Remote Transaction Call (RTC) Administrator and Programmer’s Guide

Abstract
This publication describes the Emperex Corporation’s Remote Transaction Call (RTC) product.

<table>
<thead>
<tr>
<th>Part Number</th>
<th>080240 D00</th>
</tr>
</thead>
<tbody>
<tr>
<td>Product Version</td>
<td>D20</td>
</tr>
<tr>
<td>Operating System</td>
<td>Guardian Gxx, Jxx</td>
</tr>
<tr>
<td>Printing Date</td>
<td>April 2016</td>
</tr>
</tbody>
</table>
Document History

<table>
<thead>
<tr>
<th>Edition</th>
<th>Part Number</th>
<th>Operating System</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>First</td>
<td>080240 D00</td>
<td>Guardian D3x-D4x, Gxx</td>
<td>January 1999</td>
</tr>
<tr>
<td>Second (API added)</td>
<td>080240 D00</td>
<td>Guardian D3x-D4x, Gxx</td>
<td>January 1999</td>
</tr>
<tr>
<td>Third</td>
<td>080240 D00</td>
<td>Guardian D3x-D4x, Gxx</td>
<td>February 1999</td>
</tr>
<tr>
<td>Fourth</td>
<td>080240 D00</td>
<td>Guardian D3x-D4, Gxx</td>
<td>December 1999</td>
</tr>
<tr>
<td>Fifth</td>
<td>080240 D00</td>
<td>Guardian D3x-D4, Gxx</td>
<td>April 2006</td>
</tr>
<tr>
<td>Sixth</td>
<td>080240 D20</td>
<td>Guardian Gxx, Jxx</td>
<td>April 2016</td>
</tr>
</tbody>
</table>

Copyright Notice

Copyright © 1999-2016 by Emperex Corporation. Printed in Canada.

All rights reserved. No part of this document may be reproduced in any form, including photocopying, optical character scanning recognition, facsimile (FAX) transmission, or translation to another language without the prior written consent of Emperex Corporation.

The information contained in this document is furnished under a license agreement or nondisclosure agreement. The software and associated documentation may be used only in accordance with the terms of the agreement.

Trademarks or Service Marks

The following are trademarks or service marks of Emperex Corporation:

PrimeCode ESD, RMS, RTC.

The following are trademarks or service marks of Tandem Computers Incorporated:

EMS, ENSCRIBE, EXPAND, Guardian, NonStop™, NSK, RSC, SAFEGUARD, TACL, Tandem, TMF, Visual Inspect.

EZ-RPC and NNETD are trademarks of NobleNet Inc.

ONC RPC and Solaris are trademarks of Sun Microsystems.
# Contents

1  **PREFACE** ........................................................................................................................................... 1-1  
    PURPOSE AND SCOPE .......................................................................................................................... 1-1  
    ORGANIZATION ..................................................................................................................................... 1-1  
    WHO SHOULD READ WHAT .................................................................................................................. 1-1  
    RELATED MANUALS AND PRODUCTS ................................................................................................. 1-2  

2  **NOTATION CONVENTIONS** ............................................................................................................... 2-1  

3  **OVERVIEW OF REMOTE TRANSACTION CALL** ............................................................................ 3-1  
    IMPORTANCE OF THE HOSTS FILE ...................................................................................................... 3-3  
    SERVER-SIDE COMPONENTS ............................................................................................................... 3-4  
    CLIENT-SIDE COMPONENTS ............................................................................................................... 3-5  

4  **INSTALLING RTC** ............................................................................................................................. 4-1  
    FEATURES OF THE NEW SETUP MACRO ............................................................................................. 4-1  
    POPULATING THE RELEASE SUBVOLUME ............................................................................................ 4-2  
    RUNNING THE SETUP MACRO ............................................................................................................. 4-2  
    SAMPLE SETUP MACRO SESSION ....................................................................................................... 4-3  
    SERVER-SIDE RTC FILES ..................................................................................................................... 4-9  
    PARAMETERS FOR RECOVERY ............................................................................................................. 4-10  
    LICENSING ........................................................................................................................................... 4-11  
    SETTING UP THE RTC CLIENT .............................................................................................................. 4-12  

5  **THE PORTMAPPER PROGRAM** ....................................................................................................... 5-1  
    RUNNING THE PORTMAPPER PROCESS FOR D30 ............................................................................... 5-1  
    RUNNING THE PORTMAPPER PROCESS FOR D40 AND ABOVE ............................................................. 5-1  
    RUNNING THE PORTMAPPER ................................................................................................................. 5-1  
    ERROR MESSAGES .................................................................................................................................. 5-4  
    PORTMAPPER PROGRAM PROTOCOL ............................................................................................... 5-4  
    CHECKING THE PORTMAPPER PROCESS ............................................................................................ 5-5  

6  **THE NNETD PROGRAM** ................................................................................................................... 6-1  
    RUNNING THE NNETD PROGRAM ......................................................................................................... 6-1  
    CONFIGURATION ................................................................................................................................. 6-2  
    SECURITY ............................................................................................................................................... 6-3  

7  **THE RTC API** .................................................................................................................................... 7-1  
    CLIENT APPLICATION INTERFACE ....................................................................................................... 7-1  
    SERVER APPLICATION INTERFACE ..................................................................................................... 7-2  
    COMPILING WITH THE RTC API ........................................................................................................... 7-2  
    LINKING WITH THE RTC API .................................................................................................................. 7-2  
    RUNNING AN RTC CLIENT APPLICATION ............................................................................................ 7-3  
    RTC API DATA TYPES ............................................................................................................................. 7-4  

Emperex Corporation  Table of Contents  1-1
## Table of Contents

**RTC API Functions** ................................................................................................................. 7-5
- rtcAbortTransaction ................................................................................................................. 7-6
- rtcBeginTransaction() .............................................................................................................. 7-7
- rtcEnd ........................................................................................................................................... 7-8
- rtcEndTransaction() .................................................................................................................. 7-9
- rtcInfo() ...................................................................................................................................... 7-10
- rtcPathSend() ............................................................................................................................ 7-12
- rtcStart() ...................................................................................................................................... 7-14
- rtcTranslate() .............................................................................................................................. 7-16
- rtcWrite() ..................................................................................................................................... 7-17
- rtcWriteRead() ............................................................................................................................ 7-18

### 8 RTC Test Script
- Starting a Process-To-Process Session ...................................................................................... 8-1
- Allocating a Buffer ...................................................................................................................... 8-3
- Analyzing Errors ....................................................................................................................... 8-4
- Beginning a TMF Transaction ................................................................................................... 8-5
- Writing to a Process ................................................................................................................... 8-6
- WriteRead to a Process .............................................................................................................. 8-7
- Ending a Transaction ................................................................................................................. 8-7
- Interpreting Error Codes .......................................................................................................... 8-8
- Starting a PATHWAY Session ................................................................................................... 8-8
- PATHSEND to a Server-Class .................................................................................................... 8-9
- Ending a Session ....................................................................................................................... 8-9

**Appendix A - Error Messages** ................................................................................................. A-1
- Symbolic Names and Values ....................................................................................................... A-1
- Symbolic Names, Messages, and Explanations ......................................................................... A-2

**Appendix B - Sample API Client Program** .............................................................................. B-1
- A Simple RTC Example .............................................................................................................. B-1

**Appendix C - Remote Procedure Call (RPC)** .......................................................................... C-1
- Brief Overview ........................................................................................................................... C-1

**Appendix D - Glossary** ............................................................................................................. D-1
1 Preface


Emperex Corporation is proud of its products and technical support reputation. If you have any questions regarding the RTC software or support materials, please contact your Emperex Corporation representative.

Purpose and Scope

- This manual includes instructions for installing RTC on the NonStop to support the communication layer for the PrimeCode GUI.
- It will also be of use to experienced users of the Guardian operating system who want to become familiar with and use the RTC programs and its Application Programmatic Interface (API).

Organization

The information here is organized to help you understand both the basic concepts of RTC and how to use it.

- **Section 3** provides a brief overview of RTC.
- **Section 4** describes how to install RTC.
- **Section 5** describes how to install and start the portmapper program and explains some of the error messages.
- **Section 6** describes the NNETD program.
- **Section 7** describes the RTC API (Application Programmatic Interface).
- **Section 8** describes RTCTEST.

**Appendix A** describes some of the RTC error messages.
**Appendix B** describes an easy RTC test you can run.
**Appendix C** describes some RPC terminology and concepts.
**Appendix D** is a Glossary of RPC and RTC terms.

Who Should Read What

This guide is written for experienced Tandem users and system administrators who will actually be using the RTC programs and/or the API.

**RMS/PrimeCode users** who need only to install the RTC servers on the NonStop to support a communications layer for PrimeCode can limit their focus to **Section 4 – Installing RTC**.
Related Manuals and Products

These are supplemental publications:

- Guardian Programmer’s Guide
- Guardian System Operations Guide
- PATHWAY PATHCOM Reference Manual
- PATHWAY System Management Guide
- SUN ONC Programming Manual
- TACL Programming Guide
- TACL Reference Manual
- Tandem TCP/IP Management and Operations Manual
- Tandem NonStop™ TM/MP Reference Manual
- Tandem NonStop™ TS/MP Pathsend and Server Programming Manual
2 Notation Conventions

The following list summarizes the conventions for syntax notation in this manual:

<table>
<thead>
<tr>
<th>Syntax Conventions</th>
<th>Notation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>UPPERCASE LETTERS</td>
<td></td>
<td>Uppercase letters represent keywords and reserved words; you must enter these items exactly as shown.</td>
</tr>
<tr>
<td>lowercase letters</td>
<td></td>
<td>Lowercase italic letters represent variable items that you must supply.</td>
</tr>
<tr>
<td>Brackets [ ]</td>
<td></td>
<td>Brackets enclose optional syntax items. A vertically-aligned group of items enclosed in brackets represents a list of selections from which you may choose one or none.</td>
</tr>
<tr>
<td>Braces { }</td>
<td></td>
<td>Braces enclose required syntax items. A vertically-aligned group of items enclosed in braces represents a list of selections from which you must choose only one.</td>
</tr>
<tr>
<td>Vertical Line(())</td>
<td></td>
<td>A vertical line separating items in a list enclosed in either braces or brackets is an alternative to vertical alignment of the options (usually in short lists that can be easily shown on one line).</td>
</tr>
<tr>
<td>Ellipsis ...</td>
<td></td>
<td>An ellipsis immediately following a pair of brackets or braces indicates that you can repeat the enclosed syntax item any number of times.</td>
</tr>
<tr>
<td>Spaces</td>
<td></td>
<td>If two items are separated by a space, that space is required between the items. If one of the items is a punctuation symbol such as a parenthesis or a comma, spaces are optional.</td>
</tr>
<tr>
<td>Punctuation</td>
<td></td>
<td>Parenthesis, commas, semicolons, and other symbols or punctuation not described above must be entered precisely as shown. If any of the punctuation above appears enclosed in quotation marks, that character is a required character and you must enter it as shown.</td>
</tr>
<tr>
<td>&amp;</td>
<td></td>
<td>Continue to type the command as one line.</td>
</tr>
</tbody>
</table>

**RTC-XXXX**

This is the change number, found in the left margin. It indicates that a change or addition has been made to both the software and the manual since the last edition.

This line indicates where the changed or new information ends.

**All Returns are implicit.**
3 Overview of Remote Transaction Call

The Remote Transaction Call (RTC) protocol provides a simple communication interface between a client application and one or more Tandem processes. RTC supports process server classes under PATHWAY and TMF transactions. Unlike other similar products, RTC is specifically designed for TCP/IP networks. As such, it exhibits more simplicity and better performance than these other products.

In this diagram, the RTC client can communicate with the NSK process, PATHWAY server class and TMF using the RTC API and the RTC server on the NSK server.

Figure 1-1

RTC Overview
**Figure 1-2**

**How the RTC Server Allows the Client to Communicate with NSK Process, PATHWAY Server and TMF**

1. **PORTMAP (the portmapper) starts and listens on Port 111.**
2. **NNETD starts and registers both RTC’s and its own port number to the portmapper.**
3. The RTC client asks PORTMAP for the RTC port number. PORTMAP returns NNETD’s port number. The RTC client contacts NNETD.
4. **NNETD launches the RTC server and passes the client’s incoming connection.**
5. The RTC client can now issue WRITE, WRITEReads and PATHSENDs to processes and PATHWAY server classes on the NSK. It can also begin, end and abort TMF transactions.
6. **RTCTEST and RPCINFO can be used to diagnose RTCSERV and the portmapper respectively.**
Importance of the Hosts File

RTC uses TCP/IP hostnames to determine the IP address of the machine to which it will connect. This mapping of name to IP address is kept in the *hosts* file.

- On UNIX, the file is `/etc/hosts`.

- On Guardian, the file is `$SYSTEM.SYSTEM.HOSTS` (and `$SYSTEM.ZTCP/IP.HOSTS` may mirror it).

- On Windows, the file is usually `c:\windows\hosts` (where `C:\windows` is the folder containing the OS itself).

Whatever the platform, RTC will only be able to connect to the machines which are named in the hosts file of that machine. This file needs be correct for RTC to work properly. The file must be sufficiently complete to define all machines RTC will connect to from the current machine.
Server-Side Components

PORTMAPPER

The portmapper registers the port number used by servers using Sun’s Remote Procedure Call (RPC) protocol. A remote RPC client contacts the portmapper to determine the port number used by an RPC server. The portmapper is a stand-alone process that can be shared by any Emperex Corporation or Tandem product offering RPC-based services, such as NFS, Visual Inspect and RTC.

RPCINFO

RPCINFO is a program that shows the program and version numbers, the protocol transport supported (TCP or UDP), the IP port numbers and the name of the Remote Procedure Call (RPC) services available on the server. RPCINFO is a very useful network diagnostic tool because it can “ping” an RPC server to determine whether a service is available. For more information on RPC terminology, see Appendix C.

NNETD

NNETD is an EZ-RPC version of INETD. It is a process that provides listening and launching services for the RTC server. When NNETD is started, it opens a root configuration file called NNETDLST that points to the RTC service configuration file RTCCFG. Then, NNETD creates a TCP listening socket in the desired TCP/IP stack and binds it to a random port. Next it registers the RTC program number and port number with the portmapper and starts accepting incoming connections. If requested, NNETD launches a backup process.

When an incoming connection for RTC is received, NNETD launches an RTC server passing the connection information and the incoming socket. Once the RTC server is started, NNETD continues to listen for other incoming connections.

RTCSERV

RTCSERV is the RTC server process providing interface services to Tandem processes and transaction services for the remote client. One such process is started by NNETD for each incoming connection. The RTC server stops when the client terminates the connection.

RTCTEST

RTCTEST emulates an RTC client interactively.
All RTC API calls can be issued manually, making RTCTEST a very powerful diagnostic and modelling tool.

SETUP

SETUP is an interactive TACL macro that installs RTC and dependent products such as RPC. It generates macros to start up, shut down and monitor RTC called RTCCOLD, RTCWARM, RTCSTOP and RTCCHECK.
Client-Side Components

Application Programmatic Interface

The RTC API provides a simple interface for communicating between a client application and one or more Tandem processes. The communication can be one or two-way, but is always initiated by the client-side application. RTC is implemented as a client/server application on top of the RPC (Remote Procedure Call) protocol. The RTC API is implemented as a set of ANSI-C and Visual Basic headers and library files which are platform specific. See Setting Up the RTC Client for a list of files specific to your platform.

RPCINFO

RPCINFO is a process similar to the one described in Server-Side Components, running on all supported client platforms.

RTCTEST

RTCTEST is a process similar to the one described in Server-Side Components, running on all supported client platforms.
4 Installing RTC

The release subvolume will contain all the files necessary to install both the RTC on the server, and on the client.

PrimeCode users
If your aim is to install the RTC server and other required processes on the NonStop to support PrimeCode, simply run a SETUP macro as shown on page 4-3 in Running the SETUP Macro. (No client side installation is required apart from installing PrimeCode.)

For programmers using RTC and the API for another application
In addition to running setup for the NonStop, you will also FTP the appropriate self-extracting file to the client and install it.

This process is explained in more detail in Setting Up The RTC Client.

Features of the new SETUP Macro

The new SETUP procedure (introduced April 2016) includes the following enhancements.

- Capability to assign a chosen port number for each of the RTC and Daemon processes and the same ports will be used in subsequent start-ups of RTC.

  A random pair of ports can be accepted at the time RTC is installed but these port values are not stored in the configuration files. Any restart of RTC will cause new random ports to be selected. However, configuration files may be modified to ensure the port numbers that are used.

- Capability to accept the input of a known TCP/IP stack or present available options for your selection during installation of RTC.

- Run a “typical” default installation plan using the Volume of your choice and predetermined subvolume names for the installed files (RTCCONF, ZRPC, EZRPC, and ZRTC)

  OR

  Run a “non-typical” customized installation for the user to specify specific subvolumes as desired.

- Capability to install RTC for the first time on a NonStop system, reinstall, or upgrade over an existing RTC installation without having to manually stop it first.
Populating the Release Subvolume

Before you can run the SETUP macro, you need to decide on a subvolume name to contain the RTC files. ($DATA.RTCD40RL is used here in the examples).

1. Download the PAK file ‘comm’ from the Emperex Web Site.

2. FTP the PAK file in binary to the Release Subvolume (i.e. $<VOL>.RTCD40RL)
   This PAK file should expand without changing its file code from 0 to 1729.
   UNPAK the file:

   ```bash
   $DATA.RTCD40RL 1> UNPAK COMM *..*, VOL $DATA.RTCD40RL, MYID, LISTALL
   ```

   Among the 14 files extracted from the PAK, the following are essential to the NonStop installation:

   - **NNETD**: NNETD executable.
   - **NNETDLST**: NNETD server configuration file.
   - **RPC**: Portmapper service name configuration file.
   - **RPCINFO**: RPCINFO executable.
   - **RTCCFG**: RTC server configuration file.
   - **RTCECHO**: RTCECHO executable.
   - **RTCSERV**: RTCSERV executable.
   - **RTCTEST**: RTCTEST executable.
   - **SETUP**: SETUP macro.

   If you ftp individual files to the NonStop, you will need to alter the file codes. All executable files must be file code 100. All other files are code 101.

Running the SETUP Macro

To run the SETUP macro you need to log on as SUPER.SUPER.

Following is a sample interaction with this macro which assumes that RTC must be configured for $2TC0 TCP/IP stack specifically.

Accepting all defaults, subvolumes on $SYSTEM will be created to contain the RPC, EZ-RPC and RTC product files.

The prompts you will be asked to respond to when you run the SETUP macro found in the release subvolume are illustrated in the Sample that follows.
Sample SETUP Macro Session

**Note!** When running SETUP, default responses are indicated by square brackets []. If you hit the Return key without typing an entry for a parameter, the default is selected.

```bash
$SYSTEM RTCD40RL 1> RUN SETUP

SETUP - RTC Installation and Setup Macro - D9024 D40
Emperex Corporation copyright © 2016

SETUP: SETUP must be run from the RTC release subvolume:
e.g., $DATA.RTCDXXRL.

SETUP must be run only by SUPER.SUPER just after the
initial product installation.

Should we continue? ([y]/n): Y

Typical installation (recommended) uses default settings unless specified.

Typical installation? ([Y]/[N]):

**Yes, (typical)...**

Enter an existing Volume to install : $SYSTEM
All previously installed RTC components will be replaced. Proceed? ([Y]/[N]):

**No, (not typical)...**

Enter a configuration subvolume ($SYSTEM.RTCCONF):

> SETUP: Checking the subvolume: $SYSTEM.RTCCONF . . . .
> SETUP: Detected existing files in $SYSTEM.RTCCONF.
> Are you sure you want to rebuild the RTCCOLD, RTCWARM,
> RTCCHECK and RTXSTOP macros? If you go ahead, existing
> configuration in TRCPWY, RTCCFG and NNETDLST will be lost.
Rebuild configuration? ([y]/n):

> SETUP: This MACRO generates files to configure and manage RTC
> processes. Therefore, to avoid any potential FE_INUSE error,
> the RTC processes, which are currently running, must be stopped.
```
If RTC is currently running – you’ll be asked for permission to stop it.

```
Should we stop the RTC subsystem? ([y]/n):

RTCSTOP>>> Stopping the NNETD daemon(s). . .
Process             Pri PFR %WT Userid  Program file           Hometerm
$NTD0       0,360 160  005 255,255 $SYSTEM.EZRPC.NNETD $ZHOME
RTCSTOP>>> Un-registering RTC services in $ZPM0. . .
RTC server processes:
TCP/IP Stack $ZTC0:
-------NNETD Daemon $NTD0 is DOWN!
+++++++Portmapper $ZPM0 is UP.
+++++++TCP/IP $ZTC0 is UP.
  Program vers proto   port
  100000    2   upd    111
  100000    2   tcp    111
```

Provide a Console for RTC...

```
SETUP: A physical terminal, $NULL or a static window in TELSERV must be used as a console to run RTC. Using a dynamic window (e.g., $ZTNT.#PTYnnnn) will cause problems when re-starting the RTC subsystem.

Enter the name of physical terminal ($NULL): $ZHOME
```

In a non-typical installation you will be prompted for subvolume names...

```
SETUP: The portmapper and RPC files should be installed in their own subvolume. The subvolume in parentheses below will be assumed if no input is provided.
Enter the RPC installation subvolume ($SYSTEM.ZRPC):
SETUP: The NNETD daemon and files should be installed in their own EZ-RPC subvolume. The subvolume in parentheses below will be assumed if no input is provided.
Enter the EZ-RPC installation subvolume ($SYSTEM.EZRPC):
SETUP: RTC programs should be installed in their own subvolume. The subvolume in parentheses below will be assumed if no input is provided.
Enter the RTC installation subvolume ($SYSTEM.ZRTC):
```

Specify the $Volume.Subvolume where your license is; not its name...
(it is expected the license is named LICENSE)

```
SETUP: The LICENSE file is shared by many Emperex processes and it should be installed in its own subvolume. The subvolume in parentheses below will be assumed if no input is provided.
Enter the LICENSE subvolume ($SYSTEM.LICENSE): $SYSTEM.RTCCONF
```
SETUP: Checking if Portmappers are running...
SETUP: Looking for the HOSTS file...
SETUP: Detected TACL define "TCPIP^HOST^FILE \NERO\$.SYSTEM.ZTCPIP.HOSTS"
SETUP: Discovering your TCP/IP configuration...

You can specify the TCP/IP stack or select it...

If you don’t enter a TCP/IP stack...
(you will need to select from the possible options that are displayed until you say Yes to accept one. After a selection has been made, other TCP/IP stacks that may be available are not displayed.)

>SETUP: Nothing was entered.
SCF W20052 Creating file \NERO\$.SYSTEM.RTCD40RL.SCFLDEV
>SETUP: Detected the following IP addresses associated with this stack
TCPIP Info SUBNET \NERO\$.ZTC3.*
Name Devicename *IPADDRESS TYPE *SUBNETMASK SuName QIO *R
#LOOP0 \NOSYS\$.NOIOP0.0.0.0 LOOP-BACK %H00000000 OFF N
Do you want to run RTC on $ZTC3? ([y]/n): n
>SETUP: Detected the following IP addresses associated with this stack
TCPIP Info SUBNET \NERO\$.ZTC2.*
Name Devicename *IPADDRESS TYPE *SUBNETMASK SuName QIO *R
#LOOP0 \NOSYS\$.NOIOP0.0.0.0 LOOP-BACK %H0000000 OFF N
Do you want to run RTC on $ZTC2? ([y]/n): n
>SETUP: Detected the following IP addresses associated with this stack
TCPIP Info SUBNET \NERO\$.ZTC1.*
Name Devicename *IPADDRESS TYPE *SUBNETMASK SuName QIO *R
#LOOP0 \NOSYS\$.NOIOP127.0.0.1 LOOP-BACK %HFFFF0000 OFF N
#SN1 \NERO\$.L01B 172.23.10.16 ETHERNET %HFFFF0000 ON N
Do you want to run RTC on $ZTC1? ([y]/n): y

>SETUP: Please note the IP address listed above. You will need it to configure the GUI client.
If you do enter the TCP/IP stack...

> TCP/IP process entered: $ZTC0

> SETUP: Detected the following IP addresses associated with this stack

TCPIP Info SUBNET \Nero.$ZTc1.*

<table>
<thead>
<tr>
<th>Name</th>
<th>Devicename</th>
<th>*IPAddress TYPE</th>
<th>*Subnetmask</th>
<th>SuName</th>
<th>QIO</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>#loop0</td>
<td>\Nosys.$Noiop127.0.0.1 loop-back</td>
<td>$ff000000</td>
<td>OFF</td>
<td>N</td>
<td></td>
<td></td>
</tr>
<tr>
<td>#Sni</td>
<td>\Nero.L01B</td>
<td>172.23.10.16</td>
<td>Ethernet</td>
<td>$ffff0000</td>
<td>ON</td>
<td>N</td>
</tr>
</tbody>
</table>

> SETUP: Please note the IP address listed above. You will need it to configure the GUI client.

Installation files are copied...

(If you have not chosen the "typical" installation you will be advised before any existing files are purged.)

> SETUP: Copying files to the specified installation subvolumes . . .

> SETUP: File in use. $SYSTEM.ZRPC.PORTMAP is not replaced.

Files duplicated: 1

FILES DUPLICATED:1
FILES DUPLICATED:1
FILES DUPLICATED:1
FILES DUPLICATED:1
FILES DUPLICATED:1
FILES DUPLICATED:1
FILES DUPLICATED:1
FILES DUPLICATED:1
FILES DUPLICATED:1
FILES DUPLICATED:1
FILES DUPLICATED:1

> SETUP: Generating RTC configuration files . . .

Enter a port number for nnetd, default is any port: 2016

$SYSTEM.RTCCONF.RTCVARS
$SYSTEM.RTCCONF.RTCSTOP
$SYSTEM.RTCCONF.RTCCHECK
$SYSTEM.RTCCONF.RTCOLD
$SYSTEM.RTCCONF.RTCWARM
$SYSTEM.RTCCONF.RTCFWY
$SYSTEM.EZRPC.NNETDLST

Enter a port number for RTC, default is any port: 2017
RTC Installation Notes:

$SYSTEM.RTCConf.RTCOLD coldstarts RTC. Typically, this file should be run once after the initial install. It must be run by SUPER.SUPER. WARNING: A COLDSTART PURGES PREVIOUS CONFIGURATION INFORMATION!

$SYSTEM.RTCConf.RTCSTOP shuts down RTC processes, if possible in an orderly manner.

$SYSTEM.RTCConf.RTCWARM only re-starts RTC processes that died.

$SYSTEM.RTCConf.RTCCHECK reports the state of all processes associated with RTC, including the test processes and the TCP/IP-related processes.

$SYSTEM.RTCConf.RTCVARS contains all subsystem variables. If you need to change parameters such as CPU assignments or process names, you should edit this file.

$SYSTEM.RTCConf.RTCPWY is the configuration file for the test PATHWAY.

$SYSTEM.EZRPC.NNETDLST is the master configuration file for the NNETD daemon.

$SYSTEM.EZRPC.RTCFG is the configuration file for the RTC service to be launched by NNETD.

To keep track of which RTC processes are associated with which TCP/IP stacks, the following naming convention is recommended:

- Portmapper: $ZPMn
- NNETD Daemon: $NTDn
Where “n” is a unique TCP/IP stack numerical identifier.

SETUP: At this point, you should try to bring up RTC, then shut it down to test the RTCCOLD and RTCSTOP files.

If any errors are encountered, please refer to the EMS events log for diagnostic. Configuration errors can be corrected by re-running this SETUP MACRO, or by directly editing the subsystem variables in $DAT6.RTCConf.RTCVARS.

Once the RTC management files operate correctly, it is suggested to re-start the subsystem under a userid in the SUPER group, but other than SUPER.SUPER.

Do you want to coldstart RTC now? ([y]/n): Y

WARNING: RTCCOLD will erase the following configuration files: $SYSTEM.ZRTC.RTCPWY, $DAT6.EZRPC.RTCFG and $SYSTEM.EZRPC.NNETDLST.

Should we continue? (y/[n]): Y

Do you want to start a test environment? ([y]/n): Y
If you chose to start a test environment,"

RTCCOLD>> Starting the test Pathmon $ZRTCP...
RTCCOLD>> Starting the test process $ECHO...
RTCCOLD>> Starting the test process $TEST...

In this example, the user chose to coldstart RTC...

( RTC is started and the results of the command RTCCHECK are displayed.)

In this example, the user chose to coldstart RTC...

In this example, the user chose to coldstart RTC...

With RTC installed and a start-up tested you can stop it....

(Respond with N and RTC will stay up.)

This is the end of RTC installation steps for PrimeCode users.
Server-Side RTC Files

Upon successful completion, the RPC installation subvolume (default is $SYSTEM.ZRPC) will contain the following files:

**PORTMAP**  The portmapper executable.

**RPC**  The RPC service configuration file.

**RPCINFO**  The RPCINFO executable.

The EZ-RPC installation subvolume (default is $SYSTEM.EZRPC) will contain:

**NNETD**  The NNETD executable.

**NNETDLST**  The NNETD server configuration file, including RTCSERV.

**RPC**  The RPC service configuration file, including NNETD and RTC.

**RTCCFG**  The RTC server configuration file used by NNETD.

The RTC installation subvolume (default is $SYSTEM.ZRTC) will contain:

**RTCECHO**  The RTCECHO executable (used if a test environment was configured by RTCCOLD).

**RTCSERV**  The RTC server executable.

**RTCTEST**  The RTC interactive test executable.

The RTC management subvolume (default is $SYSTEM.RTCCONF) will contain:

**PATHCTL**  PATHWAY control file for the RTC test environment (generated by RTCCOLD).

**RTCCHECK**  Macro to check RTC component status.

**RTCCOLD**  Macro to cold start the portmapper, RTC, and NNETD.

**RTCPWY**  PATHWAY input file for the RTC test environment (generated by RTCCOLD).

**RTCSTOP**  Macro to stop RTC and NNETD.

**RTCVARS**  RTC variables used by RTCCOLD, RTCCHECK, RTCSTOP and RTCWARM.

**RTCWARM**  Macro to warm start RTC and NNETD.

The LICENSE installation subvolume (default is $SYSTEM.LICENSE) will contain:

**LICENSE**  The LICENSE file customized to run RTC and dependent products on your system.
Parameters for Recovery

When you establish a connection from one machine to another, you get a “socket” which connects you to the right place on the other machine. It is a dedicated wire or “pipe” between the two machines. Each gets a socket. If nothing is happening on one side of the socket, the other side will send a probe message to the other to see if it is still alive. If it is not, the messages will keep being sent until there are several of them hanging around. These are called “rogue” servers.

To avoid running out of servers, there is a socket level protocol option called KeepAlive. It is needed with RTC to properly configure the RTC server installation. This option is controlled at the TCP/IP stack level by a series of options/parameters, which the RTC System Administrator needs to specify to properly control recovery behaviour. This is a list of the available parameters and how they affect the TCP/IP stack regarding KeepAlive.

Note!
These are the parameters and commands for Guardian D45. Please consult the current version of the SCF manual for TCP/IP for the exact syntax and parameters.

KeepAlive is controlled by 3 parameters within the TCP/IP stack:

* **Keep Alive Idle (TCPKEEPIDLE)**
  is the amount of time in seconds before TCP will issue a keep alive packet on sockets that have enabled this option. The default is 45 seconds. The range is 1 to 7200.

* **Keep Alive Interval (TCPKEEPINTVL)**
  is the time interval in seconds between retransmissions of unacknowledged keep alive packets. The default is 45 seconds. The range is 1 to 1260.

* **Keep Alive Retry Count (TCPKEEPCNT)**
  is the number of times a keep alive packet will be sent without receiving an acknowledgment, after which the TCP connection will be dropped. The default is 8. The range is 1 to 20.

To check these parameters enter these commands. The object is the TCP/IP process.

```
SCF
INFO PROCESS $ZTC5,DETAIL
```

To alter a parameter enter these commands:

```
SCF
ALTER PROCESS $ZTC5,TCPKEEPCNT 7
```
Licensing

Our products are protected against unauthorized use by checking a product feature against an associated encrypted key in a license file. These keys are tied to your specific system serial number and are obtained from an Emperex Corporation support representative. To report your system serial number, enter this command from your TACL prompt:

1>> TMDS; SWITCHTO MDS; STATUS; EXIT

Note!
This line will not run on a Gxx series system because TMDS is not supported.

To run RTC you need two license keys. The first key is to authorize the use of RPC. The PORTMAP, NNETD and the RTCSERV processes all check for the ‘rpclib’ product feature. The other key is for the ‘rtc’ product feature which is checked only by the RTC server.

The default location for the LICENSE file is the subvolume where the executable making the license check resides. Since many executables in different subvolumes will need to access this file, you must add the TACL define =DLM_LICENSE_FILE before starting the server to specify the location of the license file.

Note!
If you don’t want to enter this define each time you run one of the programs mentioned in this manual, you can simply add this line to your TACLCSTM file.

ADD DEFINE =DLM_LICENSE_FILE,CLASS MAP,FILE $SYSTEM.SYSTEM.LICENSE

If the license file is not found or if the ‘rpclib’ product feature in that file is not found, or has expired, or has an invalid license key, the following event will be generated and the server will abend:

$VOLUME.SUBVOLUME.LICENSE Remote Procedure
Call svc_register failed, error 4013: Permission denied

RTCSERV checks for the ‘rtc’ product feature in that file. If it is not found, or has expired, or has an invalid license key, the following event will be generated and the server will abend:

rtc_start() Currently unlicensed
Setting Up the RTC Client

RTC supports Solaris 2.x, Windows™ 95/98 and NT clients. The procedure to install the RTC client on various platforms is provided below. Although the steps will vary somewhat on different operating systems, the process and the resulting installation is basically the same.

16-Bit PC Operating Systems

To install, you must decide on a directory to contain the RTC files (C:\RTC is used here).

1. Create the folder C:\RTC to contain the RTC files.
2. FTP the file RTC16EXE from the RTC subvolume on the Tandem as a binary file called rtc16.exe to your local RTC folder C:\RTC.
3. Double-click on C:\RTC\rtc16.exe file to run it.
4. In the pop-up window that appears (WinZip Self-Extractor - rt16.exe), specify “C:\RTC” as the ‘Unzip to folder’ text value.
5. Click on the ‘Unzip’ button.
6. Ten files will be unzipped. Click on the ‘OK’ button in the pop-up window that announces this.

When you are done, the following files will be installed in C:\RTC:

- **rtc_main.exe**
  A short test program that quickly exercises the RTC API.
- **rtctest.exe**
  A line-oriented interactive interface to the RTC API.
- **rtc.h, rtcapi.h, rtcdefs.h(*)**
  C/C++ header files for programs using the RTC API.
- **rtc16.dll, rtc16.lib**
  The 16-bit RTC client API.
- **ezrpcr4w.dll, winrpc.dll**
  Extra DLLs needed by RTC.

(*)
These files are required to program the RTC API. They are not required to run an RTC client program such as RTCTEST.
32-Bit PC Operating Systems

To install, you must decide on a directory to contain the RTC files (C:\RTC is used here).

1. Create the folder C:\RTC to contain the RTC files.
2. FTP the file RTC32EXE from the RTC subvolume on the Tandem as a binary file called rtc32.exe to your local RTC folder C:\RTC.
3. Double-click on C:\RTC\rtc32.exe file to run it.
4. In the pop-up window that appears (WinZip Self-Extractor -rtc32.exe), specify “C:\RTC” as the ‘Unzip to folder’ text value.
5. Click on the ‘Unzip’ button.
6. Fourteen files will be unzipped. Click on the ‘OK’ button in the pop-up window that announces this.
8. Open the “C:\RTC” folder. If you are planning to use 16-bit and 32-bit applications, drag the contents of the “DLL32s” folder into the “C:\RTC” folder. Use the “Dll32” folder if no 16-bit applications will use RTC. Do not use both at the same time. We suggest you don’t use 16-bit compatible files found in “DLL32s”.

When you are done, the following files will be installed in C:\RTC:

- rtc_main.exe
  A short test program that quickly exercises the RTC API.
- rtctest.exe
  A line-oriented interactive interface to the RTC API.
- rtc.h, rtcapi.h, rtcdefs.h(*)
  C/C++ header files for programs using the RTC API.
- rtc32.dll, rtc32.lib
  The 32-bit RTC client API.
- rtc32.bas(*)
  A Visual Basic wrapper for using the RTC API.
- exzpcw32.dll, winrpc32.dll
  Extra DLLs needed by RTC.
- msvcrt40.dll
  Optional DLL used by RTC with 16-bit applications (from the “DLL32s” subfolder).

(*)
These files are required to program the RTC API. They are not required to run an RTC client program such as RTCTEST.
SUN SOLARIS 2.X

To install, you must decide on a directory to contain the RTC files (~/rtc is used here).

1. Create the directory ~/rtc to contain the RTC files and make it your current directory:

   ```
   mkdir ~/rtc
   cd ~/rtc
   ```

2. FTP the file RTCTARZ from the RTC subvolume on the Tandem as a binary file called rtc.tar.Z to your current directory.

3. Extract the files from the self-extracting archive:

   ```
   zcat rtc.tar.Z | tar -xvf -
   ```

When you are done, the following files will be installed in ~/rtc:

- **rtc_main**
  A short test program that quickly exercises the RTC API.

- **rtctest**
  A line-oriented interactive interface to the RTC API.

- **rtc.h, rtcapi.h, rtcddefs.h**(*
  C/C++ header files for programs using the RTC API.

- **rtc.a**(*
  The 32-bit RTC client API.

- **libezrpc.a**(*
  The extra archive needed by RTC.

(*)
These files are required to program the RTC API. They are not required to run an RTC client program such as RTCTEST.
5 The Portmapper Program

The portmapper registers services using Sun’s Remote Procedure Call (RPC) protocol. It is a stand-alone component that could be used by any Emperex Corporation or Tandem product offering an RPC-based service. Portmapper communication goes through TCP/IP instead of using a direct Guardian inter-process message.

Running the Portmapper Process for D30

```
PARAM TCPIP^PROCESS^NAME $ZTC5
PARAM TCPIP^HOST^FILE $SYSTEM.ZTCPIP.HOSTS
PARAM ZRPC^RPC^FILE $SYSTEM.ZRPC.RPC
ADD DEFINE =DLM_LICENSE_FILE,CLASS MAP,FILE
$SYSTEM.SYSTEM.LICENSE
RUN $SYSTEM.ZRPC.PORTMAP /NAME $ZPM5, NOWAIT/
```

Running the Portmapper Process for D40 and Above

```
ADD DEFINE =TCPIP^PROCESS^NAME,CLASS MAP,FILE $ZTC5
ADD DEFINE =TCPIP^HOST^FILE,CLASS MAP, &
FILE $SYSTEM.ZTCPIP.HOSTS
ADD DEFINE =ZRPC^RPC^FILE,CLASS MAP,FILE $SYSTEM.ZRPC.RPC
ADD DEFINE =DLM_LICENSE_FILE,CLASS MAP,FILE
$SYSTEM.SYSTEM.LICENSE
RUN $SYSTEM.ZRPC.PORTMAP /NAME $ZPM5, NOWAIT/
```

Running the Portmapper

Run the portmapper using the following TACL run command:

```
RUN $SYSTEM.ZRPC.PORTMAP &
/ NAME $ZPMnn [,run-option-list]... / param [, param ] &
...
```

PORTMAP is the file name of the portmapper.

NAME $ZPMnn specifies that the portmapper is to run as a named process. By convention, the name $ZPMnn is recommended where nn is the TCP/IP stack number. For example, $ZPM12 should be used to run portmapper over $ZTC12.

run-option-list is any other valid TACL or run option (such as CPU, NOWAIT, PRI, SWAP, and so on).
Param is one of the following permissible parameters:

- `[ BACKUPCPU ] cpu`: specifies the number of the CPU to be used as the backup CPU. The BACKUPCPU keyword may be omitted only if it is the first parameter specified. If supplied, it must be an existing CPU, different from the primary CPU. If not supplied, the portmapper cannot run as a NonStop™ process pair.

- `TCPIP $name`: specifies the name of a Tandem TCP/IP process that provides a TCP/IP interface to an Ethernet LAN. If not specified and a parameter exists for `TCPIP^PROCESS^NAME`, (C30 and D30 only) or a DEFINE exists for `=TCPIP^PROCESS^NAME`, that parameter or DEFINE is used; otherwise `$ZTC0` is assumed.

- `COLLECTOR $name`: specifies the name of the EMS collector to be used for EMS events. If not specified, the primary collector `$0` is used.

The commands below will start the portmapper process in NonStop™ mode with CPU 1 as the primary CPU and CPU 0 as the backup.

```plaintext
PARAM ZRPC^RPC^FILE $SYSTEM.ZRPC.RPC
RUN $SYSTEM.ZRPC.PORTMAP &
   / NAME $ZPM5, PRI 148, NOWAIT,&
   CPU 1, OUT $S.#ZRPC.MSG / BACKUPCPU 0, TCPIP $ZTC5
```

### Considerations

- Error messages received on startup are displayed on the OUT file, if possible. If an error occurs while the process is writing to the OUT file, the home terminal is used.
- TAACL ASSIGN commands are ignored.
- TCP/IP PARAMs are not supported beyond the D30 release of Guardian.

### Restarting the portmapper Manually

**Note!**

Anyone or anything that resets (stops and then restarts the portmapper) will cause that program to lose track of all its currently registered RPC protocols. RTC registers itself with the portmapper **only** when it starts, so resetting the portmapper makes the RTC server disappear from the list of known RPC services.

*Whenever the portmapper is reset, all registered protocols need to be restarted, including RTC.*
Since the portmapper uses the socket library provided by Tandem TCP/IP, a PARAM for TCPIP^HOST^FILE (C30 and D30 only) or DEFINE for =TCPIP^HOST^FILE must be specified if the Domain Name Resolver is not being used.

In order for the RPCINFO program to function correctly, a PARAM for ZRPC^RPC^FILE or DEFINE for =ZRPC^RPC^FILE must be specified to configure well-known RPC programs.

PARAM TCPIP^HOST^FILE $SYSTEM.ZTCPIP.HOSTS
DEFINE =TCPIP^HOST^FILE,FILE $SYSTEM.ZTCPIP.HOSTS
PARAM ZRPC^RPC^FILE $SYSTEM.ZRPC.RPC

See the Tandem TCP/IP Management and Operations Manual for more detailed information on Tandem TCP/IP configuration files.

If the PORTMAP file resides in $SYSTEM.ZNFS.PORTMAP (for C30-D30 systems) stop the portmapper and purge the file. Then run the RTC SETUP macro and adjust your NFS startup files to reference the portmapper in $SYSTEM.ZRPC. Finally, change the NFS macro files to reference the portmap process name $ZPMnn.
**Error Messages**

This section lists the portmapper process error messages.

**portmap: process must be named**

The PORTMAP process was run as an unnamed process. Run it again, but with the NAME option specified.

**portmap: backup cpu <<n>> is not valid**

The specified backup CPU does not exist or is the same as the primary CPU.

**portmap: process <<name>> does not exist**

The specified collector or TCP/IP process does not exist.

**portmap: <<tcpip-procname>> is not a valid tcpip process**

The specified process has the wrong process type; it must be a Tandem TCP/IP process (device type 48, subtype 0).

**Portmap: <<collect-name>> is not a valid collector process**

The specified collector is not an existing collector process.

In addition to the above error messages, invoking the portmapper may result in I/O or TCP/IP socket library errors.

**Portmapper Program Protocol**

The portmapper program maps RPC program and version numbers to transport-specific port numbers. This program makes dynamic binding of remote programs possible.

Dynamic binding is desirable because the range of reserved port numbers is very small and the number of potential remote programs is very large. By running only the portmapper on a reserved port, the port numbers of other remote programs can be ascertained by querying the portmapper.

The portmapper also aids in broadcast RPC. A given RPC program usually has different port number bindings on different machines, so there is no way to directly broadcast to all of these programs. The portmapper, however, does have a fixed port number. So, to broadcast to a given program, the client actually sends its message to the portmapper located at the broadcast address. Each portmapper that picks up the broadcast, then calls the local service specified by the client. When the portmapper gets the reply from the local service, it sends the reply back to the client.
Checking the Portmapper Process

1. Run the RPCINFO command.

   rpcinfo -p

2. If the portmapper is not running, issue a TACL RUN command to start it, or run the RTCWARM or RTCCOLD macro in the RTC configuration subvolume (RTCCONF).
# The NNETD Program

NNETD is an EZ-RPC function that provides listening and launch services to the RTC server. Therefore, to activate RTC access on the Tandem server, NNETD must be running and its databases must be configured correctly. We recommend that you keep its database files in the EZ-RPC installation subvolume (EZRPC).

## Running the NNETD Program

```tcl
ADD DEFINE =DLM_LICENSE_FILE,CLASS MAP,FILE $SYSTEM.SYSTEM.LICENSE
RUN $VOLUME.EZRPC.NNETD
     &
     / NAME $NTDnn [, RUN-PARAMETERS, ... ] /&
     [ OPTIONS ] REPOSITORY-SUBVOL
```

Where:

- **NAME $NTDnn**
  - is the name of the NNETD process currently running. In a multi-TCP/IP stack environment, it may be confusing to determine which NNETD process is running over a given TCP/IP stack. To facilitate identification, a process naming convention for NNETD should be adopted. We recommend starting the process name with $NTD and adding the TCP/IP stack number at the end. For example, if NNETD runs over $ZTC1, the resulting name in this example would be $NTD1.

- **RUN-PARAMETERS**
  - one or more TACL parameters to run a program such as NAME, NOWAIT, OUT, etc.

- **OPTIONS**
  - `-backupcpu cpu`
    - indicates the server must run as a NonStop™ process pair. The backup will be launched in the CPU designated in `cpu`.

If `cpu` is -1, a running CPU (other than the primary CPU) will be automatically selected. Because the server uses the C-run time library which automatically opens the stdin, stdout and stderr streams, make sure to specify a permanent terminal or device for the IN, OUT and TERM run-time TACL parameters when specifying a backup CPU. Otherwise, TACL defaults to the name of your hometerm, which is likely to be a dynamic window name. Dynamic window names will cause problems if the backup needs to take over, as they will no longer exist after you terminate your telnet session. The C run-time library will try to open a non-existent window, forcing your backup to abend.
**-tcpip $name**

indicates the server must interface with the TCP/IP process $name. Therefore, this option overrides the TACL define or the TCPIP^PROCESS^NAME parameter.

**-collector $name**

allows the server to send events to a process, a file or a collector specified in $name. If this option is not used, the EMS collector $0 will be assumed.

**REPOSITORY-SUBVOL**

is the name of a subvolume containing the NNETD configuration database. We recommend using the EZ-RPC installation subvolume (EZRPC) where NNETD is installed.

---

**Configuration**

NNETD configuration is done using a database of files collected in a single repository subvolume. NNETDLST is the root file. It describes which services need to be launched. Each service has its own configuration file and the name of that file is also contained within NNETDLST. The configuration process is outlined below.

1. Add a configuration entry in the NNETDLST file. For example:

   ```
   # Service Name       Unqualified Configuration File
   # __________________________
   rtc             RTCCFG
   ```

   The entry consists of a keyword you choose to identify your server (for example: ‘rtc’ followed by the name of the configuration file). The configuration must be unqualified since NNETD assumes the file resides within the repository subvolume you specified at startup time.

2. Optionally, you can add or alter parameters and attributes in the configuration file.

   The following example configures the RTC program for 20 concurrent clients (max_servers = 20). The number of concurrent clients can be configured by changing the value of this attribute. In addition, if RTC is to be bound to a specific port, the Port attribute can be changed to the desired port number.
Description = 
Program = RTC : 300691 
Version = RTC_1 : 1 
Version = 
Attributes = 
Command = $DATA.MDRTC03.RTCSERV 
Arguments = 
Comment = 
Port = 0 
Protocol = tcp 
Activation = processperclient 
Max_servers = 20 
Max_servers_delay = 3 
Launch = ondemand 
Owner = 

NT_server_mode = 
Location_Broker = FALSE 

Please be aware that the parser in NNETD is very strict. Attribute names are case-sensitive and must be entered as shown. Also, do not delete any entries, even if they have no value.

Security

If your RTC server needs to be bound to a reserved port, NNETD must be started by a user in the SUPER group (for example: SUPER.OPER). This is because TCP/IP disallows regular user processes to bind to ports below 1024.

Once launched, the RTC server will execute with NNETD’s Process Access ID.
7 The RTC API

The RTC API defines a set of operations which allow a client application to communicate with one or more server applications on one or more Tandem hosts simultaneously.

Note!
See the RTCTEST chapter for a brief hands-on demonstration of the RTC API. Also, see Appendix B for a simple RTC client sample program.

Client Application Interface

Each client-server communications channel is called an “RTC service”. A service is initiated by a call to RtcStart and terminated by a call to RtcEnd. Both PATHWAY and process-to-process services are supported by RTC, but each service will support only one mode (determined at the time of service establishment). Data exchange between the client and server applications is initiated by calls to RtcWrite, RtcWriteRead, and RtcPathSend. All transaction operations are initiated on the client-side by calls to RTC operations RtcAbortTransaction, RtcBeginTransaction and RtcEndTransaction. Each RTC service supports at most one transaction at a time. Multiple transactions can be active if the client starts multiple RTC services.

Three types of errors are returned by the RTC API, depending on where the error was detected. All error codes are returned in RTC_Result by the RTC API. Please see Appendix A for detailed error conditions and recovery.

Local errors are generated if the parameters passed are invalid or if insufficient resource conditions are encountered. These errors are detected on the client system and do not result in an RPC call to the server. These types of errors cause the RTC operation invoked to return error codes other than RTC_SUCCESS, RTC_RPC_ERROR or RTC_SERVER_ERROR. The return code can be converted to a plain-text explanation with a call to RtcTranslate.

Communication errors between the client and the server typically cause the RTC operation to fail with the return code RTC_RPC_ERROR. In this case, RtcInfo can determine the exact code returned by RPC.

Server application errors are reported by the RTC server and cause the RTC operation to fail with the return code RTC_SERVER_ERROR. Here again RtcInfo can determine the file system error returned by the NSK application.
Server Application Interface

There is no API to be programmed on the server-side. The RTC server receives the client requests over the RPC protocol and communicates with the requested NSK processes on behalf of the client application. It also acts as intermediary to TMF (Transaction Monitoring Facility) when transaction operations are requested by the client.

The RTC server generates no errors of its own. It presumes that all RPC-passed values are sufficiently valid to merit passing on to the appropriate Tandem processes (and presumes that those processes will sufficiently validate the actual communication).

Compiling with the RTC API

The RTC API can be invoked from C, C++ and Visual Basic programming languages. The API defined in this document is declared in the rtc.h ANSI-C header file. This file is suitable for use in either ANSI-C or C++ program compilations where the interface functions are referenced. Your system administrator can tell you where the RTC header files suitable for your environment are located. The interface definition file for Visual Basic is rtc32.bas and is only supported for 32-bit Windows™ platforms.

Linking with the RTC API

The actual use of the API requires that the client-side application link with the object library installed on their machine by the system administrator. Your system administrator can tell you where the RTC library files suitable for your environment are located. Please refer to Setting Up the RTC Client for a list of RTC libraries.

For Windows™ applications, the *.dll (Dynamic Link Library) files are dynamically linked into the application. Once linked, the application still needs access to those libraries (usually accommodated by copying both the application and the DLLs to wherever they are wanted).

For UNIX applications, the *.a (archive) files must be statically bound into the application. Once linked, the application no longer needs access to those archives.
Running an RTC Client Application

Once an application has been compiled and linked with the RTC API, nothing further needs to be done on the client side.

Note!

Windows™ applications must use a DLL (Dynamic Linked Library), which requires that those applications continue to have access to the DLL at run-time. Your system administrator will have ensured that this is possible for all machines that were known to use RTC-based applications.

On the server side, the portmapper and NNETD must be started. The RTC server is started dynamically by NNETD when the client starts a service, and stops when the client ends that service. Once the RTC environment is started, it normally runs indefinitely until stopped.

Note!

TACL macros to manage the RTC environment are generated when the product is initially installed. See Running the SETUP Macro and Server-Side RTC Files for details.

Finally, the target NSK application must be started as a named process, otherwise it will be inaccessible over RTC. The process must be ready to accept a file system open message from the RTC server. If the server application is a server class, a PATHWAY environment must be configured and the server class must be started.
**RTC API Data Types**

This section covers the data types used in the RTC API. Only the name of the data type is provided because the actual definition depends on the language used and the development platform.

**RTC_Buffer**

The address of a buffer. Each buffer can be potentially used as both an input and an output buffer.

**RTC_BufSize**

The number of bytes available or useable in a buffer for input or output, depending on the parameter’s context. Buffer sizes vary depending on the type of connection established. The following literals define the maximum buffer sizes for the connection types supported:

- **RTC_MAX_WRBUFSIZE**
  - Maximum buffer size for inter-process (PTP) communication.
- **RTC_MAX_PSBUFSIZE**
  - Maximum buffer size for server class (PATHWAY) communication.

**RTC_Method**

For RtcStart, specifies the desired connection type either to a PATHWAY server class or to a named process. The following enumerated values are available:

- **RTC_PATHWAY**
  - Indicates a service to a PATHWAY server class.
- **RTC_PROCESS**
  - Indicates a service to a named process.

**RTC_Result**

The return code of an RTC function. Any value other than RTC_SUCCESS indicates a problem. Please see Appendix A - *Error Messages* for details.

- **RTC_SUCCESS**
  - Operation was successful.
- **RTC_SUBSYSTEM_NOT_AVAILABLE**
  - Host is not accessible.
- **RTC_PC_HEAP_EXHAUSTED**
  - No memory is available.
- **RTC_TOO_MANY_NOWAIT_REQ**
  - Maximum concurrent requests is exceeded.
- **RTC_UNAVAILABLE**
  - RTC subsystem is not accessible.
- **RTC_SESSION_TERMINATED**
  - Service is disconnected.
- **RTC_UNHOSTED_ADDRESS**
  - Host name is not found.
- **RTC_INCOMPATIBLE_VERSIONS**
  - Client-server versions are incompatible.
- **RTC_INVALID_INPUT_PARAMETERS**
  - Invalid input parameter.
RTC RetCode

The return code returned by the RPC subsystem or by the NSK file system. This value is limited to 16 bits stored in the least significant bits of the value. Non-zero values normally imply that an error occurred.

RTC Service

The handle associated with an RTC service or session. Except for RtcStart which creates the service handle, every RTC operation uses the service handle to access the necessary context for the connection. An RTC client program may have multiple handles and multiple programs may use RTC simultaneously.

RTC String

The name of an RTC object identified by an ASCII string, such as TCP/IP hosts and NSK process names. Except for host names, RTC names must be from 1 to RTC_MAX_NAMELEN characters in length, excluding the terminal NUL. A host name must be from 1 to RTC_MAX_HOSTLEN characters in length, excluding the terminal NUL. RTC String objects are always NUL-terminated.

RTC Timeout

The maximum delay the application is willing to accept, ranging from 0 to RTC_MAX_TIMEOUT. A timeout of "0" implies infinite wait, as do all values greater than RTC_MAX_TIMEOUT. The delay is always measured in seconds; slightly longer delays may occur depending on the client operating system’s scheduling mechanism.

RTC API Functions

This section describes the RTC functions. It is intended for programmers who need to implement an RTC client program in C, C++ or Visual Basic.
RtcAbortTransaction

Cancels an active TMF transaction. When a client calls this function, the TMF subsystem backs out the database changes associated with the current transaction. The RTC server implements this function by calling ABORTTRANSACTION on the NSK. For details please consult the NonStop™ TM/MP Reference Manual.

Syntax for C Programmers

```c
extern RTC_Result RtcAbortTransaction(RTC_Service service, 
                                        RTC_Timeout timeout);
```

Syntax for Visual Basic Programmers

```vbnet
Public Declare Function RtcAbortTransaction Lib "rtc32" Alias "RtcAbortTransaction_VB" (
    ByVal service As Long, _
    ByVal timeout As Integer _
) As Long
```

Parameters

- **service**  INPUT  The RTC service handle with an active transaction to be aborted.
- **timeout**  INPUT  The request timeout, up to RTC_MAX_TIMEOUT.

Errors

- **RTC_SUCCESS**  If message processed successfully.
- **RTC_INVALID_INPUT_PARAMETERS**  If bad IN-type arguments.
- **RTC_INVALID_SERVICE_HANDLE**  If invalid ‘service’.
- **RTC_RPC_ERROR**  If other RPC error.
- **RTC_SERVER_ERROR**  If server application problem.
- **RTC_SESSION_TERMINATED**  If host disconnected.
- **RTC_SUBSYSTEM_NOT_AVAILABLE**  If unable to access.
- **RTC_TIMEOUT**  If call timed out.
- **RTC_TOO_MANY_NOWAIT_REQ**  If pending RTC calls.

Considerations

- A TMF transaction must have been started by calling RtcBeginTransaction for the specified service.
- TMF transactions will be implicitly aborted if the client terminates an established service with an active transaction without calling RtcAbortTransaction.
RtcBeginTransaction()

Starts a new TMF transaction. The RTC server implements this function by calling BEGINTRANSACTION on the NSK. For details, please consult the NonStop™ TM/MP Reference Manual.

Syntax for C Programmers

```c
extern RTC_Result
RtcBeginTransaction(RTC_Service service,
                    RTC_Timeout timeout);
```

Syntax for Visual Basic Programmers

```vbnet
Public Declare Function RtcBeginTransaction
Lib "rtc32" Alias "RtcBeginTransaction_VB"
    (    ByVal service as Long,
        ByVal timeout As Integer ) As Long
```

Parameters

- **service**  **INPUT**  The RTC service handle to start a TMF transaction.
- **timeout**  **INPUT**  The request timeout, up to RTC_MAX_TIMEOUT.

Errors

- RTC_SUCCESS  If message processed successfully.
- RTC_INVALID_INPUT_PARAMETERS  If bad IN-type arguments.
- RTC_INVALID_SERVICE_HANDLE  If invalid ‘service’.
- RTC_RPC_ERROR  If other RPC error.
- RTC_SERVER_ERROR  If server application problem.
- RTC_SESSION_TERMINATED  If host disconnected.
- RTC_SUBSYSTEM_NOT_AVAILABLE  If unable to access.
- RTC_TIMEOUT  If call timed out.
- RTC_TOO_MANY_NOWAIT_REQ  If pending RTC calls.

Considerations

- Only one TMF transaction may be active for a given RTC service at any time.
- TMF transactions must be explicitly started and ended.
- Any active TMF transaction implicitly ended by terminating an RTC service without first invoking either RtcEndTransaction or RtcAbortTransaction is treated as if RtcAbortTransaction were invoked prior to the termination of the RTC service.
RtcEnd

Terminates an RTC service. All resources associated with the service are freed, stopped or closed. This operation always completes without timing out. Once ended, a service is invalid for all future RTC function calls.

Syntax for C Programmers

```c
extern RTC_Result RtcEnd(RTC_Service service);
```

Syntax for Visual Basic Programmers

```vbnet
Public Declare Function RtcEnd
Lib "rtc32" Alias "RtcEnd_VB"
( ByVal service As Long ) As Long
```

Parameters

- **service**  
  **INPUT**  
  The RTC service handle to terminate.

Errors

- **RTC_SUCCESS**  
  If closed connection/session.
- **RTC_INVALID_SERVICE_HANDLE**  
  If invalid ‘service’.
- **RTC_RPC_ERROR**  
  If other RPC error.
- **RTC_TOO_MANY_NOWAIT_REQ**  
  If pending RTC calls.

Considerations

- If the connection is not closed with this operation, the RTC server will automatically close it when the client that originally established the service terminates.
- Any outstanding TMF transaction will be implicitly aborted if the client ends an established service without calling RtcEndTransaction or RtcAbortTransaction.
- Any outstanding file systems opened from the RTC server to the server application are closed.
- This function causes the RTC server who serviced the session to stop.
RtcEndTransaction()

Ends an outstanding TMF transaction. The RTC server implements this function by calling ENDTRANSACTION on the NSK. For details, please consult the NonStop™ TM/MP Reference Manual.

Syntax for C Programmers

```c
extern RTC_Result
RtcEndTransaction(RTC_Service service,
                   RTC_Timeout timeout);
```

Syntax for Visual Basic Programmers

```vba
Public Declare Function RtcEndTransaction
    Lib "rtc32" Alias "RtcEndTransaction_VB"
    ( ByVal service as Long,
      ByVal timeout As Integer
    ) As Long
```

Parameters

- **service** INPUT
  - The RTC service handle with an active transaction to be ended.

- **timeout** INPUT
  - The request timeout, up to RTC_MAX_TIMEOUT.

Errors

- **RTC_SUCCESS**
  - If message processed successfully.

- **RTC_INVALID_INPUT_PARAMETERS**
  - If bad IN-type arguments.

- **RTC_INVALID_SERVICE_HANDLE**
  - If invalid ‘service’.

- **RTC_RPC_ERROR**
  - If other RPC error.

- **RTC_SERVER_ERROR**
  - If server application problem.

- **RTC_SESSION_TERMINATED**
  - If host disconnected.

- **RTC_SUBSYSTEM_NOT_AVAILABLE**
  - If unable to access.

- **RTC_TIMEOUT**
  - If call timed out.

- **RTC_TOO_MANY_NOWAIT_REQ**
  - If pending RTC calls.

Considerations

- TMF transactions must be explicitly started and ended. Outstanding transactions will be implicitly aborted if the client terminates an established service with an active transaction without calling RtcEndTransaction or RtcAbortTransaction.

- A TMF transaction must have been started by calling RtcBeginTransaction for the specified RTC service.
RtcInfo()

Queries the return code and the error message for the last operation attempted by an RTC service.

Syntax for C Programmers

```c
extern RTC_Result
RtcInfo(RTC_Service service,
       RTC_Timeout timeout,
       RTC_BufSize maxout,
       RTC_Buffer obuf,
       RTC_RetCode FAR * retcode);
```

Syntax for Visual Basic Programmers

```vb
Public Declare Function RtcInfo Lib "rtc32" Alias "RtcInfo_VB"
( ByVal service As Long,
  ByVal timeout As Integer,
  ByVal maxout As Integer,
  ByRef obuf As Any,
  ByRef retcode As Integer
) As Long
```

Parameters

- **service** INPUT: The RTC service handle to query.
- **timeout** INPUT: The request timeout on the NSK, up to RTC_MAX_TIMEOUT.
- **maxout** INPUT: The buffer size to be returned, up to RTC_MAX_MSGLEN.
- **obuf** OUTPUT: The error text buffer.
- **retcode** OUTPUT: The return code from the API, the RPC subsystem or the NSK application.

Errors

- **RTC_SUCCESS**: If query successful.
- **RTC_INVALID_INPUT_PARAMETERS**: If bad IN-type or NULL OUT-type arguments.
- **RTC_INVALID_SERVICE_HANDLE**: If invalid ‘service’.
- **RTC_SERVER_ERROR**: If server application problem.
- **RTC_TIMEOUT**: If call timed out.
- **RTC_TOO_MANY_NOWAIT_REQ**: If pending RTC calls.
Considerations

- If the last operation failed with a non-zero result other than RTC_SERVER_ERROR and RTC_RPC_ERROR, the return code is the API error that was originally returned to the service.
- If the last operation failed with RTC_RPC_ERROR, the return code is an RPC subsystem error code. Please consult the *SUN ONC Programming Manual* for details.
- If the last operation failed with RTC_SERVER_ERROR, the return code is an NSK file system error that was returned by the server application to the RTC server. The format of the error message returned is:

  Server error [nnn]

where [nnn] is the returned NSK file system error.

- RTC_SERVER_ERROR and RTC_TIMEOUT are only possible if an RTC_SERVER_ERROR is being examined and there are problems communicating with the RTC server.
RtcPathSend()

Sends and receives data to and from a PATHWAY server class. This function is analogous to a RtcWriteRead to a named process. The RTC server implements this function by calling PATHSEND on the NSK. For details, please consult the NonStop™ TS/MP Pathsend and Server Programming Manual.

Syntax for C Programmers

```c
extern RTC_Result
RtcPathSend(RTC_Service service,
            RTC_Timeout timeout,
            RTC_String server,
            RTC_BufSize ilen,
            RTC_Buffer ibuf,
            RTC_BufSize maxout,
            RTC_Buffer obuf,
            RTC_BufSize FAR * olen);
```

Syntax for Visual Basic Programmers

```vb
Public Declare Function RtcPathSend Lib "rtc32" Alias "RtcPathSend_VB"
    (ByVal service As Long,
     ByVal timeout As Integer,
     ByVal server As String,
     ByVal ilen As Integer,
     ByRef ibuf As Any,
     ByVal maxout As Integer,
     ByRef obuf As Any,
     ByRef olen As Integer)
    As Long
```

Parameters

- service: INPUT
  - The RTC service handle with an RTC_PATHWAY connection.
- timeout: INPUT
  - The request timeout on the NSK, up to RTC_MAX_TIMEOUT.
- server: INPUT
  - The name of the server class to communicate with.
- ilen: INPUT
  - The length of the data to send, up to RTC_MAX_PSBUFSIZE.
- ibuf: INPUT
  - The address of the data buffer to send.
- maxout: INPUT
  - The maximum length of the data to receive, up to RTC_MAX_PSBUFSIZE.
- obuf: OUTPUT
  - The address of the data buffer to receive.
- olen: OUTPUT
  - The actual length of the data received.
Errors

- **RTC_SUCCESS**: If message processed successfully.
- **RTC_INVALID_INPUT_PARAMETERS**: If bad IN-type or NULL OUT-type arguments.
- **RTC_INVALID_SERVICE_HANDLE**: If invalid ‘service’ or ‘service’ started for Process method.
- **RTC_RPC_ERROR**: If other RPC error.
- **RTC_SERVER_ERROR**: If server application problem.
- **RTC_SESSION_TERMINATED**: If host disconnected.
- **RTC_SUBSYSTEM_NOT_AVAILABLE**: If unable to access.
- **RTC_TIMEOUT**: If call timed out.
- **RTC_TOO_MANY_NOWAIT_REQ**: If pending RTC calls.

Considerations

- The server class name supplied must exist and be started under the PATHMON specified in the RtcStart function.
- The same buffer address can be used to send and receive data.
RtcStart()

Establishes a connection service on a specified host. This function is required prior to any other RTC operations, as they all use the RTC service handle created by this function.

Syntax for C Programmers

```c
extern RTC_Result
RtcStart(RTC_String host,
    RTC_String process,
    RTC_Method method,
    RTC_Timeout timeout,
    RTC_Timeout fudge,
    RTC_Service FAR * service);
```

Syntax for Visual Basic Programmers

```vb
Public Declare Function RtcStart _
Lib "rtc32" Alias "RtcStart_VB" _
(ByVal host As String,
    ByVal process As String,
    ByVal method As Long,
    ByVal timeout As Integer,
    ByVal fudge As Integer,
    ByRef service as Long _
) As Long
```

Parameters

- **host** INPUT
  The name of the host running the RTC server to connect to.

- **process** INPUT
  The process or PATHMON name.

- **method** INPUT
  The connection type to be established: RTC_PROCESS to communicate with a named process or RTC_PATHWAY to access a PATHWAY server class.

- **timeout** INPUT
  The request timeout on the NSK, up to RTC_MAX_TIMEOUT.

- **fudge** INPUT
  The client timeout, up to RTC_MAX_FUDGING.

- **service** OUTPUT
  The RTC service handle returned.
Errors

RTC_SUCCESS If got a ‘service handle’.
RTC_PC_HEAP_EXHAUSTED If insufficient memory.
RTC_INCOMPATIBLE_VERSIONS If mismatched version.
RTC_INVALID_INPUT_PARAMETERS If bad IN-type or NULL OUT-type arguments.
RTC_RPC_ERROR If other RPC error.
RTC_SESSION_TERMINATED If host disconnected.
RTC_SUBSYSTEM_NOT_AVAILABLE If unable to access.
RTC_TIMEOUT If call timed out.
RTC_UNAVAILABLE If can’t connect to host.
RTC_UNHOSTED_ADDRESS If unknown host.

Considerations

☐ All RTC operations on a given RTC service are single-threaded and waited. A client can be multi-threaded by establishing multiple services on the same or a different host by calling this function repeatedly.

☐ The host name must be resolved to a valid IP address on the client system. This can be accomplished by either adding an entry in the local host file, or via DNS. The RTC environment on the specified host must be started and ready to accept incoming connections.

☐ The RTC server is launched by the NNETD process as a result of this call. Therefore, its access ID will be inherited from NNETD. For security reasons, it is recommended that the RTC client program establishes a simple logon sequence prior to issuing I/O requests to the application running on the NSK.

☐ To process this function, the RTC server sends a file system open message to the specified process name. Therefore, this process must be started and ready to accept an open message from the RTC server.

☐ The timeout value associated with each RTC operation defines how long the RTC server is expected to wait to complete the request. The timeout value is the local request timeout on the NSK.

☐ The fudge parameter is an additional timeout allowance to be added to all RTC operations initiated by this RTC service.

☐ This function always returns an RTC service handle, even in the case of failure. If the RTC service could not be established, a special handle is returned which can only be used in calls to RtcInfo to diagnose the source of the problem.
**RtcTranslate()**

Translates an RTC return code into a plain-text error message. The return code is mapped into a string containing an interpretation of the error in US English suitable for display. This is a local operation and never times out.

**Syntax for C Programmers**

```c
extern RTC_Result
RtcTranslate(RTC_Result result,
             RTC_String FAR * string);
```

**Syntax for Visual Basic Programmers**

```vb
Public Function RtcTranslate
    (  ByVal result As Integer,  
       ByRef stringX As String  
    ) As Long
```

**Parameters**

- **result**  **INPUT**  The return code to interpret.
- **string**  **OUTPUT**  The plain-text error message returned.

**Errors**

- **RTC_SUCCESS**  If interpretation successful.
- **RTC_INVALID_INPUT_PARAMETERS**  If bad IN-type or NULL OUT-type arguments.
RtcWrite()

Sends data to a named process. The RTC server implements this function by calling WRITEX on the NSK. For details, please consult the Guardian Procedure Calls Reference Manual.

Syntax for C Programmers

```c
extern RTC_Result
RtcWrite(RTC_Service service,
    RTC_Timeout timeout,
    RTC_BufSize ilen,
    RTC_Buffer ibuf);
```

Syntax for Visual Basic Programmers

```vb
Public Declare Function RtcWRite
Lib "rtc32" Alias "RtcWrite_VB"
(    ByVal service As Long,
    ByVal timeout As Integer,
    ByVal ilen As Integer,
    ByRef ibuf As Any,
) As Long
```

Parameters

- **service**  INPUT  The RTC service handle with an RTC_PROCESS connection.
- **timeout**  INPUT  The request timeout on the NSK, up to RTC_MAX_TIMEOUT.
- **ilen**  INPUT  The length of the data to send, up to RTC_MAX_WRBUFSIZE.
- **ibuf**  INPUT  The address of the data buffer to send.

Errors

- **RTC_SUCCESS**  If message processed successfully.
- **RTC_INVALID_INPUT_PARAMETERS**  If bad IN-type arguments.
- **RTC_INVALID_SERVICE_HANDLE**  If invalid ‘service’ or ‘service’ started for PATHWAY method.
- **RTC_RPC_ERROR**  If other RPC error.
- **RTC_SERVER_ERROR**  If server application problem.
- **RTC_SESSION_TERMINATED**  If host disconnected.
- **RTC_SUBSYSTEM_NOT_AVAILABLE**  If unable to access.
- **RTC_TIMEOUT**  If call timed out.
- **RTC_TOO_MANY_NOWAIT_REQ**  If pending RTC calls.
RtcWriteread()

Sends data to and receives data from a named process. The RTC server implements this function by calling WRITEREADX on the NSK. For details, please consult the **Guardian Procedure Calls Reference Manual**.

Syntax for C Programmers

```c
extern RTC_Result
RtcWriteRead(RTC_Service service,
             RTC_Timeout timeout,
             RTC_BufSize ilen,
             RTC_Buffer ibuf,
             RTC_BufSize maxout,
             RTC_Buffer obuf,
             RTC_BufSize FAR * olen);
```

Syntax for Visual Basic Programmers

```vb
Public Declare Function RtcWriteRead
    Lib "rtc32" Alias "RtcWriteRead_VB"
    ByVal service As Long,
    ByVal timeout As Integer,
    ByVal ilen As Integer,
    ByRef ibuf As Any,
    ByVal maxout As Integer,
    ByRef obuf As Any,
    ByRef olen As Integer
) As Long
```

Parameters

- **service** **INPUT**
  - The RTC service handle with an RTC_PROCESS connection.
- **timeout** **INPUT**
  - The request timeout on the NSK, up to RTC_MAX_TIMEOUT.
- **ilen** **INPUT**
  - The length of the data to send, up to RTC_MAX_WRBUFSIZE.
- **ibuf** **INPUT**
  - The address of the data buffer to send.
- **maxout** **INPUT**
  - The maximum length of the data to receive, up to RTC_MAX_WRBUFSIZE.
- **obuf** **OUTPUT**
  - The address of the data buffer to receive.
- **olen** **OUTPUT**
  - The actual length of the data received.
Errors

RTC_SUCCESS If message processed successfully.
RTC_INVALID_INPUT_PARAMETERS If bad IN-type or NULL OUT-type arguments.
RTC_INVALID_SERVICE_HANDLE If invalid ‘service’ or ‘service’ started for PATHWAY method.
RTC_RPC_ERROR If other RPC error.
RTC_SERVER_ERROR If server application problem.
RTC_SESSION_TERMINATED If host disconnected.
RTC_SUBSYSTEM_NOT_AVAILABLE If unable to access.
RTC_TIMEOUT If call timed out.
RTC_TOO_MANY_NOWAIT_REQ If pending RTC calls.

Considerations

☐ The same buffer address can be used to send and receive data.
8  RTCTEST Script

RTCTEST provides an interactive access to the RTC API calls. It is a line mode program that can be executed on both the Tandem server and the remote clients. Because of its versatility, it is a powerful diagnostic and modelling tool. A sample interactive session is used to demonstrate this program’s capabilities. See Appendix A for more information on the RTC API errors.

You can use RTCTEST to manually simulate how your program will behave. This test program should be run from the client machine.

Invoking RTCTEST

From your TACL prompt, VOLUME to the RTC product subvolume (e.g., $SYSTEM.ZRTC), then type RTCTEST:

```
\JDEV.$DATA3.RTCCONF 23>> volume $system.zrtc
\JDEV.$SYSTEM.ZRTC 24>> rtctest
Remote Transaction Call Test Harness
(version 1.1)
for RTC version 1.0
Copyright Emperex Corporation, Inc. 1997-2004
script source: ‘standard input’
```

Note!

If you hit return without entering any information, the data between the square brackets is assumed. RTCTEST will save your answers to previous prompts to save you from typing information over and over. Both the PATHSEND Read and Write buffer identifiers share a common default value. Similar behaviour can be found in the PTP (process-to-process) read and write buffer identifiers. Only the read/write buffer identifiers have this behaviour.
So, let’s have an overview of the features provided by RTCTEST by simply pressing the return key:

```
Command [HELP]-->

Keywords are case-insensitive. Enter one of the
following (alternate names are separated by '|'):
Allocate | New --> create/define a new buffer
Comment | ; | # | @ | * --> a comment (the rest of line is ignored)
Exit | Quit | Bye --> exit test harness program
Dump | Buffer --> display the contents of a buffer
Release | Free --> release a buffer, allowing reuse/reinitialization
Help | ? --> show this list
LogFile | Log --> create a log of this session (for later use as a
script)

RtcAbortXaction | Xabort --> generate an RtcAbortTransaction operation
RtcBeginXaction | Xbegin --> generate an RtcBeginTransaction operation
RtcEnd | End --> generate an RtcEnd operation
RtcEndXaction | Xend --> generate an RtcEndTransaction operation
RtcInfo | Info --> generate an RtcInfo operation
RtcPathsend | Pathsend --> generate an RtcPathsend operation
RtcStart | Start --> generate an RtcStart operation
RtcTrace | Trace --> generate an RtcTrace operation
RtcTranslate | Xlate --> generate an RtcTranslate operation
RtcWrite | Write --> generate an RtcWrite operation
RtcWriteRead | WriteRead --> generate an RtcWriteRead operation
Show | Status | Display --> display the status of an RTC 'service'
Verbosity --> change the dumping mode for buffers
```
Starting a Process-to-Process Session

First we establish a process-to-process RTC session to $ECHO, which is an RTC test process on a host named dds1.

Note!

You need to include the initial dollar sign ($) in all Tandem process names (e.g., using $ECHO or $echo instead of just ECHO). If you don’t, RTC will appear to fail. It allows all potentially valid RTC API calls, even calls that will fail. If you leave out the dollar sign, you will get an Error 32516 and RtcInfo applied to the resulting invalid service will return an error 11. RTCTEST will exactly mimic the behaviour of the RTC API, even though it may appear confusing at first or that RTC itself is not working.

This process will be automatically started by RTCCOLD if you agreed to start a test environment. The Fudge prompt is different from the timeout prompt which only relates to the application timeout between RTCSERV and a Tandem process.

| Command [HELP]->> start  |
| Host []->> dds1  |
| Pname []->> $echo  |
| Connection [HELP]->> ptp  |
| Timeout [0]->> 10  |
| Fudge [0]->> 20  |
| API Result: 0 == 0x0000 (RTC operation completed successfully)  |
| Started (valid) Service: <<0>>  |

The operation succeeded. The service handle returned is 0. We will refer to this session by using this handle in future RTC calls. Now let’s introduce an error by establishing a session to a non-existent Guardian process:

Note!

Previously-entered values are displayed between square brackets. If they are OK, just hit return.

| Command [start]->> start  |
| host[dds1]->>  |
| Pname [$echo]->> $abc  |
| Connection [ptp]->>  |
| Timeout [10]->>  |
| Fudge [20]->>  |
| API Result: 32516 == 0x7F04 (RTC server successful, but server application had problem(s))  |
| Started (invalid) Service: <<1>>  |
Allocating a Buffer

Since RTCTEST is an interactive programming interface, we first need to allocate and initialize any buffer to be sent. You can experiment with different buffer basis values. HELP was displayed to provide an overview of the different types of buffer you can create. For our session we simply allocate a buffer to be filled with a binary pattern:

```
Command [info]->>> new
Buffer basis [HELP]->>>
Keywords are case-insensitive. Enter one of the following (alternate names are separated by '|'):
Help | ? ->>> show this list
BinFile | Bin | Binary ->>> contents of file is copied into buffer, byte for byte
Pattern | Fill ->>> an 8-bit value fills the buffer to the specified size
HexFile | Hex | Hexadecimal ->>> a Tandem-style hexadecimal dump is copied into the buffer
TextLine | Line ->>> a text line (with terminal NL) is copied to the buffer
TrimmedLine | Trim | Text ->>> a text line (stripped of terminal NL) is copied to the buffer
Buffer basis [HELP]->>> fill
Buffer Size (bytes) [0]->>> 32767
8-bit value [0]->>> 1
Allocated Buffer: <<0>>>
```

We just created a 32K-byte buffer and initialized with binary 01. The allocated buffer handle is 0. That’s how we will refer to it in future calls.

Analyzing Errors

A previous call returned an error and an invalid service handle (1). The error tells us that the RTC interface is OK but a problem occurred at the application level on the server. Using service handle 1, let’s investigate the error:

```
Command [start]->>> info
Service [0]->>> 1
Timeout [10]->>>
Message Buffer [0]->>>
API Result: 0 == 0x0000 (RTC operation completed successfully)
Remote Return Code: 14 == 0x000e (Server error 14)
```

Indeed, we discover the return code from the Tandem file system is 14 (Device does not exist). Later we will learn how to translate the error code into an error message. You should refer to the Allocating a Buffer section. Using an allocated buffer with the info command will allow a text string to be returned in addition to the code value.
Beginning a TMF Transaction

Let’s begin a TMF transaction:

```
Command [info]-->> xbegin
Service [0]-->>
Timeout [10]-->>
API Result: 0 == 0x0000 (RTC operation completed successfully)
```

It worked. Let’s begin another transaction that will fail. We will then analyze why it failed:

```
Command [xbegin]-->>
Service [0]-->>
Timeout [10]-->>
API Result: 32516 == 0x7F04 (RTC server successful, but server application had problem(s))
Command [xbegin]-->> info
Service [0]-->>
Timeout [10]-->>
Message Buffer [0]-->>
API Result: 0 == 0x0000 (RTC operation completed successfully)
Remote Return Code: 83 == 0x0053 (0083 Attempt to begin more concurrent transactions than can be handled)
```

We failed with a file system error 83 because only one transaction can be active per session.
Writing to a Process

Now, let’s WRITE a buffer (0) to $ECHO (the service handle is 0) from a previously-established session. But before we send the buffer, we need to reduce the level of verbosity. Otherwise, RTCTEST displays by default every byte in the buffer. You can experiment with other values for the verbosity level to fit your needs.

```
Command [new]->> verbosity
Level [HELP]->>
Keywords are case-insensitive. Enter one of the following (alternate names are separated by ‘|’):
Help | ? ->>> show this list
Automatic | Auto | Full ->>> dump buffer content in all commands
Demand | Manual | Man ->>> only dump buffer content in DUMP command
Reduced | Minimal | Min ->>> suppress buffer content dump in all commands
Level [HELP]->> min
```

Command [verbosity]->> write
Service [0]->>
Timeout [10]->>
Process Write Buffer [0]->>
Write buffer size: 32767 (of 32767) bytes
API Result: 0 == 0x0000 (RTC operation completed successfully)

The WRITE operation worked. Now let’s create a bigger buffer and watch the WRITE fail:

```
Command [write]->> new
Buffer basis [fill]->>
Buffer Size (bytes) [32767]->> 32768
8-bit value [1]->>
Allocated Buffer: <<1>>
```

Command [new]->> write
Service [0]->>
Timeout [10]->>
Process Write Buffer [0]->> 1
Write buffer size: 32768 (of 32768) bytes
API Result: 32513 == 0x7F01 (invalid value(s) for IN-type parameters and/or NULL OUT-type parameters)

You will notice the API caught the fact that the buffer is too big. It never went over the wire.
Writeread to a Process

To Writeread to a process, we can use the same buffer for input/output, or we can allocate a different buffer for the read operation. Let’s do that and initialize it with nulls:

```
Command [write]->>> new
Buffer basis [fill]->>>
Buffer Size (bytes) [32767]->>> 32767
8-bit value [1]->>> 0
Allocated Buffer: <<2>>>

Command [new]->>> writeread
Service [0]->>>
Timeout [10]->>>
Process Write Buffer [0]->>>
Process Read Buffer [0]->>> 2
Write buffer size: 32767 (of 32767) bytes
API Result: 0 == 0x0000 (RTC operation completed successfully)
Actual Read buffer size: 32767 (of 32767) bytes
```

Ending a Transaction

Let’s end the transaction we started earlier:

```
Command [writeread]->>> xend
Service [0]->>>
API Result: 0 == 0x0000 (RTC operation completed successfully)
timeout [103]->>>
```
Interpreting Error Codes

Let’s end the transaction again to introduce an error. This time we will translate the error returned into an error message. This error message will be stored in a buffer we need to allocate:

```
Command [xend]-->>
Service [0]-->>
Timeout [10]-->>
API Result: 32516 == 0x7F04 (RTC server successful, but server application had problem(s))
```

```
Command [xend]-->> new
Buffer basis [fill]-->>
Buffer Size (bytes) [32767]-->> 100
8-bit value [0]-->>
Allocated Buffer: <<3>>
```

```
Command [new]-->> info
Service [0]-->>
Timeout [10]-->>
Message Buffer [0]-->> 3
API Result: 0 == 0x0000 (RTC operation completed successfully)
Remote Return Code: 75 == 0x004b (0075 No current process transaction identifier)
```

We now read in plain English that the operation failed because our process had no current transaction identifier.

Starting a PATHWAY Session

Let’s establish a PATHWAY session to the test PATHMON $ZRTCP on a host named dds1. This PATHWAY will be automatically started by RTCCOLD if you agreed to start a test environment.

```
Command [info]-->> start
Host [dds1]-->>
Pname [$echo]-->> $zrtcp
Connection [ptp]-->> path
Timeout [10]-->>
Fudge [10]-->>
API Result: 0 == 0x0000 (RTC operation completed successfully)
Started (valid) Service: <<2>>
```

The RTC call succeeded and our new service handle is 2.
**PATHSEND to a Server-Class**

The server class RTC-ECHO under $ZRTCP will be automatically started by RTCCOLD. Let’s send it a PATHSEND call using previously-allocated buffers 0 and 2:

```
Command [start]--> pathsend
Service [0]--> 2
Timeout [10]-->
Server[]-->rtc-echo
Pathsend Write Buffer [0]-->
Pathsend Read Buffer [0]--> 2
Write buffer size: 32767 (of 32767) bytes
API Result: 0 == 0x0000 (RTC operation completed successfully)
Actual Read buffer size: 32767 (of 32767) bytes
```

**Ending a Session**

Let’s end the sessions we started earlier:

```
Command [pathsend]--> end
Service [0]-->
API Result: 0 == 0x0000 (RTC operation completed successfully)

Command [end]--> end
Service [0]--> 2
API Result: 0 == 0x0000 (RTC operation completed successfully)
```

To introduce an error, let’s end the session with service handle 0 again. You will notice the API caught the fact the service handle is no longer valid:

```
Command [end]--> end
Service [0]-->
API Result: 32514 == 0x7F02 (unrecognized “RTC service handle” supplied)
```
Notes:
Appendix A - Error Messages

This appendix defines all the messages generated by RTC on the client side.

There are 13 messages that RTC generates directly; any other messages directly output while RTC is running are not listed here. There are two lists: a list of error code values and associated symbolic names (sorted by value), and a list of symbolic names, associated messages, and detailed explanations (sorted alphabetically by symbolic name).

Symbolic Names and Values

Each symbolic name in the following list is followed by a two-part value in parenthesis and an equivalent decimal value in square brackets. The first part of the parenthetical value is a 16-bit hexadecimal value representing the “subsystem” generating the message. The first two digits indicate the kind of error and the last two digits are the detail. Subsystem 0x0000 messages correspond to like-numbered messages in Tandem’s RSC interface. Subsystem 0x7000 messages correspond to like-numbered messages in the TCP/IP subsystem of Tandem’s RSC. Subsystem 0x7F00 messages are unique to RTC.

- **RTC_SUCCESS (0x0000) [0]**
- **RTC_SUBSYSTEM_NOT_AVAILABLE (0x0003) [3]**
- **RTC_PC_HEAP_EXHAUSTED (0x0016) [22]**
- **RTC_TOO_MANY_NOWAIT_REQ (0x0018) [24]**
- **RTC_UNAVAILABLE (0x7032) [28722]**
- **RTC_SESSION_TERMINATED (0x7036) [28726]**
- **RTC_UNHOSTED_ADDRESS (0x7041) [28737]**
- **RTC_INCOMPATIBLE_VERSIONS (0x7050) [28764]**
- **RTC_INVALID_INPUT_PARAMETERS (0x7F001) [32513]**
- **RTC_INVALID_SERVICE_HANDLE (0x7F002) [32514]**
- **RTC_RPC_ERROR (0x7F003) [32515]**
- **RTC_SERVER_ERROR (0x7F004) [32516]**
- **RTC_TIMEOUT (0x7F005) [32517]**
Symbolic Names, Messages, and Explanations

**RTC_INCOMPATIBLE_VERSIONS (50)**  
RTC or RPC or TCP/IP protocol version mismatch

*Details:* RTC was unable to successfully connect to the requested host machine. The RPC protocol and version for TCP/IP or RTC was incompatible between the client and server host machines.

*Response:* Install compatible versions of TCP/IP and RTC on all machines you intend to use RTC on. Rerun the application.

**RTC_INVALID_INPUT_PARAMETERS (1)**  
Invalid value(s) for IN-type parameters and/or NULL OUT-type parameters

*Details:* An IN-type parameter (inputs to an RTC operation) was invalid/unacceptable for the requested operation, or an OUT-type parameter (pointer to where RTC should place a result from the RTC operation) was NULL. Multiple parameters may be “bad”.

*Response:* Determine the offending RTC call and correct the client application’s code. Rerun the corrected client application program.

**RTC_INVALID_SERVICE_HANDLE (2)**  
Unrecognized “RTC service handle” supplied

*Details:* An RTC service handle has been supplied to an RTC operation and that service handle has been determined to be invalid (not created by **RtcStart()** or since terminated by **RtcEnd()**).

*Response:* Determine the offending RTC call and correct the client application’s code. Rerun the corrected client application program.

**RTC_PC_HEAP_EXHAUSTED (16)**  
Insufficient memory available to complete operation

*Details:* RTC (and the client application) has exhausted all available dynamic memory and can’t create any additional RTC service handles (or any other dynamically-created object).

*Response:* RTC uses dynamic memory only for its service handles. Reduce the number of simultaneous RTC services and check the client application for code that creates large objects (or large numbers of smaller objects). Change the client application to use fewer services and/or less dynamic memory. Increase the amount of real memory available on the client machine and/or virtual memory. Rerun the client application program.
<table>
<thead>
<tr>
<th>RTC_RPC_ERROR (3)</th>
<th>Miscellaneous RPC error</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>details:</strong> An RPC error has occurred which is not otherwise noted (in this list) by the RTC API.</td>
<td></td>
</tr>
<tr>
<td><strong>response:</strong> Use RtcInfo() to determine the 16-bit RPC error code: issue this call within the client application immediately after receiving an RTC_RPC_ERROR value as the RTC_Result from an RTC operation. Correct the problem (possibly changing the application program) and rerun the client application program.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RTC_SERVER_ERROR (4)</th>
<th>RTC server successful, but server application had problem(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>details:</strong> RTC has determined that a non-zero return code was issued by the server application. RTC itself has not detected any problems internally.</td>
<td></td>
</tr>
<tr>
<td><strong>response:</strong> Use RtcInfo() to determine the 16-bit RPC error code. Issue this call within the client application immediately after receiving an RTC_SERVER_ERROR value as the RTC_Result from an RTC operation. Correct the problem (possibly changing the application program) and rerun the client application program.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RTC_SESSION_TERMINATED (36)</th>
<th>RTC service terminated at/by server</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>details:</strong> The host machine for an RTC server has unilaterally closed, dropped or terminated a connection, resulting in a non-functional RTC service.</td>
<td></td>
</tr>
<tr>
<td><strong>response:</strong> This normally indicates either a failure on the server (system or subsystem shutdown, abnormal termination, or crash) or within the communications medium itself. Determine the actual cause, correct it, and rerun the client application program.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RTC_SUBSYSTEM_NOT_AVAILABLE (3)</th>
<th>Unable to connect with RTC server on requested host</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>details:</strong> RTC is not available or installed on the requested server.</td>
<td></td>
</tr>
<tr>
<td><strong>response:</strong> Normally only possible when a service is initially requested (by RtcStart()). Have RTC installed on the necessary host machine and rerun the client application program, or select other hosts which contain the desired server applications (and RTC).</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RTC_SUCCESS (0)</th>
<th>RTC operation completed successfully</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>details:</strong> The RTC operation (and the server application or service) completed the requested operation without error. The RTC_RetCode (obtained via RtcInfo()) will be 0.</td>
<td></td>
</tr>
<tr>
<td><strong>response:</strong> Everything worked properly. No corrective action required.</td>
<td></td>
</tr>
</tbody>
</table>

| RTC_TOO_MANY_NOWAIT_REQ (18) | Illegal (shared) use of an "RTC service handle" |
**details:** Only possible if two processes in the client application share an RTC service handle, which is not allowed or supported — every process must have its own, non-shared, services, even if multiple processes have identically-configured connections to a single server application on a single machine. The shared service is detected when two processes make concurrent, overlapped use of the same service.

**response:** Determine the offending application code and correct it, ensuring each process does not share its RTC service handles with any other process. Rerun the corrected client application program.

**RTC_TIMEOUT (5)**

**RTC operation time expired**

**details:** Excessive RPC timeouts or the RTC timeout specified in an RTC operation was exceeded.

**response:** If the connection between the client and server machines has failed, restore it. If the operation on the RTC server machine took too long, either correct the problem, change the timeout, or explicitly (in the client application) handle this error. Rerun the client application program after correcting the problem.

**RTC_UNAVAILABLE (32)**

**RTC protocol not supported on named host machine**

**details:** RTC is not installed on the host machine (specified in RtcStart()).

**response:** Install RTC on the desired server hosts and check for misnamed hosts in the RtcStart() operation. Rerun the client application program after correcting the problem.

**RTC_UNHOSTED_ADDRESS (41)**

**Unable to locate named host machine**

**details:** The named RTC server host machine (via RtcStart()) could not be located on the network.

**response:** Check the spelling of the host machine’s name. Make sure the machine is available on the network and its name (and IP address) are available to the RTC client machine. Rerun the client application program after correcting the problem.
Notes:
Appendix B - Sample API Client Program

This appendix contains a simple RTC client example in C to demonstrate how to program various RTC calls. It is a quick and easy way to exercise each call once.

A Simple RTC Example

The RTC API is briefly demonstrated by this simple program which exercises some of the functionality within the interface. No checking for errors is performed. The RtcInfo operation does not set the retcde OUT parameter if its return code is not RTC_SUCCESS, so there is some checking to define a known value in this case (in macro DUMP()). The goal is purely to verify that all RTC operations used complete. A full test harness could compare the output from this test with a known good output to verify correctness.

Also note that the use of rtc2rpc.h and rtc_rpc2.h files (not a part of the RTC API) was only required because the program wanted to access internal details of the implementation (the RPC number and version). These two files are not part of the distributed RTC product.

This program is not a complete exercising of RTC. This program makes use of the $ECHO process on Tandem, a (DDS internal) program specifically written to assist in testing RTC. This Tandem program responds to WRITEREAD requests by echoing the WRITE “input buffer” as the READ “output buffer”, limited only by the size of the input buffer and the maximum desired output.

```c
static const char _sid[]
= "@(#)rtc_main.c 1.3 - 07/22/97"
" - Copyright (c) 1997, DDS Inc., all rights reserved"

#include <stdlib.h> /* EXIT_SUCCESS, exit() */
#include <stdio.h> /* stderr, stdout, fflush(), fprintf() */
#include <string.h> /* strlen() */
#include "rtc.h" /* RTC_PROCESS, RTC_SUCCESS, */
/* RTC_BufSize, RTC_RetCode, RTC_Result,*/
/* RTC_Service, RTC_String, RTC_Timeout,*/
/* RtcAbortTransaction(), */
/* RtcBeginTransaction(), RtcEnd(), */
/* RtcEndTransaction(), RtcInfo(), */
/* RtcStart(), RtcTranslate(), */
/* RtcWrite(), RtcWriteRead() */
#include "rtc2rpc.h" /* (for rtc_rpc2.h) */
#include "rtc_rpc2.h" /* RTC, RTC_1 */

#define DUMP(svc) \
if (RtcInfo(svc, &retcode) != RTC_SUCCESS) \
(void) RtcTranslate(result, &string); \
(MSG3("t- t- %d:%d (%s)\n", result, retcode, string);
#define ERR1(fmt,arg1) OUT1(stderr, fmt, arg1)
#define MSG0(fmt) OUT0(stdout, fmt)
#define MSG1(fmt,arg1) OUT1(stdout, fmt, arg1)
#define MSG2(fmt,arg1,arg2) OUT2(stdout, fmt, arg1, arg2)
#define MSG3(fmt,arg1,arg2,arg3) OUT3(stdout, fmt, arg1, arg2, arg3)
#define OUT0(dev,fmt) \
```

Emperex Corporation Appendix B B-1
(void) fprintf(dev, fmt "\n"), fflush(dev)
#define OUT1(dev,fmt,argv1) \(\>
(void) fprintf(dev, fmt "\n", argv1), fflush(dev)
#define OUT2(dev,fmt,argv1,argv2) \(\>
(void) fprintf(dev, fmt "\n", argv1, argv2), fflush(dev)
#define OUT3(dev,fmt,argv1,argv2,argv3) \(\>
(void) fprintf(dev, fmt "\n", argv1, argv2, argv3), fflush(dev)

void
main(int argc, char * argv[])
{
  RTC_RetCode retcode = 0;
  RTC_Result result = RTC_SUCCESS;
  RTC_Service serviceE = (RTC_Service) NULL;
  RTC_Service serviceP = (RTC_Service) NULL;
  RTC_Service serviceN = (RTC_Service) NULL;
  RTC_String string = (RTC_String) NULL;
  RTC_String ECHOpmon = (RTC_String) "RTC-ECHO";
  RTC_String ECHOprog = (RTC_String) "$ECHO";
  RTC_String PMONprog = (RTC_String) "$ZRTCP";
  RTC_String NULLprog = (RTC_String) "$NULL";
  RTC_String message = "This is a test. (DUH!)";
  RTC_Timeout timeout = 10;
  const char * progname = (argc 0 ? argv[0] : "TESTRTC");
  const char * hostname = (argc 1 ? argv[1] : "localhost");

  if (argc != 1)
  {
    ERR1("Usage: %s "hostname", progname);
    ERR1("\tusing %s for hostname", hostname);
  }

  MSG2("RTC (RPC#%d) version %d test START\n", RTC, RTC_1);

  MSG1("ECHOprog = %s", ECHOprog);
  MSG1("ECHOpmon = %s", ECHOpmon);
  MSG1("PMONprog = %s", PMONprog);
  MSG1("NULLprog = %s", NULLprog);
  MSG1("timeout = %d (seconds)\n", timeout);
  MSG0("RtcEnd(serviceN)");
  result = RtcEnd(serviceN);
  DUMP(serviceN);
  MSG1("RtcStart(%s, ECHOprog, RTC_PROCESS, timeout, &serviceE)", hostname);
  result = RtcStart(hostname, ECHOprog, RTC_PROCESS, timeout, &serviceE);
  DUMP(serviceE);

  MSG1("RtcStart(%s, PMONprog, RTC_PATHWAY, timeout, &serviceP)", hostname);
  result = RtcStart(hostname, PMONprog, RTC_PATHWAY, timeout, &serviceP);
  DUMP(serviceP);

  MSG1("RtcStart(%s, NULLprog, RTC_PROCESS, timeout, &serviceN)", hostname);
  result = RtcStart(hostname, NULLprog, RTC_PROCESS, timeout, &serviceN);
  DUMP(serviceN);

  MSG0("RtcBeginTransaction(serviceN)");
  result = RtcBeginTransaction(serviceN);
  DUMP(serviceN);

  MSG2("RtcWrite(serviceN, timeout, %d, %s\n",
      strlen(message), message);
  result = RtcWrite(serviceN, timeout,
strlen(message), (RTC_Buffer) message);
DUMP(serviceN);

MSG0("RtcAbortTransaction(serviceN"));
result = RtcAbortTransaction(serviceN);  
DUMP(serviceN);

MSG0("RtcBeginTransaction(serviceE"));
result = RtcBeginTransaction(serviceE);  
DUMP(serviceE);

MSG2("RtcPathSend(serviceP, timeout, ECHOpmon, %d, \"%s\", 0, NULL,  
NULL")", (int) strlen(message), message);
result = RtcPathSend(serviceP, timeout, ECHOpmon,  
strlen(message), (RTC_Buffer) message,  
(RTC_BufSize) 0,  
(RTC_Buffer) NULL, (RTC_BufSize *) NULL);  
DUMP(serviceP);

MSG2("RtcWriteRead(serviceE, timeout, %d, \"%s\", 0, NULL, NULL)",  
strlen(message), message);
result = RtcWriteRead(serviceE, timeout,  
strlen(message), (RTC_Buffer) message,  
(RTC_BufSize) 0,  
(RTC_Buffer) NULL, (RTC_BufSize *) NULL);  
DUMP(serviceE);

MSG2("RtcWrite(serviceN, timeout, %d, \"%s\")",  
strlen(message), message);
result = RtcWrite(serviceN, timeout,  
strlen(message), (RTC_Buffer) message);
DUMP(serviceN);

MSG0("RtcEnd(serviceN"));
fflush(stdout);
result = RtcEnd(serviceN);  
DUMP(serviceN);

MSG0("RtcEndTransaction(serviceE"));
result = RtcEndTransaction(serviceE);
DUMP(serviceE);

MSG0("RtcEnd(serviceE"));
result = RtcEnd(serviceE);  
DUMP(serviceE);

MSG0("RtcEnd(serviceP"));
result = RtcEnd(serviceP);  
DUMP(serviceP);

MSG2("RTC (RPC#%d) version %d test -END-\n", RTC, RTC_1);
exit(EXIT_SUCCESS);

return;
}
Appendix C - Remote Procedure Call (RPC)
Terminology and Concepts

Brief Overview

RTC is based on NobleNet’s EZ-RPC product. RPC is a mechanism for sending, processing and receiving information between processes on separate machines, or between two processes on the same machine. It doesn’t matter where the machine is located. It allows you to run procedure calls on remote machines. The terms used below have specific meanings in this setting. These terms and others can be found in Appendix D.

port

A channel of communications over TCP/IP. The portmapper is the handler of ports. It listens to lookup requests, and handles them on a well-known port, so it is the only one you need to know about. If you are looking for program XYZ version 2 the portmapper will tell you on what port it is currently listening for requests. There are between 1 and 65 000 ports. The portmapper itself starts and listens on port 111.

program

A specific program number (not the executable) such as NFS or RTC which defines a protocol. It determines which functions or procedures you can call and which parameters you must pass on to the program.

version

As the program changes over time, each change is represented in a new version number. If you change the interface by adding or taking away a parameter of function, you must increase the version number by one.

If you were to request information on another machine, RPC would pass this information between the client and server machines. Clients send the requests through RPC to the servers. Servers actually execute the procedures.

For example, let’s say you wanted to look up some information in a database that was located on another machine in another country. The client machine would contact the portmapper. The portmapper would then listen to the lookup request for the program, and reply with the port number where the program is running. Then, the client would contact the server at the appropriate port. Finally, the server machine would find the information you requested and send a reply back to the client machine.
Appendix D - Glossary

Some of these definitions are based on terms used in John Bloomer’s Power Programming with RPC, O’Reilly & Associates, Inc., Sebastopol, CA. 1991.

API
Application Programmatic Interface.

Bind
The way servers are associated with sockets.

Client
An application which makes procedure calls to be run on remote servers.

Daemon
A program running in the background which lies dormant until it receives a call. Most servers are examples of daemons.

DLL
Dynamic Link Library.

Echo
The name of a program used in the Internet to test the reachability of destinations by sending them an Internet Control-and-Management-Protocol (ICMP) echo request and waiting for a reply.

Handle
An identifier (name of the session) used in RTC calls to mark that session for future reference.

Host
A machine connected to the rest of a network of other machines.

FTP
File Transfer Protocol. The Internet standard, high-level protocol for transferring files from one machine to another. The server side requires the client to supply a logon identifier and password before it honors requests. It makes no assumptions about the file naming structure of the source and destination systems. It allows the file names of each system to be represented in everyday English.

Fudge
An RTCTEST prompt indicating the additional timeout a client application is willing to wait for an operation to complete, including client system delays, and network delays. The total request timeout is therefore the timeout value of the operation plus the fudge value.

IP
Internet Protocol. A way of sending and delivering information between networks.

NFS
Network File System. A protocol developed by SUN Microsystems, Inc., that uses IP to allow a set of cooperating computers to access each other’s file systems as if they were all local. The key advantage of NFS over conventional file-transfer protocols is that NFS hides the differences between local and remote files by placing them in the same name space. NFS is used primarily on UNIX systems, but has been implemented for
many systems, including personal computers like the IBM PC and Apple Macintosh.

**NSK**  NonStop™ Kernel - Tandem's operating system.

**Ping**  Packet InterNet Groper. See Echo.

**Port**  A channel of communications over TCP/IP.

**Portmapper**  The handler of ports. It is the traffic cop which allocates ports to programs. It listens to the same lookup requests, and handles them on a well-known port, so it is the only one you need to know about. If you are looking for program XYZ version 2, the portmapper will tell you on what port it is currently testing for requests. There are between one and 65 000 ports. The portmapper itself starts and listens on port 111.

**Program**  A specific interface (not the executable) such as NFS or RTC which defines a protocol. It determines which functions or procedures you can call and which parameters you must pass on to the program.

**Protocol**  A means of communicating information over a network.

**Register**  To inform the network that a server is there, such as when RPC servers register themselves with the portmapper.

**Reply**  Information sent by the server to the client, usually indicating whether or not the procedure call was successful.

**Request**  Information sent from the client to the server, asking that some information be returned. For example, looking up something in a database.

**RPC**  Remote Procedure Call

**Server**  An application which makes resources available to clients on a network.

**Socket**  A socket is an endpoint for stream-oriented communication. It is an abstraction provided by a set of commonly-used system calls provided by the Berkeley version of UNIX. The Tandem TCP/IP product provides a socket-like interface for access to TCP and UDP communications using IP across an Ethernet LAN.

**Stack**  A communications layer used to pass information to an application. Think of it like a pyramid with the application at the top and the wire at the bottom. Information is stripped off the message as it travels up the stacks or layers. As the application sends information back down, information is added until it reaches the wire and is sent out.

**TCP**  Transmission Control Protocol. The Internet standard transport-level protocol that provides the reliable, full-duplex stream service that many application protocols depend on. TCP allows a process on one machine to send a stream of data to a process
Before transmitting data, participants must establish a connection. Software implementing TCP usually resides on the operating system and uses the IP protocol to transmit information across the Internet. It is possible to terminate (shut down) one direction of flow across a TCP connection, leaving a one-way (simplex) connection. The Internet protocol suite is often referred to as TCP/IP because TCP is one of the two most fundamental protocols. This protocol is used by applications that require reliable end-to-end data transfer. This protocol is a byte-stream-oriented protocol. The protocol includes no concept of packet boundaries. The only guarantee is that all of the data sent will be received in the same order in which it was sent.

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>UDP</td>
<td>User Datagram Protocol. The Internet standard protocol that allows an application program on one machine to send a datagram to an application program on another machine. UDP uses the Internet Protocol to deliver datagrams. Conceptually, the important difference between UDP and IP is that UDP messages include a protocol port number, allowing the sender to distinguish among multiple destinations (application programs) on the remote machine. In practice, UDP also includes a checksum over the data being sent. This protocol provides unreliable datagram service. The integrity of the packets of data sent is maintained. However, the delivery of the packet is not guaranteed. When a packet is received, it is guaranteed to be exactly what was sent from the remote site. Also, the ordering of datagrams is not guaranteed. It is possible to receive packets out of order when using UDP.</td>
</tr>
<tr>
<td>Version</td>
<td>As the program changes over time, each change is represented in a new version number. If you change the interface by adding or taking away a parameter of a function, you must increase the version number by one.</td>
</tr>
</tbody>
</table>