               If so, we can save some transfer time by telling the
               browser that the page is unchanged. If not, send the page.
            */
            if (theCgiPtr->fBrowserDate == gUserExitSeconds) {
                /*
                   Yes, so tell the browser that it is unchanged.
                */
                theCgiPtr->fHttpResponse = eRpCgiHttpNotModified;
            }
            else {
                /*
                   No, so we serve another normal page as static.
                */
                theCgiPtr->fHttpResponse = eRpCgiHttpOkStatic;
                theCgiPtr->fObjectDate = gUserExitSeconds;
                theCgiPtr->fResponseBufferPtr = UserExitAnotherPage;
                theCgiPtr->fResponseBufferLength =
                    RP_STRLEN(UserExitAnotherPage);
            }
        }
    }
}

```

```

else if (RP_STRCMP(theCgiPtr->fPathPtr, "/Redirect") == 0) {
    /*
        The browser is asking for a page that we are
        storing under a different URL, so send it there.
    */
    theCgiPtr->fHttpResponse = eRpCgiHttpRedirect;
    theCgiPtr->fResponseBufferPtr = UserExitAnotherPath;
    theCgiPtr->fResponseBufferLength = RP_STRLEN(UserExitAnotherPath);
}
else if (RP_STRCMP(theCgiPtr->fPathPtr, "/FormDisplay") == 0) {
    /*
        A normal page with a form on it.
    */
    theCgiPtr->fHttpResponse = eRpCgiHttpOk;
    theCgiPtr->fResponseBufferPtr = UserExitDisplayFormPage;
    theCgiPtr->fResponseBufferLength =
        RP_STRLEN(UserExitDisplayFormPage);
}
else {
    /*
        The command was a GET request, but we don't have a
        handler for the page the browser is looking for,
        so tell the browser that no page was found.
    */
    theCgiPtr->fHttpResponse = eRpCgiHttpNotFound;
}
}
else if (theCgiPtr->fHttpRequest == eRpCgiHttpPost) {
    /*
        We got a POST request, so see if it matches the form that
        we know.
    */
    if (RP_STRCMP(theCgiPtr->fPathPtr, "/ProcessForm") == 0) {
        /*
            This is our form, so go retrieve the values.
        */
        theFormBufferPtr = theCgiPtr->fArgumentBufferPtr;
        theFoundFlag = False;
        while (!theFoundFlag && *theFormBufferPtr != '\0') {
            RpGetFormItem(&theFormBufferPtr, theName, theValue);
            if (RP_STRCMP(theName, "The text") == 0) {
                theFoundFlag = True;
            }
        }
        /*
            Set up the form response page.
        */
        RP_STRCPY(gUserExitFormResponseBuffer, UserExitFormResponse1);
        RP_STRCAT(gUserExitFormResponseBuffer, theValue);
        RP_STRCAT(gUserExitFormResponseBuffer, UserExitFormResponse2);
        theCgiPtr->fHttpResponse = eRpCgiHttpOk;
        theCgiPtr->fResponseBufferPtr = gUserExitFormResponseBuffer;
        theCgiPtr->fResponseBufferLength =
            RP_STRLEN(gUserExitFormResponseBuffer);
    }
    else {
        /*
            Treat any other POST request as a no page found.
        */
        theCgiPtr->fHttpResponse = eRpCgiHttpNotFound;
    }
}
}

```

```
else {
    /*
        The request was not a GET or a POST, so we'll just
        assume that it is a HEAD request, and return OK
        with no content sent, regardless of the URL requested.
    */
    theCgiPtr->fResponseState = eRpCgiLastBuffer;
    theCgiPtr->fHttpResponse = eRpCgiHttpOk;
    theCgiPtr->fResponseBufferLength = 0;
}
return;
}
```

AsTask.c for ThreadX

The code below shows a sample of the `AsTask.c` routine that is included with RomPager Intro Web server for ThreadX. It shows how the RomPager server task is started in a ThreadX environment.

The `AsTask.c` module defines the `main()` and `tx_application_define()` functions which are required for ThreadX. These two functions together is where the ThreadX operation system begins running the RomPager application. The `tx_application_define()` function tells ThreadX to create the Main RomPager and Allegro threads, as well as create the packet pool and the IP control block/thread task which is where ThreadX/NetX does the muscle work of TCP/IP operations. The IP control block is created with appropriate user defined driver (Ethernet network) entry function specific to the device processor type. Lastly, it enables the IP instance for TCP.

In creation of the Main and Allegro threads, the `tx_thread_create()` call includes their respective thread entry functions for executing these threads. These functions are also defined in `AsTask.c`. The `MainThreadEntry()` function is where user application code should be added to customize their host HTTP server application.

`AsTask.c` also contains user configurable options for setting stack size, packet pool payload and packet pool size for the Allegro packet pool, and device host IP address and network mask.

When the `tx_application_define()` function completes, it hands control back to the ThreadX scheduler which will schedule the `MainThreadEntry()` and `AllegroThreadEntry()` entry functions to run.

```
/*
 * File:          AsTask.c
 *
 * Contains:     Main task for Allegro products for the ThreadX platform
 *
 */

#include "tx_api.h"
#include "nx_api.h"
#include "AsProtos.h"

/*
 * Local definitions.
 */

#define ASTASK_DEBUG    1
```

```

#define DEVICE_IP_ADDRESS      IP_ADDRESS(1, 2, 3, 4)
#define DEVICE_NETWORK_MASK   IP_ADDRESS(255, 255, 255, 0)
#define ALLEGRO_STACK_SIZE    (2048)
#define MAIN_STACK_SIZE       (2048)

/*
 *   Exported global data.
 */

NX_IP *      gNetxIpInstancePtr;

#define PACKET_POOL_SIZE 6000
#define PACKET_PAYLOAD  1500

/*
 *   Static local data.
 */

static NX_IP      gNetxIpInstance;
static NX_PACKET_POOL gNetxPool;
static TX_THREAD  MainThread;
static TX_THREAD  AllegroThread;

/*
 *   Routine prototypes.
 */
extern void      AllegroThreadEntry(ULONG thread_input);
extern void      MainThreadEntry(ULONG thread_input);

/* This is a generic network driver which should eventually be replaced with
the actual Ethernet network driver for the host. It is used here (and in
the nx_ip_create() function call to facilitate initial application build
and execution. Packets never get to the hardware layer but are processed
at the TCP/IP layer only.
 */
void _nx_ram_network_driver(struct NX_IP_DRIVER_STRUCT *driver_req);

/*
 *   Define the main entry point.
 */

int main() {
    /*
     *   Enter the ThreadX kernel. This routine does not return,
     *   but it will call tx_application_define.
     */
    tx_kernel_enter();
}

```

```

/*
 This application thread is called from ThreadX.
*/

void tx_application_define(void *first_unused_memory) {
    CHAR *   theMemoryPointer;
    UINT     theStatus;

    /*
     Setup the working pointer.
    */
    theMemoryPointer = (CHAR *) first_unused_memory;

    /*
     Initialize the NetX system.
    */
    nx_system_initialize();

    /*
     Create a packet pool.
    */
    theStatus = nx_packet_pool_create(&gNetxPool,
                                     "NetX Main Packet Pool",
                                     PACKET_PAYLOAD, /* payload size */
                                     theMemoryPointer,
                                     PACKET_POOL_SIZE); /* memory size */
    theMemoryPointer += PACKET_POOL_SIZE;

#ifdef ASTASK_DEBUG
    if (theStatus != NX_SUCCESS) {
        printf("tx_application,
              nx_packet_pool_create failed, theStatus = %d\n" theStatus);
    }
#endif

    /*
     Create an IP instance.
    */
    if (theStatus == NX_SUCCESS) {
        gNetxIpInstancePtr = &gNetxIpInstance;
        theStatus = nx_ip_create(&gNetxIpInstance,
                                "NetX IP Instance",
                                DEVICE_IP_ADDRESS,
                                DEVICE_NETWORK_MASK,
                                &gNetxPool,
                                _nx_ram_network_driver,

/*
 The driver above is a sample Ethernet driver. Substitute the generic
 driver with the driver for your hardware such as this one:

                                _nx_etherDriver_mcf5485,
*/
                                theMemoryPointer,
                                2048,
                                1);
        theMemoryPointer += 2048;
    }
}

```

```

#if ASTASK_DEBUG
    if (theStatus != NX_SUCCESS) {
        printf("tx_application, nx_ip_create failed, theStatus = %d\n",
            theStatus);
    }
#endif

/*
 * Enable ARP and supply ARP cache memory.
 */
if (theStatus == NX_SUCCESS) {
    theStatus = nx_arp_enable(&NetxIpInstance,
        (void *) theMemoryPointer,
        1024);
    theMemoryPointer += 1024;
}

#if ASTASK_DEBUG
    if (theStatus != NX_SUCCESS) {
        printf("tx_application, nx_arp_enable failed, theStatus = %d\n",
            theStatus);
    }
#endif

/*
 * Enable TCP for this IP instance.
 */
theStatus = nx_tcp_enable(&NetxIpInstance);

#if ASTASK_DEBUG
    if (theStatus != NX_SUCCESS && theStatus != NX_ALREADY_ENABLED) {
        printf("tx_application, nx_tcp_enable failed, theStatus = %d\n",
            theStatus);
    }
#endif

/*
 * Create the Main thread.
 */
if (theStatus == TX_SUCCESS) {
    theStatus = tx_thread_create(&MainThread,
        "Main Thread",
        MainThreadEntry,
        0,
        theMemoryPointer,
        MAIN_STACK_SIZE,
        1,
        1,
        TX_NO_TIME_SLICE,
        TX_AUTO_START);
    theMemoryPointer += MAIN_STACK_SIZE;
}

```

```

#if ASTASK_DEBUG
    if (theStatus != TX_SUCCESS) {
        printf("tx_application,
            Main Thread create failed, theStatus = %d\n", theStatus);
    }
#endif

    /*
    Create the thread that runs the Allegro products.
    */
    if (theStatus == TX_SUCCESS) {
        tx_thread_create(&AllegroThread,
            "Allegro Thread",
            AllegroThreadEntry,
            0,
            theMemoryPointer,
            ALLEGRO_STACK_SIZE,
            1,
            1,
            TX_NO_TIME_SLICE,
            TX_AUTO_START);
        theMemoryPointer += ALLEGRO_STACK_SIZE;
    }
#if ASTASK_DEBUG
    if (theStatus != TX_SUCCESS) {
        printf("tx_application,
            Allegro Thread create failed, theStatus = %d\n", theStatus);
    }
#endif

    return;
}

/*
This is the User Application's main thread.
*/

void MainThreadEntry(ULONG thread_input) {
    int    theOkayFlag;

    theOkayFlag = 1;

    while (theOkayFlag) {
        /*
        User Application code should be inserted here.
        */

        /*
        Give time to other threads.
        */
        tx_thread_relinquish();
    }

    return;
}

```

```

/*
 * This thread initializes and runs the Allegro products.
 */

void AllegroThreadEntry(ULONG thread_input) {
    void * theAllegroDataPtr;
    int theHttpTasks;
    int theOkayFlag;
    int theTcpTasks;

    /*
     * Initialize the Allegro Engine.
     */
    theAllegroDataPtr = AllegroTaskInit();

    if (theAllegroDataPtr != (void *) 0) {
        theOkayFlag = 1;
    }
    else {
        theOkayFlag = 0;
    }

#ifdef ASTASK_DEBUG
    printf("AllegroThread, AllegroTaskInit failed!\n");
#endif

    while (theOkayFlag) {
        /*
         * Run the Allegro products.
         */
        theOkayFlag = AllegroMainTask(theAllegroDataPtr,
                                       &theHttpTasks,
                                       &theTcpTasks);

        if (theOkayFlag) {
            /*
             * Give time to other threads. To call this thread
             * 10 times/second as the documentation recommends,
             * sleep for the specified number of seconds.
             */
            tx_thread_sleep(10);
        }
        else {
            AllegroTaskDeInit(theAllegroDataPtr);
        }

#ifdef ASTASK_DEBUG
        printf("AllegroThread, AllegroMainTask failed!\n");
#endif
    }

    return;
}

```