Sun™ MTP Client User's Guide

Release 7.2

Adobe PostScript™

Contents

Figures

Tables

Preface

This document describes how to install and use the Sun™ MTP Client software.

How This Book Is Organized

Chapter 1 describes the client capabilities of Sun MTP Client, the supported protocols, and the operating requirements.

Chapter 2 explains how to install the software.

Chapter 3 describes how to configure the software.

Chapter 4 This chapter describes how to start Sun MTP Client and how to start Sun MTP on the host.

Chapter 5 describes how to administer Sun MTP Client.

Chapter 6 describes how to configure, start and stop the 3270 Terminal application.

Chapter 7 describes how to use the 3270 Printer application.

Chapter 8 describes the Sun MTP Client External Call Interface (ECI) as used with the C programming language.

Chapter 9 describes EPI as C program function specifications and the data structures they require.

Appendix A describes the `KIXTERM.INI` initialization file.

Appendix B lists the Sun MTP Client error messages.

Using UNIX Commands

This document does not contain information about basic UNIX® commands and procedures such as shutting down the system, booting the system, and configuring devices.

See one or more of the following for this information:

■ *Solaris Handbook for Sun Peripherals*
■ AnswerBook2™ online documentation for the Solaris™ operating environment
■ Other software documentation that you received with your system

Typographic Conventions

Typeface	Meaning	Examples
`AaBbCc123`	The names of commands, files, and directories; on-screen computer output	Edit your `.login` file. Use `ls -a` to list all files. `% You have mail.`
`AaBbCc123`	What you type, when contrasted with on-screen computer output	`% `**`su`** `Password:`
AaBbCc123	Book titles, new words or terms, words to be emphasized. Replace command-line variables with real names or values.	Read Chapter 6 in the *User's Guide*. These are called *class* options. You *must* be superuser to do this. To delete a file, type `rm` *filename*.

Shell Prompts

Shell	Prompt
C shell	*machine-name*`%`
C shell superuser	*machine-name*`#`
Bourne shell and Korn shell	`$`
Bourne shell and Korn shell superuser	`#`

Related Documentation

Product	Title	Part Number
Sun Mainframe Transaction Processing software	*Sun Mainframe Transaction Processing Software Administrator's Guide*	816-5327-10
	Sun Mainframe Transaction Processing Software Configuration Guide	816-5328-10
	Sun Mainframe Transaction Processing Software Reference Guide	816-5334-10
3270 Data Streams	*IBM 3270 Information Display System Data Stream Programmers Reference*	CA23-0069

Accessing Sun Documentation

You can view, print, or purchase a broad selection of Sun documentation, including localized versions, at:

`http://www.sun.com/documentation`

Sun Welcomes Your Comments

Sun is interested in improving its documentation and welcomes your comments and suggestions. You can email your comments to Sun at:

`docfeedback@sun.com`

Please include the part number (816-2797-11) of your document in the subject line of your email.

Overview

This chapter describes the client capabilities of Sun MTP Client, the supported protocols, and the operating requirements.

Capabilities of Sun MTP Client

Sun MTP Client provides the following client capabilities:

- 3270 Terminal (only available for Windows)
- 3270 Printer (only available for Windows)
- External Presentation Interface (EPI) application programming interface (API)
- External Call Interface (ECI) API
- Pascal bindings in the form of a Dynamic Link Library (DLL)

These capabilities allow a user to connect a terminal or printer, EPI application or ECI application running on a PC or UNIX machine, directly into one or more Sun MTP server systems running on one or more UNIX machines. In addition, Sun MTP Client provides examples that you can use for ideas when creating your own applications.

Each Sun MTP Client in FIGURE 1-1 can support multiple ECI applications, EPI applications, and 3270 Terminals or Printers as shown in FIGURE 1-2.
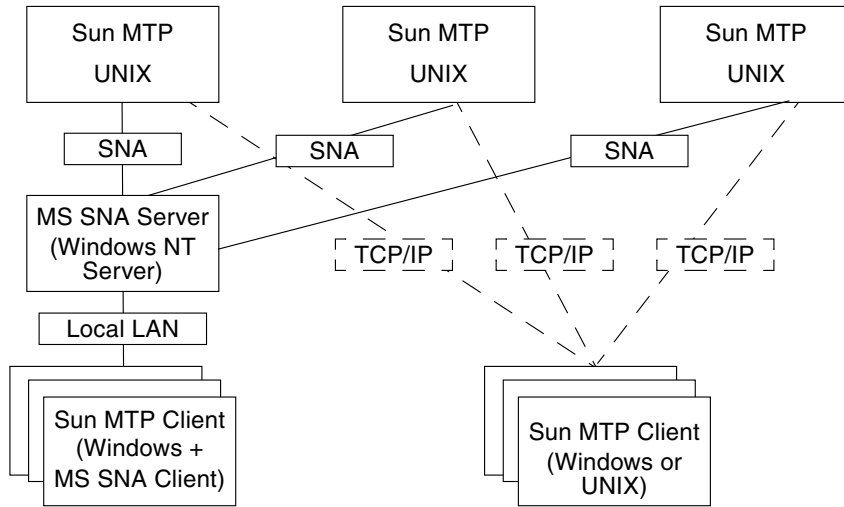
FIGURE 1-1   Sun MTP Client Network Connection Alternatives
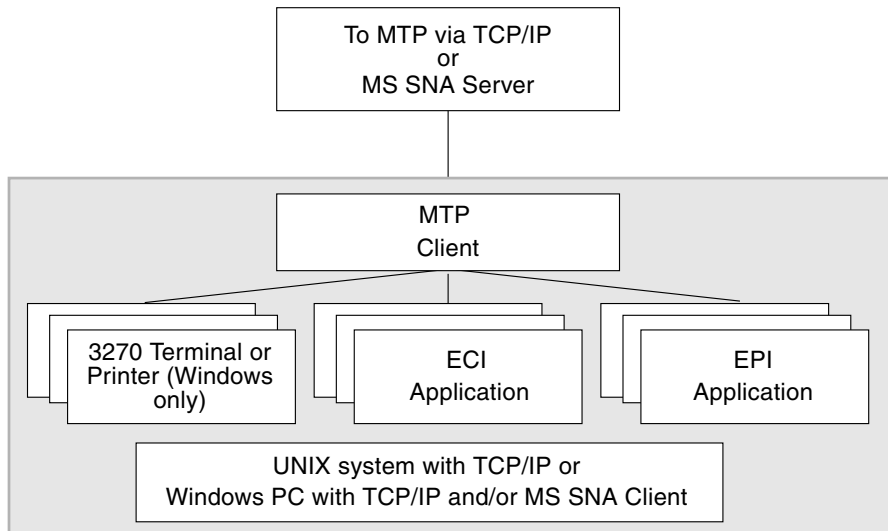
FIGURE 1-2   Sun MTP Client Application Support

Supported Transport Protocols

This section describes how Sun MTP Client supports connection to Sun MTP via TCP/IP and SNA.

TCP/IP

Sun MTP Client supports the TCP/IP protocol. It is available for Solaris and Windows versions.

On Windows, it is implemented by any WINSOCK compliant sockets Dynamic Linked Library (DLL). Chapter 2, describes the method used by Sun MTP Client to determine the version of the DLL it uses, if more than one implementation is installed on your PC.

On Solaris, TCP/IP is part of the operating system.

SNA

Connection from Sun MTP Client for Windows to Sun MTP is supported using either the Microsoft SNA Server or IBM Communications Server. This requires a machine running the Windows Server and the SNA Server that has an LU 6.2 connection configured to each Sun MTP Client region to which access is required. Then you can connect Sun MTP Client for Windows to Sun MTP from one or more Windows machines via the SNA Server (See FIGURE 5-3). Each machine running Sun MTP Client must also have the SNA Client software installed.

Connection between each SNA Client and the SNA Server is achieved using a number of alternatives. For more details, see the documentation for your SNA Server.

SNA is not currently supported for Sun MTP Client on Solaris.

Operating Requirements

Sun MTP Client is available on several platforms. Following are the minimum operating requirements:

- Microsoft Windows NT 4 or Windows 2000
  - `WINSOCK.DLL` or Microsoft SNA Client for Windows NT as described in "Supported Transport Protocols" on page 3.
  - Program development environment of your choice. Explicit support is provided for C and C++.
- Solaris
  - Program development environment of your choice. Explicit support is provided for C and C++.

Terminology

This document uses the following terms:

- *Windows* refers to Microsoft Windows NT Version 4 or Windows 2000
- *UNIX* refers to the Solaris operating environment

This document uses the following conventions when referring to files:

- Where applicable, all files are referred to in their DOS name version, that is, path specifications use the backslash(\) character as the directory separator and may include a drive letter, such as `C:`.
- Where applicable, the term `$INSTROOT` is used to refer to the directory into which Sun MTP Client was installed.

Installation

This chapter provides the steps for installing on the following platforms:

- Windows NT or Windows 2000
- Solaris

Installing Sun MTP Client on Windows

▼ To Install Sun MTP Client on Windows

1. **Start the installation by double-clicking the** `.exe` **file.**

   InstallShield will guide you through the installation process.

2. **Choose the destination folder for the installation.**

   If the default value is not what you require, click Browse to choose a different directory (see FIGURE 2-1).
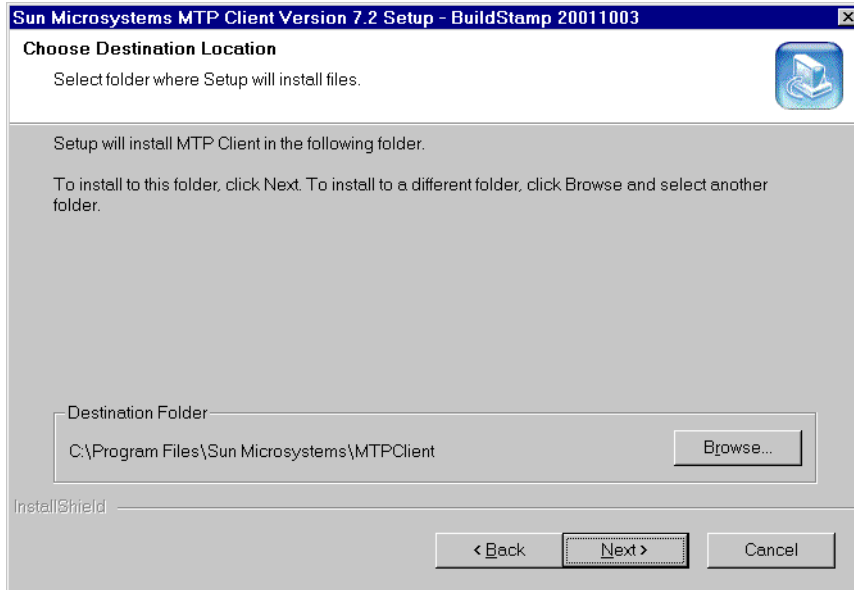
FIGURE 2-1   Choose Destination Location

3. **Make a selection on the Setup Type screen and click Next.**

   There are three choices for the type of installation:

   - Typical: Installs the complete Sun MTP Client. This is sufficient for application development and execution.

   - Compact: Installs only the parts required to run applications. Application development is not possible.

   - Custom: Displays the Select Components screen, which allows you to choose the parts of the product that you want to install. The Production Files component is required for all other components.

4. **When the installation completes, the Setup Complete screen is displayed.**

   There will be an option to reboot your system now or later. However, you must reboot before using the Sun MTP Client software.

   Sun MTP Client will be installed on the Start Menu as shown in FIGURE 2-2.
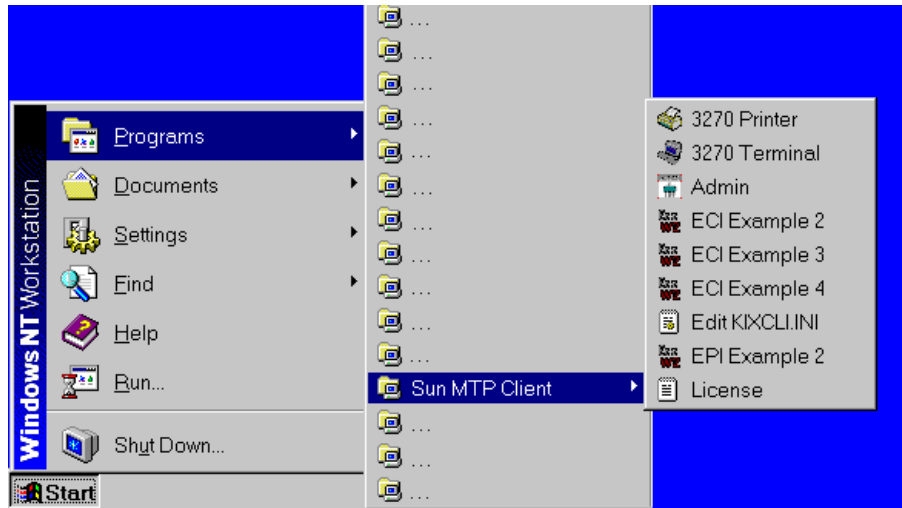
FIGURE 2-2 Sun MTP Client Start Menu

5. **When the installation is complete, you must change the** `PATH` **environment variable.**

You can change the `PATH` environment variable using the Control Panel.

Installing Sun MTP Client on Solaris

Sun MTP Client for Solaris is supplied in a `tar` image. You can install the files wherever it is required on the target machine. The standard installation directory is `/opt/kixcli`.

If the Sun MTP Client is installed into the standard directory, it simplifies the configuration. If, however, the software is installed into a different directory, you must configure the users so that they can access the shared libraries provided by the Sun MTP Client. The system documentation for your platform provides this information. It is strongly recommended that you use the standard directories for installation.

Configuring Sun MTP Client and Sun MTP

There are two procedures you must perform before using Sun MTP Client to connect to Sun MTP.

- Configuring Sun MTP Client, described in "Configuring Sun MTP Client" on page 9
- Configuring Sun MTP, described in "Configuring Sun MTP for a TCP/IP Connection to Sun MTP Client" on page 15

Configuring Sun MTP Client

Sun MTP Client can connect a machine running Windows or Solaris to multiple Sun MTP regions simultaneously (see FIGURE 1-1). Each of these regions must be defined to Sun MTP Client in the Systems section of KIXCLI.INI, a control file. The location of this file is platform dependent:

- Solaris platforms: /opt/kixcli/config
- Windows: C:\KIXCLI\CONFIG

The default version of the file is illustrated in CODE EXAMPLE 3-1.

Before running Sun MTP Client, you must edit this file to define the Sun MTP regions, as described in "Adding Systems to the KIXCLI.INI File" on page 10. In addition, you can modify some of the behavior of Sun MTP Client by entries in the General section of the file as described in "Defining Client Behavior in the KIXCLI.INI File" on page 12.

CODE EXAMPLE 3-1   KIXCLI.INI Control File

```
;-------------------------------------------------------------
;
:Sun MTP Client Configuration File
;-------------------------------------------------------------

[Systems]
Accounts=TCP,abc.mycompany.com,9111,Customer Accounting System
Payroll=TCP,def.mycompany.com,9111,Employee Payroll System
Ordering=TCP,ghi.mycompany.com,9111,Internal Ordering System
Test=TCP,555.555.55.5,9111,Test System
SnaSyst=MSSNA,RLUALIAS,LLUALIAS,MODENAME,MS SNA Connected System

[General]
DefaultSystem=Accounts
;TraceDir=C:\tmp
;TraceMask=0
;MsgDir=C:\tmp
;MaxRequests=20
;MaxSystems=3
;EnableConnect=false
```

Adding Systems to the KIXCLI.INI File

Each line in the Systems section defines a Sun MTP region. Add an entry in the Systems section for each region that you plan to use. The format of each region entry is different, depending on the transport protocol used to connect to that system (TCP/IP or SNA). The use of SNA as a transport is only supported on the Microsoft Windows versions of Sun MTP Client, not on the Solaris version.

Required Fields for a TCP/IP-Connected System

All fields are required. Even if you do not enter a comment, you must type a comma delimiter after the port number field.

The fields in each entry are described in the following table:

Field	Description
Name	Name (up to 8 characters) by which the region is known by Sun MTP Client.
Transport protocol	Set to the characters, TCP, to indicate the definition of a TCP/IP-connected system.
Host address	TCP/IP address of the host where the region is running.
Port number	TCP/IP port number on which the region is listening for TCP connections from Sun MTP Client. On the UNIX host, the Sun MTP server, unikixmain, must be started with the -P option and this port number (see "Starting Sun MTP Client on Windows" on page 17).
Comment	Comment (up to 60 characters), which can be zero length. However, you must type a comma to delimit the port number field.

In CODE EXAMPLE 3-1, the first line in this section, Systems defines a system that will be known locally as Accounts. This system uses the TCP transport to connect to a Sun MTP region that is running on host abc.mycompany.com and listening for TCP connections from Sun MTP Clients on port 9111. The last field contains a description of the system, Customer Accounting System.

Required Fields for an SNA-Connected System

All fields are required. Even if you do not enter a comment, you must type a comma delimiter after the port number field.

The fields in each entry are described in the following table:

Field	Description
Name	Name (up to 8 characters) by which the region is known by Sun MTP Client.
Transport protocol	Set to the characters, MSSNA, to indicate the definition of a SNA -connected system.
Remote LU Alias	Set to the LU alias of the Sun MTP region to which to connect as defined on the SNA Server. This is referred to on the SNA Server as a remote or partner LU alias.

Field	Description
Local LU Alias	Set to the local LU alias that Sun MTP Client should use as defined on the SNA Server. This is referred to on the SNA server as an LU Alias.
ModeName	Set to the APPC ModeName to be used for the connection between the local and remote LUs as defined on the SNA Server.
Comment	Comment (up to 60 characters), which can be zero length. However, you must type a comma to delimit the port number field.

In CODE EXAMPLE 3-1, the last line in the Systems section defines a system that is known locally as SnaSyst. This system uses Microsoft SNA or IBM Communications Manager to connect to Sun MTP using the local LU LLUALIAS, where Sun MTP has a remote LU Alias of RLUALIAS. The APPC mode to be used is MODENAME. The last field contains a description of the system, MS SNA Connected System.

Defining Client Behavior in the KIXCLI.INI File

Entries in the General section of the KIXCLI.INI file control the behavior of Sun MTP Client and are optional. These entries take the form of Attribute=value pairs, and are described below.

Note – Use the semicolon character to comment out any entries you do not need. As shipped and illustrated in CODE EXAMPLE 3-1, most of the entries in the General section are commented out.

DefaultSystem=*System_name*

Sets which default system to use by Sun MTP Client; system must be defined in the Systems section. If DefaultSystem is not specified, it defaults to the first entry in the Systems section of the KIXCLI.INI file. For example, Accounts in CODE EXAMPLE 3-1 is the default system even if this entry is commented out.

TraceDir=C:\TMP

Controls the destination directory for Sun MTP diagnostic trace files. Each trace file is given the name *.trc, where * is a file name root identifying the task or process concerned. If TraceDir is not specified, it defaults to $INSTROOT/BIN.

TraceMask=0

A decimal integer that controls the generation of a diagnostic trace by Sun MTP Client. This entry should be left commented out or set to zero, unless you are asked to obtain a diagnostic trace by your authorized Sun service provider.

When using diagnostic trace, be aware of the following:

- Trace files `*.trc` can quickly use large quantities of disk space. Therefore, you should disable trace and delete unwanted trace files as soon as possible.
- Generation of diagnostic traces degrades the performance of Sun MTP Client.

`MsgDir=C:\TMP`

As part of its normal function, Sun MTP Client generates messages, which are written to the file `KIXCLI.MSG` in the `MsgDir` directory. If `MsgDir` is not specified, it defaults to `$INSTROOT\BIN`. The message file, `KIXCLI.MSG`, is overwritten each time Sun MTP Client is started.

`MaxRequests=20`

Defines the maximum number (default 20) of concurrent requests allowed by Sun MTP Client. A request is defined as an ECI unit-of-work, an EPI, or terminal emulator session. This parameter is designed to limit any problems that an errant application may cause, such as an ECI application that erroneously loops, continually starting new units-of-work, but never completing them.

`MaxSystems=3`

Defines the maximum number (default 3) of systems to which Sun MTP Client can be simultaneously connected.

`EnableConnect=false`

Allows you to configure the action taken by Sun MTP Client when the right mouse button is pressed from the Systems display.

| `true` | Displays the menu for connecting or disconnecting from a Sun MTP region. (Default) |
| `false` | Does not display the connect/disconnect menu. This facility is provided for environments in which users should not be able to select a system. This is the recommended value for production installations. |

Enabling and Disabling a Diagnostic Trace

For support purposes, you may be asked by your authorized Sun service provider to obtain diagnostic information by running a diagnostic trace of Sun MTP Client.

▼ To Run the Diagnostic Trace

1. **Stop Sun MTP Client.**

2. **Ensure that the** `TraceDir` **attribute in the Sun MTP Client configuration file,** `KIXCLI.INI`**, is uncommented, and that the directory specified exists, is writable, and has sufficient space for diagnostic trace output.**

3. **Ensure that the** `TraceMask` **attribute in the Sun MTP Client configuration file,** `KIXCLI.INI`**, is uncommented and set to the value requested by your authorized Sun service provider.**

4. **Delete any unnecessary existing diagnostic trace files** `*.trc` **from the** `TraceDir`**.**

5. **Restart Sun MTP Client.**

Note – On Windows NT you can also run a diagnostic trace from the Control panel of the admin program, `KIXCTLG`.

▼ To Disable the Diagnostic Trace

1. **Stop Sun MTP Client.**

2. **Edit the** `KIXCLI.INI` **file, and set the** `TraceMask` **back to zero (or comment out the** `TraceMask` **line) to disable diagnostic trace.**

3. **Copy the trace file(s)** `*.trc` **to another location before restarting Sun MTP Client.**

Configuring Sun MTP for a TCP/IP Connection to Sun MTP Client

Before using Sun MTP Client to connect to a Sun MTP region, the region must be configured to accept incoming TCP/IP connections. Access to the region by the PC Client is provided by specifying a port by number or name when starting the Sun MTP server on the host. The name is used to look up a well-known port number in `/etc/services` or NIS tables.

▼ To Configure Sun MTP

1. **On the host, define the well-known port in** `/etc/services` **or NIS tables, for example:** `cicstcp 1435/tcp`**.**

   The well-known port can be used for one region. If additional regions running on the same machine will listen for a TCP connection, each Sun MTP server must identify a port that is unique on that machine.

2. **Define the** `TCPRTERM` **environment variable in your Sun MTP setup file.**

   This sets the maximum number of concurrent inbound requests for TCP/IP connections from Sun MTP Client and remote Sun MTP or CICS regions. If there are more requests than available sessions, Sun MTP queues the extra requests, which may affect performance.

   Each `$TCPRTERM` configured requires 32KB of shared memory.

3. **Define the** `TCPSTERM` **environment variable in your Sun MTP setup file.**

   This sets the maximum number of concurrent outbound requests for TCP/IP connections to remote Sun MTP or CICS regions. If there are more requests than available sessions, Sun MTP queues the extra requests, which may affect performance.

   Each `$TCPSTERM` configured requires 32KB of shared memory.

4. **Define the** `KIXMAXIST` **environment variable in your Sun MTP setup file.**

   This sets the maximum number of autoinstalled Sun MTP Client and remote Sun MTP or CICS regions. A request from a remote region is rejected if there are no available entries to install the region.

   See also "Enabling Sun MTP to Receive Connections" on page 18 and the *Sun Mainframe Transaction Processing Software Configuration Guide* for additional information.

Configuring Sun MTP and SNA for an SNA-Connected Sun MTP Client

For information about configuring Sun MTP and SNA for a Sun MTP Client, refer to the chapter describing Intersystem Communication (ISC) in the *Sun Mainframe Transaction Processing Software Configuration Guide*.

Starting Sun MTP Client and Sun MTP

This chapter describes how to start Sun MTP Client and how to start Sun MTP on the host.

What should I start first? Although you can start the processes in either order, you cannot connect to a region that is not started. When you start the region, the server listens for TCP and SNA requests. If you start an application automatically, it fails if the region is not started.

Starting Sun MTP Client on Windows

Sun MTP Client for Windows systems is designed to run as a Windows Service. Normally, it is not necessary to start Sun MTP Client; it starts for you. However, there are cases where you may want to control the starting and stopping of Sun MTP Client.

When the software is installed, a service is created that is started each time your machine is booted. However, this may not be the mode of operation that you require. Whether or not the service is started is determined using the Control Panel, Services option. If the Sun MTP Client service has the automatic flag set, it is started when your machine boots. To modify this setting, press the Startup button, then select the options you require.

Sun MTP Client can be started using the Control Panel, Services option. This operation is performed in the same way as for all services, by selecting the required service and pressing the Start button.

In addition to using the Control Panel to start and stop Sun MTP Client, you can use the Sun MTP Client Administrator (see "Administering Sun MTP Client on Solaris" on page 25).

Starting Sun MTP Client on Solaris

Sun MTP Client for Solaris is designed to run as a daemon process. It can be started manually from the shell, using the `kixcli` command, but this is not the normal method of operation. Normally, you should run Sun MTP Client as a daemon started from `inittab`. This means that ECI/EPI facilities are available whenever the machine is running.

To create an `inittab` entry, consult your operating system documentation for the exact syntax for your platform. The following is an example of an `inittab` entry:

```
kixcli:2:once:/opt/kixcli/bin/kixcli >/dev/console 2>&1
```

Enabling Sun MTP to Receive Connections

The connection on the Sun MTP host is established by starting the Sun MTP server, `unikixmain`, on the host with the `-P` option. When started in this way, in addition to starting transaction servers, it starts the `unikixtcp` server, which listens on the port number for incoming requests.

The server also provides the `-L` option for setting the number of file descriptors.

`-L` *connections*

Number of socket connections that the Sun MTP server can support for TCP connections. This number represents the maximum number of connections that can be opened by a process. Reduce the number to restrict the number of TCP Clients that can be attached or raise it if Clients are being rejected.

The default is the current system soft limit for files.

`-P` *port#*

Port name or number for the Sun MTP server to use as a listening port for CICS Clients running ECI and EPI applications over a TCP/IP connection. There is no default port so you must supply one, either by name or by number, for example:

```
-P cicstcp
-P 5100
```

The port number must match the port number of a system identified in the Client INI file. For example, to listen for requests from the Payroll system listed in CODE EXAMPLE 3-1, type port number 9111 as the argument to the -P option. Note that cicstcp is the name defined in /etc/services in "Configuring Sun MTP for a TCP/IP Connection to Sun MTP Client" on page 15.

You can start unikixmain directly or use the shell script, kixstart, to start it indirectly. kixstart passes any command line options to unikixmain. The *Sun Mainframe Transaction Processing Software Configuration Guide* describes the procedure for starting Sun MTP and the *Sun Mainframe Transaction Processing Software Reference Guide* describes all the options for unikixmain.

Sun MTP Client Administration

This chapter describes how to administer Sun MTP Client on Windows and Solaris platforms.

Administering Sun MTP Client on Windows

As stated in "Starting Sun MTP Client on Windows" on page 17, Sun MTP Client for Windows runs as a system service. Therefore, you can use the Windows Control Panel to perform some of the administration of the Client. However, you cannot do all the administration functions.

Sun MTP Client contains a program (`KIXCTLG.EXE`) that is a monitor and administrator for the Client. `KIXCTLG.EXE` is a windowed application that contains a single window with up to four notebook-style tabs. Using these tabs, you can select up to four displays of information, which are described in the following sections

- Control panel
- TCP Systems panel
- MS SNA Systems panel
- Messages display

Control Panel

On the Control panel, it is possible to perform two types of tasks:

- Start and stop Sun MTP Client. On Windows, pressing the Start/Stop buttons is the equivalent of starting and stopping the system service.

■ Trace enables you to dynamically control the trace output by Sun MTP Client. Only use trace under the supervision of the Sun MTP system administrator.

The following figure shows an example of the Control Panel.
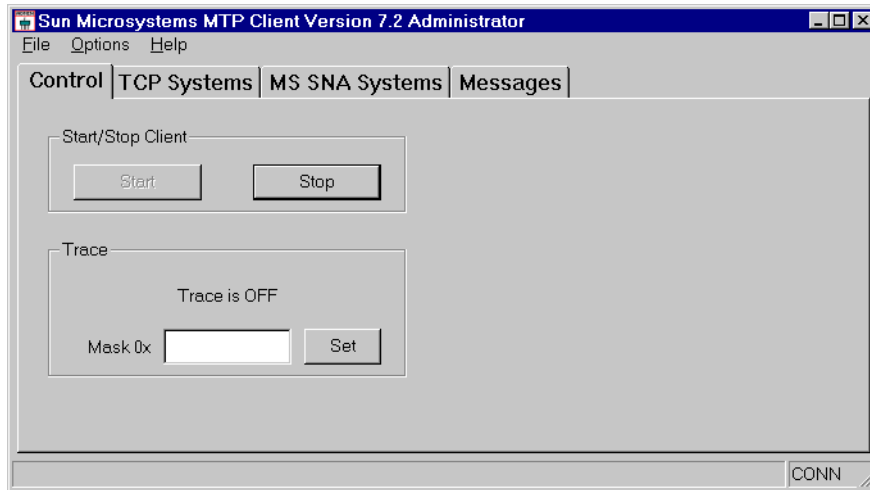
FIGURE 5-1   Control Panel

TCP Systems Panel

The TCP Systems panel is only displayed when there are TCP systems defined in the KIXCLI.INI file. This panel contains a list of the defined TCP systems, along with their configuration values. It also gives an indication of whether Sun MTP Client is connected to those systems.

Each system has an icon to the left of it:

Icon	Indicates ...
X	The system is not connected.
-	The system is being connected.
(Check)	The system is connected.

The systems in the list are manipulated by pressing their name button. This produces a menu that allows the connection or forced disconnection of the system. Note that you should only use disconnection when absolutely necessary.
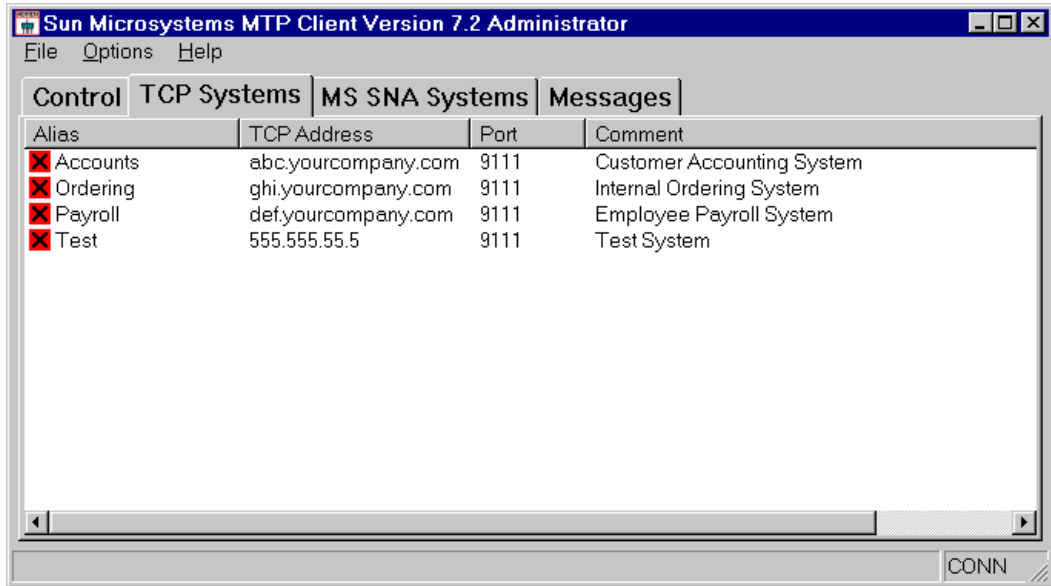
FIGURE 5-2   TCP Systems Panel

MS SNA Systems Panel

The MS SNA Systems panel only displays when there are MS SNA systems defined in the KIXCLI.INI file.

This panel contains a list of the defined MS SNA systems, along with their configuration values. It also gives an indication of whether Sun MTP Client is connected to those systems.

Each system has an icon to the left of it:

Icon	Indicates ...
X	The system is not connected.
-	The system is being connected.
(Check)	The system is connected.

The systems in the list are manipulated by pressing their name button. This produces a menu that allows the connection or forced disconnection of the system.

Note – You should only use disconnection when absolutely necessary.
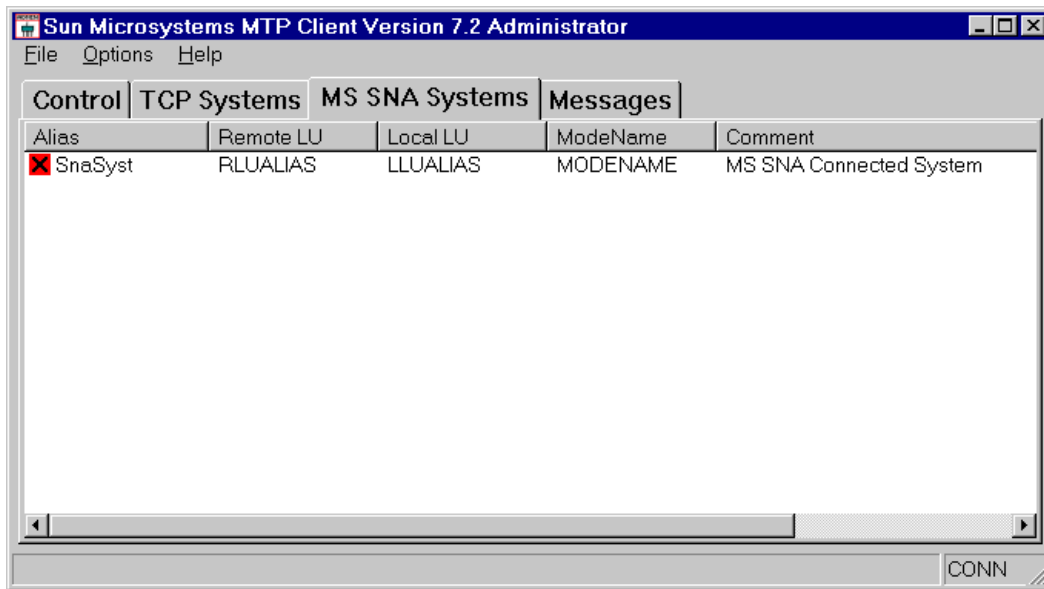
FIGURE 5-3 MS SNA Systems Panel

Messages Panel

The Messages panel shows the 100 most recent messages issued by Sun MTP Client. There are three message categories that display in different colors:

- Informational (I)
- Warning (W)
- Error (E)

The same messages are also written to a message file, KIXCLI.MSG, and include a date and time stamp. Appendix B describes the Sun MTP Client messages.
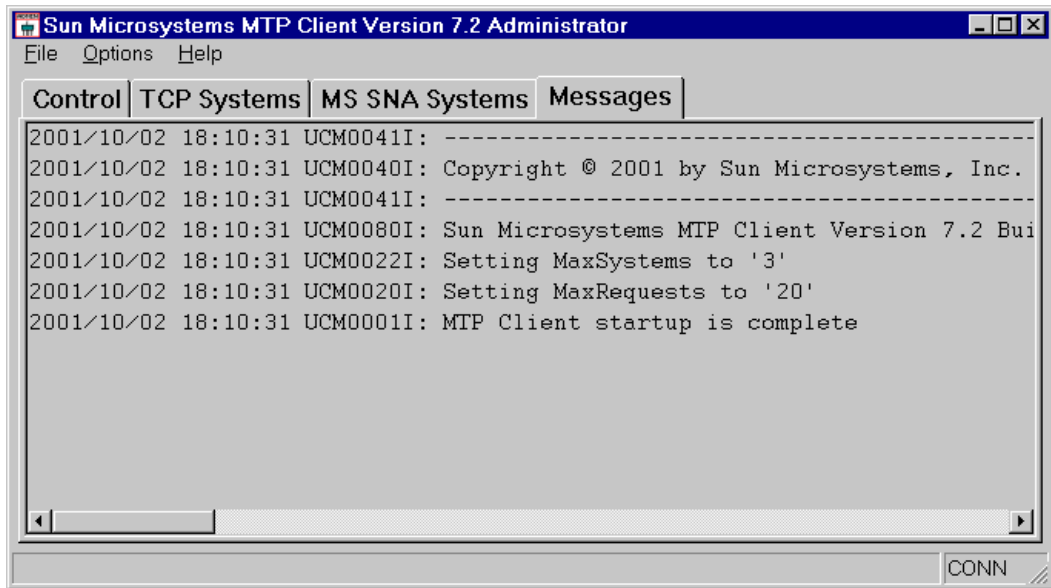
FIGURE 5-4 Messages Panel—Example

Administering Sun MTP Client on Solaris

There is no graphical administration tool available for Sun MTP Client on Solaris; you must use `kixctl`.

Format

`kixctl [-v] [-s] [-l] [-m] [-D] [-c` *system*`] [-d` *system*`] [-t` *mask*`]`

where:

-v	Prints out the Sun MTP Client version information.
-s	Shuts down Sun MTP Client.
-l	Lists the defined systems and their status.
-m	Displays messages when the status of a system changes.
-D	Performs a dump. Only the system administrator should perform this function.

-c *system*	Connects to the named *system*.
-d *system*	Disconnects from the named *system*.
-t *mask*	Sets the trace mask to *mask*.
	Only the system administrator should perform this function.

The system administrator should monitor the `kixcli.msg` file, which contains information about any significant errors and/or events that occur on the system.

3270 Terminal

The 3270 Terminal is a Windows application that uses the Sun MTP Client EPI interface to provide a 3270 gateway into Sun MTP. The terminal allows remote access to standard 3270 Sun MTP transactions from a PC running Windows. You can configure the terminal using command line parameters and an initialization file.

This chapter describes how to configure, start and stop the 3270 Terminal.

Note – The 3270 Terminal is not available for the Solaris operating environment.

Configuring the 3270 Terminal

The terminal is configured using either of the following methods:

- Command line parameters to configure the startup conditions; for example, the window name, the system, the transaction, as described in "To Configure Using Command Line Parameters" on page 27.

- Initialization file to configure the colors, key mappings and field attributes, as described in "To Configure Using the Initialization File" on page 28.

To Configure Using Command Line Parameters

1. **Highlight the 3270 Terminal icon from the Sun MTP Client program group.**

2. **Select** Properties **from the Program Manager File menu.**

3. **When the dialog box for** kixterm.exe **appears, add the command line parameters you need using the following syntax:**

- Specify each parameter with either a "-" or "/" character followed by the parameter identifier either lower- or upper case.
- Separate data passed with a parameter flag from the flag by at least one space character.
- If an argument to a parameter includes spaces, enclose the argument with quotation marks, for example:

```
kixterm /t "CEBR" -w "Accounts Payable" /s Accounts
```

The following command line parameters are accepted:

/d *device_type*

Sun MTP device type for this terminal; truncated if it exceeds 16 characters. "CICS_EpiAddTerminal()" on page 88, lists the Sun MTP valid models.

/i *file_name*

Name of which initialization file to use. If not specified, the terminal uses the supplied initialization file, KIXTERM.INI.

/n *netname*

Sun MTP netname (up to 8 characters) to assign to this terminal. This name is assigned in the TCT and is normally obtained from the System Administrator.

/s *system_name*

Name (up to 8 characters) of the Sun MTP region to which to connect, for example, Accounts, as illustrated in CODE EXAMPLE 3-1. You must define this name in KIXCLI.INI before the terminal is started. If not specified by the start-up command line, a selection dialog appears to allow the user to choose the system.

/t *tranid*

Initial transaction and data. Specifies which transaction to run immediately after a successful connection to Sun MTP.

/w *title*

Window title (up to 50 characters) to be displayed on the terminal title bar.

▼ To Configure Using the Initialization File

The terminal is supplied with a default initialization file, KIXTERM.INI, that contains the settings for the colors and keyboard used by the terminal.

1. **Open the default initialization file or create your own.**

2. **Specify the settings you want.**

3. **Save the initialization file in the Windows default system directory.**

During installation, the KIXTERM.INI file is placed in the default system directory if a copy does not already exist. Appendix A describes the format and contents of the KIXTERM.INI file.

Starting the 3270 Terminal

This section describes how to start the terminal and connect to a system from the 3270 Terminal icon and from a command line.

▼ To Start the 3270 Terminal From the 3270 Terminal Icon

1. **Configure the 3270 Terminal as described in "Configuring the 3270 Terminal" on page 27.**

2. **From the Sun MTP Client group, double click the 3270 Terminal icon.**

3. **If the System Selection dialog appears, as illustrated in** FIGURE 6-1**, connect to a system:**

   ■ Double click the system.

   ■ Single click the system, then click the OK button.

Note – If the terminal was configured in the Program Manager with a system name, the System Selection dialog does not appear.

FIGURE 6-1   3270 Terminal System Selection Dialog

▼ To Start the 3270 Terminal From a Command Line

1. **Select** Run **from the Program Manager File menu.**

2. **Type the** kixterm **command in the Command box with the parameters desired.**

   See "To Configure Using Command Line Parameters" on page 27 for a description of the parameters and syntax.

Note – To bypass the System Selection dialog, specify the /s parameter and a system name; the 3270 Terminal automatically attempts to connect to the specified system.

3270 Terminal Screen

The 3270 Terminal screen that is created on startup displays either:

- A blank screen if no transaction was specified in the command line.
- The specified transaction screen. The following figure illustrates a screen.
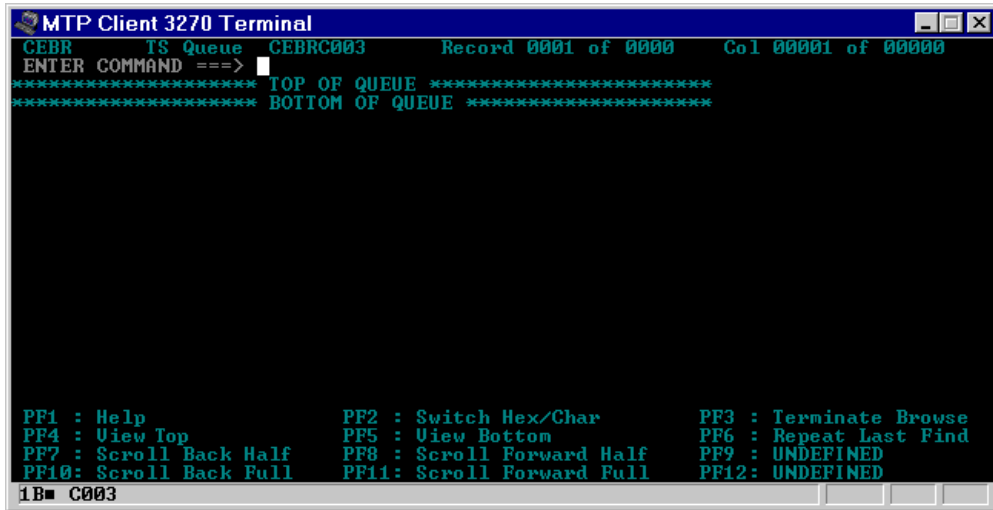
FIGURE 6-2 3270 Terminal Screen

where:

Title Bar	Displays the title assigned to the terminal, by default, Sun MTP Client 3270 Terminal. You can change it with the command line parameter -W or /W.
Display Area	Area where you type Sun MTP transactions and where data is displayed by transactions.
Status Bar	Displays information about the current status of the terminal session as described in TABLE 6-1.

TABLE 6-1   3270 Terminal Status Bar Information

Information Displayed	Example	Description
Connection status	1B#	Symbols shown in the bottom left corner when a connection is established
Terminal netname	=2AAA3M	Name of the terminal on the remote region
Keyboard lock	XSystem	Symbol displayed when the keyboard is locked.
Error messages	EMU0014E: System unavailable	Keyboard status flags

TABLE 6-1 3270 Terminal Status Bar Information *(Continued)*

Information Displayed	Example	Description
Keyboard status flags	CAPS	Caps lock indicator
	NUM	Num lock indicator
	INS	Insert Mode flag

Stopping the 3270 Terminal

▼ To Shut Down the 3270 Terminal

1. **Close the connection with the** CSSF LOGOFF **command.**

2. **Double-click the Control-menu box in the upper left hand corner of the 3270 Terminal screen (**FIGURE 6-2**).**

3270 Printer

The 3270 Printer is a Windows application that uses the Sun MTP Client EPI interface to provide a 3270 gateway into Sun MTP. The printer allows printer-based applications to send data to a virtual printer device. The printer can either

- Format and write data to a text file on the local host.
- Format the data to a temporary file, then execute a command to process the file.

Data written to a file can be edited for inclusion in spreadsheets, word processor documents or other applications.

The printer formats data according to the printing rules specified in the *IBM 3270 Information Display System Data Stream Programmers Reference*.

Note – The 3270 Printer is not available for the Solaris operating environment.

Configuring the 3270 Printer

This section describes how to configure the printer using the command line parameters.

▼ To Configure the Printer

1. **Highlight the Sun MTP Printer icon from the Sun MTP Client program group.**

2. **Select** `Properties` **from the Program Manager File menu.**

3. **When the dialog box for** `kixprnt.exe` **appears, add the command line parameters you need**

- Specify each parameter with either a "-" or "/" character followed by the parameter identifier in either lower- or uppercase.
- Separate data passed with a parameter flag from the flag by at least one space character.
- If an argument to a parameter includes spaces, enclose it with quotation marks, for example

```
kixprnt /t "hp sys1" /f c:\acct.txt  /s Accounts
```

The following command line parameters are accepted:

/d *device_type*

Sun MTP device type, up to 16 characters, for this virtual printer device.

/f *print_file*

File to which formatted output is appended. If no file name or print command are specified, the printer opens and appends to a default file, `kixprnt.txt`. You can edit the print file and extract data for use in other applications. The printer does not delete the file. If both /f and /p are given, the printer ignores the file specified with /f and writes to a temporary file created by the printer as described in /p.

/n *netname*

Sun MTP netname, up to 8 characters, to assign this virtual printer device. Check with your system administrator.

/p *command*

Command run by the printer every time a print request is received. The name of the temporary file to process is appended to the end of the command string. For example, for this command

```
kixprnt /p hppmt
```

the printer appends the name of the temporary file forming this command

```
kixprnt /p hppmt  c:\tmp\123
```

The user command must delete the temporary file. See also "Example `kixprnt` Commands" on page 35.

/s *system_name*

Name, up to 8 characters, of the Sun MTP region to which to connect. If not specified at start-up, select a system from the `System` selection dialog. "Adding Systems to the `KIXCLI.INI` File" on page 10 describes how to add systems to the `KIXCLI.INI` file.

/t *trans_id* [*data*]

Transaction to run when the printer has successfully connected to Sun MTP. If passing data with the initial transaction, you must enclose the transaction ID and data in quotes, for example:

```
kixprnt /t "CEBR testq"
```

Example `kixprnt` Commands

This section contains examples of `kixprnt` commands.

Example 1

Write text to the `3270.txt` file in the `c:\` directory:

```
$ kixprnt /f c:\3270.txt
```

Example 2

Writes the print text to a temporary file, then executes a user-written command, `usercmd`, with the name of the temporary file appended to the end of the command:

```
$ kixprnt /p usercmd
```

Example 3

If the temporary file for the previous command was created as `c:\tmp\453`, the user command executed is:

```
$ usercmd c:\tmp\453
```

Example 4

The name of the temporary file is always appended to the end of the command. Therefore, if the kixprnt command had flags defined as /c and /g, for example:

```
$ kixprnt /p "usercmd /c /g"
```

then the user command executed is:

```
$ usercmd /c /g c:\tmp\453
```

Starting the 3270 Printer

The following subsections describe the two methods for starting the printer and connecting to a system

- From the 3270 Printer icon
- From a command line

▼ To Start the 3270 Printer From the 3270 Printer Icon

1. **From the Sun MTP Client group, double click the 3270 Printer icon.**

2. **If the System Selection dialog appears, illustrated in** FIGURE 7-1**, select a system to connect to using either method:**

   - Double click the system.

   - Single click the system, then select OK.

   Note – If the printer was configured in the Program Manager with a system name, the System Selection dialog does not appear.
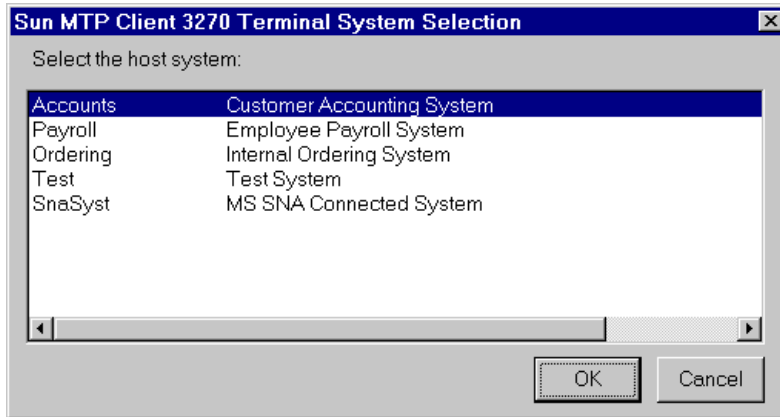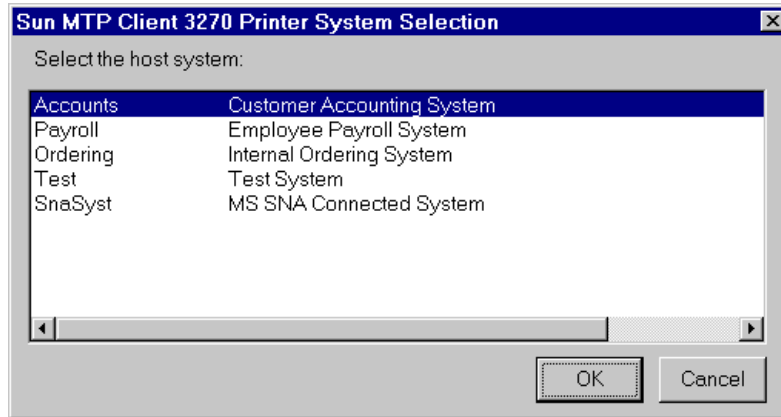
FIGURE 7-1   3270 Printer System Selection Dialog

▼ To Start the 3270 Printer From a Command Line

1. **Select** `Run` **from the Program Manager File menu.**

2. **Type the** `kixprnt` **command in the Command box with the parameters desired.**

   See "Configuring the 3270 Printer" on page 33 for a description of the parameters.

3270 Printer Screen

On startup, the printer appears as an icon, by default. When a connection is made to the Sun MTP region, the `netname` under which the printer was installed is displayed beneath the icon.

Printer - C004

FIGURE 7-2   3270 Printer Icon

The printer window displays details of the print file, print command, and any error messages, as illustrated in the following figure.
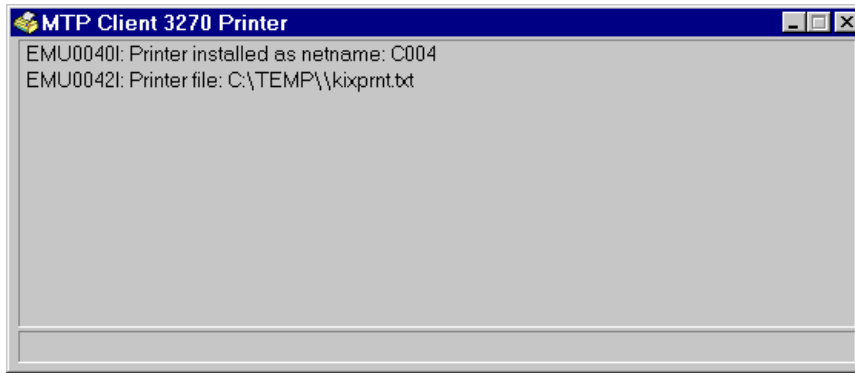
FIGURE 7-3   3270 Printer Screen

Stopping the 3270 Printer

▼ To Stop the Printer

● **Use either method:**

   ■ Select the `close` option from the system menu in the top left hand corner of the main window.

   ■ Use the icon.

External Call Interface (ECI)

This chapter describes the Sun MTP Client External Call Interface (ECI) as used with the C programming language. The C function calls and defines are usable from C++.

ECI Example Code

You can find example code illustrating the use of ECI from the C programming language in `$INSTROOT\EXAMPLES`. `$INSTROOT` indicates the name of the directory where Sun MTP Client was installed.

How Does the Sun MTP ECI Work?

The Sun MTP ECI allows applications to call Sun MTP programs in a Sun MTP server. ECI is a method of gaining access to Sun MTP programs; it does not issue `EXEC CICS` commands, but instructs Sun MTP to execute programs to do the processing. The Sun MTP program then appears to have been called by `EXEC CICS LINK` with the `COMMAREA` option.

An ECI application can call any Sun MTP program as long as the program is written to follow the rules for Distributed Program Link (DPL):

- It must not use `EXEC CICS SYNCPOINT` or `EXEC CICS SYNCPOINT ROLLBACK` commands in an extended LUW.
- For linked-to programs, specify `D` in the `APISet` field in the PPT to only execute the DPL-restricted API commands.
- It cannot issue CICS commands against the principal facility.

An application can run multiple Sun MTP programs concurrently; either on the same or different Sun MTP regions.

When considering the use of ECI, note the following points

■ A single ECI unit-of-work cannot be spanned across multiple regions. (A unit-of-work is a series of actions that must be completed before any action can be committed.)

■ A single ECI unit-of-work cannot have more than one part executing at any one time.

The complete ECI functionality comes from two API function calls:

`CICS_ExternalCall()`

Provides most of the functionality of ECI, as described in "CICS_ExternalCall Call Types" on page 40. The function takes a single parameter, which is a pointer to an `ECI_PARMS` block. It is the fields within this control structure that determine what functionality is performed. "CICS_ExternalCall()" on page 49 describes this function.

`CICS_EciListSystems()`

Provides an interface for the programmer to inquire as to which server systems are configured for use. "CICS_EciListSystems()" on page 52 describes the syntax.

CICS_ExternalCall Call Types

There are three types of calls to CICS_ExternalCall():

Program link

Causes program execution of the Sun MTP server.

Status information

Obtains connection information about specific systems.

Reply solicitation

Obtains results and status from previous program link calls.

In order to perform any of the above tasks, the `eci_call_type` field of the `ECI_PARMS` control block must be set to one of the required values. The values available are listed in TABLE 8-1.

TABLE 8-1   `eci_call_type` for Functions

Function	eci_call_type Values
Program link	`ECI_SYNC`
	`ECI_ASYNC`
	`ECI_ASYNC_NOTIFY_MSG` (Windows only)
	`ECI_ASYNC_NOTIFY_SEM` (Windows only)
Status request	`ECI_STATE_SYNC`
	`ECI_STATE_ASYNC`
	`ECI_STATE_ASYNC_MSG` (Windows only)
	`ECI_STATE_ASYNC_SEM` (Windows only)
Reply solicitation	`ECI_GET_REPLY`
	`ECI_GET_REPLY_WAIT`
	`ECI_GET_SPECIFIC_REPLY`
	`ECI_GET_SPECIFIC_REPLY_WAIT`

In addition to the categorization in TABLE 8-1, a distinction must be made between synchronous and asynchronous call types:

Synchronous calls

The function does not return control to the application until the request has completed. This means that the request traveled over the network to the Sun MTP region, the program was scheduled, completed and the response is available. This may take considerable time, especially on busy systems and networks with a narrow bandwidth. During this time, the application making the synchronous call is unable to service requests; therefore, it cannot perform other tasks.

Note – Because of their behavior, synchronous calls are not recommended.

Asynchronous calls

On Windows, the ECI is thread safe. In this case, processing could continue on another thread, if required.

The function returns to the application as soon as the request is scheduled. This means that the application gets control back before the request is given to the network. The application can perform other work and is informed asynchronously when the request has completed. The result of the request is obtained using a reply solicitation call.

Program Link Calls

Program link calls are either synchronous or asynchronous. For asynchronous calls, it is the responsibility of the calling application to solicit the reply using one of the reply solicitation calls. See "Reply Solicitation Calls" on page 42.

Program link calls either initiate a program as part of a unit-of-work or to complete a unit-of-work. The functions that a program link call can perform are as follows:

- Perform a one-shot unit-of-work
- Start a new unit-of-work
- Continue an existing unit-of-work
- Syncpoint a unit-of-work
- Roll back a unit-of-work

Reply Solicitation Calls

Reply solicitation calls get information back after asynchronous program link or asynchronous status information calls. There are two types of reply solicitation calls:

- General: Retrieving any piece of outstanding information.

- Specific: Retrieving information for a named asynchronous request

An application that uses the asynchronous method of calling may have several program link and status information calls outstanding at any time. The `eci_message_qualifier` parameter in the ECI parameter block is used on an asynchronous call to provide a user-defined identifier for the call.

Note – The use of different identifiers for different asynchronous calls within a single application is the programmer's responsibility.

When a general reply solicitation call is made, ECI uses the `eci_message_qualifier` field to return the name of the call to which the reply belongs. When a specific reply solicitation call is made, it is necessary to supply a value in the `eci_message_qualifier` field to identify the asynchronous call about which information is being sought.

Within the Windows programming environment, the use of specific reply solicitation calls is recommended because when an ECI request is made, the application can be told when the reply for that request is ready. Therefore, there is enough information to specify which request to handle.

Status Information Calls

Status information calls can be either synchronous or asynchronous. For asynchronous calls, it is the responsibility of the calling application to obtain the reply using a reply solicitation call. See "Reply Solicitation Calls" on page 42.

Status information is supplied in the ECI status block (`ECI_STATUS`), which is passed across the interface in the `eci_commarea` parameter.

The ECI status block contains the following status information

- The type of connection (whether the ECI program is locally connected to a Sun MTP server, a Sun MTP client, or nothing)
- The state of the Sun MTP server (available, unavailable, or unknown)
- The state of the Sun MTP client (available, not applicable, or unknown)

The status information calls allow you to perform these tasks

- Inquire about the type of system on which the application is running and its connection with a given server. You need to provide a `COMMAREA` in which to return the status.
- Set up a request to notify you when the status is different from the status specified. You need to provide a `COMMAREA` in which the specified status is described. You can only use asynchronous calls for this purpose.
- Cancel a request for notification of status change. No `COMMAREA` is required.

The format of the status request block is described in "ECI Data Structures" on page 47.

Application Design

Application design is different between Solaris and Windows environments because of the different paradigms that are usually followed for applications in those environments. This section describes the management of logical units-of-work, which is common to all platforms, and provides guidelines for programming in the Windows and Solaris environments.

Managing Logical Units-of-Work

An ECI application is often concerned with updating recoverable resources on the server. The application programmer must understand the facilities that ECI provides for managing logical units-of-work. A logical unit-of-work is all the processing in

the server that is needed to establish a set of updates to recoverable resources. When the logical unit-of-work ends normally, the changes are all committed. When the logical unit-of-work ends abnormally, for instance because a program abends, the changes are all backed out. You can use ECI to start and end logical units-of-work on the server.

The changes to recoverable resources in a logical unit-of-work are affected by

- A single program link call
- A sequence of program link calls. On successful return from the first of a sequence of calls for a logical unit-of-work, the eci_luw_token field in the control block contains a token that should be used for all later calls related to the same logical unit-of-work. All program link calls for the same logical unit-of-work are sent to the same server.

Caution – Be careful when extending a logical unit-of-work across multiple program link calls that may span a long time. Logical units-of-work hold locks and other Sun MTP resources on the server; this may cause delays to other users who are waiting for those same locks and resources.

When a logical unit-of-work ends, the Sun MTP server attempts to commit the changes. The last, or only, program link call of a logical unit-of-work is advised if the attempt was successful.

Only one program link call per logical unit-of-work can be outstanding at any time. (An asynchronous program link call is outstanding until a reply solicitation call has processed the reply.)

The techniques required to manipulate units-of-work from ECI are illustrated in "Common ECI Scenarios" on page 54. Using these techniques, an application can manage the control flow within the unit-of-work to perform the tasks required.

Each logical unit-of-work ties up one Sun MTP non-facility task for the duration of its execution. This means that you must define enough free tasks in the Sun MTP server to service the maximum expected number of concurrent calls. The number of tasks is affected by the TCPRTERM environment variable described in "Configuring Sun MTP for a TCP/IP Connection to Sun MTP Client" on page 15 and by the number of transaction servers created when Sun MTP is started. For example, if you need six concurrent units-of-work to the same Sun MTP region, set $TCPRTERM to at least six; the number of transaction servers must also be set to at least six in the VSAM Configuration Table (VCT).

An ECI application is not restricted to running a single unit-of-work at any one time. An application can perform many simultaneous units-of-work directed at one or many Sun MTP regions. These units-of-work are independent of each other, and can be controlled by the application so that they work in harmony. For example, an application may need to obtain data from two different regions for display on the

screen. Obtaining the data would be done with an ECI call to each of the regions, both running simultaneously but within separate units-of-work. The data is then received by the application and displayed to the end user as required.

As described in "How Does the Sun MTP ECI Work?" on page 39 and "CICS_ExternalCall Call Types" on page 40, ECI provides a series of methods for performing the same user functionality.

Designing an Application for Windows

This section describes programming methods that work best in a Windows environment.

Since Windows is a multi-threaded environment, it is possible, and in some cases desirable, to design applications using synchronous ECI calls. When these calls are executed from multiple threads, it is possible to have multiple synchronous units-of-work active at any one time. However, the creation of multi-threaded applications is problematic since it normally requires a lot of work to synchronize multiple threads.

It is easier, and more effective to use the notification facilities available on asynchronous calls within the Windows environment. This enables the application to be written as a normal Windows application and to have notification of completion of DPLs done via a variety of mechanisms. Thus, ECI can easily be fitted into the design of the rest of the application.

The most important of the notification mechanisms is the Windows message. When using this mechanism, the application is notified that work has completed by posting a Windows message. The message parameters informs the application what has happened to each ECI request. The application then uses the reply solicitation calls to obtain the completed results of the work. This message posting method fits normal Windows programming modes where an application is mainly concerned with the presentation to the user. The message posted to the application concerning ECI completion is the same as all other Windows messages.

There are two other notification mechanisms the semaphore and the callback. Semaphore notification is a misnomer because the notification is actually via a Windows Event.

An application creates an Event, and when an ECI request completes, ECI signals the Event. Then the application can use the `WaitForSingleObject()` and `WaitForMultipleObjects()` system calls to wait for the completion of the request.

The callback notification mechanism is less useful but may be what an application requires. Using this mechanism, a user-defined function is executed when an ECI request completes. The callback routine should not perform large amounts of work.

An application must not call ECI in that callback, so the callback normally just notifies the rest of the application that there is work to be done to receive the reply. On Windows, this callback is made on a separate thread that is owned by ECI.

ECI is not used to notify the application about the completion of the call. Here, it is the responsibility of the application to solicit the reply when it is appropriate. If there is no reply ready, the relevant return code is returned from the ECI solicitation call.

Designing an Application for Solaris

This section describes programming methods that work best in a Solaris environment.

There are two choices for application design:

■ Synchronous mode

  The application blocks during the processing of a DPL. In many cases, this is satisfactory.

■ Asynchronous mode

  Some applications require many units-of-work active at any one time. On the Solaris implementation, there is only one method of asynchronous request notification, the callback. However, there are several design issues with this callback mechanism.

Sun MTP Client communicates with an application via named pipes. Since Sun MTP Client is single threaded, data can only be read from the pipe when an application calls into the ECI/EPI interface. It is only at this point that any callbacks can be performed. Thus, notification callbacks are only performed while processing other ECI/EPI functions. These limitations can cause problems when designing an application that works effectively. To simplify this process, use the KixCli_QueryFD() function, which returns a standard UNIX file descriptor that is the pipe used for communications. An application can use the select() function call to wait until there is information for the ECI/EPI to process. When the application is told that there is data, it should call into the ECI/EPI APIs to ensure any callbacks are performed.

The supplied sample ECIEX2 shows how this mechanism can be fitted into a curses application that performs multiple concurrent units-of-work, while simultaneously servicing user input.

ECI Data Structures

The tables in this section define the following ECI data structures:

- `ECI_STATUS`, TABLE 8-2
- `ECI_PARMS`, TABLE 8-3

TABLE 8-2 `ECI_STATUS` Structure Fields

Field	Value	Description
ConnectionType	`ECI_CONNECTED_NOWHERE`	Should never use.
	`ECI_CONNECTED_TO_CLIENT`	Application is running on a Sun MTP Client system.
	`ECI_CONNECTED_TO_SERVER`	Should never use.
CicsServerStatus	`ECI_SERVERSTATE_UNKNOWN`	Should never use.
	`ECI_SERVERSTATE_UP`	Sun MTP server is currently connected and available for work.
	`ECI_SERVERSTATE_DOWN`	Sun MTP server is not available for work.
CicsClientStatus	`ECI_CLIENTSTATE_UNKNOWN`	Should never use.
	`ECI_CLIENTSTATE_UP`	ECI Client is available.
	`ECI_CLIENTSTATE_INAPPLICABLE`	Should never use.

TABLE 8-3 `ECI_PARMS` Structure Fields

Field	Type	Description
eci_call_type	sshort	The primary call type that is modified by the `eci_extend_mode` field, which together define the call being requested.
eci_program_name	char[8]	Name of the program to execute on the Sun MTP server.
eci_userid	char[8]	User ID for server security checking.
eci_password	char[8]	Password for server security checking.
eci_transid	char[4]	The transaction name under which the program will run, as available to the Sun MTP program via the `EIBTRNID` field.

TABLE 8-3   ECI_PARMS Structure Fields *(Continued)*

Field	Type	Description
eci_abend_code	char[4]	The abend code that is returned from a failed program. This can also be set for some ECI Client side errors.
eci_commarea	char *	A pointer to the COMMAREA. Data is either sent from, or placed into this area (or both), depending on the call type.
eci_commarea_length	sshort	The length of the eci_commarea block.
eci_timeout	sshort	A timeout in seconds for a request. This is normally set to 0 to indicate no timeout; only set to other values in rare instances.
eci_sys_return_code	sshort	A return code that provides extended information about a failure. This field is never set by the ECI Client.
eci_extend_mode	sshort	A modifier to the eci_call_type field. Together with eci_call_type, this field fully defines the functionality required.
eci_window_handle	HWND	Window handle of a window that receives posted messages upon completion of asynchronous calls. (Windows Only)
eci_hinstance	HINSTANCE	Ignored by the ECI Client.
eci_message_id	ushort	ID of the message that is sent to the eci_window_handle window upon completion of asynchronous calls.
eci_message_qualifier	sshort	User-defined value that is used to identify different asynchronous calls.
eci_luw_token	slong	A token returned by some ECI calls and used as input by others, to identify the unit-of-work that is being acted on.
eci_sysid	char[4]	Ignored by the ECI Client.
eci_version	sshort	The version of ECI in use. For new applications, set to ECI_UNIKIX_VERSION_1. However, Sun MTP Client also supports ECI_VERSION_1 for compatibility with the IBM Client ECI.
eci_system_name	char[8]	Name of the Sun MTP server, which are any of the values specified in the [Systems] section of the INI file (obtained with the CICS_EciListSystems() call). Alternatively, you can use the default system by specifying NULLs in this parameter. Note that this parameter is ignored on some calls.
eci_callback	func*	A pointer to a callback routine that may be called on completion of an asynchronous request.

TABLE 8-3   ECI_PARMS Structure Fields *(Continued)*

Field	Type	Description
eci_userid2	char[16]	Ignored.
eci_password2	char[16]	Ignored.
eci_tpn	char[4]	The transaction ID under which the program actually runs instead of CPMI. This value is ignored for ECI_VERSION_1 and ECI_UNIKIX_VERSION_1.

ECI Functions

This section lists and describes the syntax of the two ECI functions.

CICS_ExternalCall()

CICS_ExternalCall() gives access to the program link calls, status information calls, and reply solicitation calls described in the previous section. The eci_call_type field in the ECI parameter block controls the function performed.

Format

cics_sshort_t CICS_ExternalCall(ECI_PARMS *EciParms);

where EciParms is a pointer to an ECI parameter block. Different fields within this block are required for the different call types. Later sections describe those required for the normal usage of the CICS_ExternalCall().

Before calling this function, set the entire block to NULL before setting the relevant fields. See TABLE 8-3 for a description of the ECI_PARMS structure fields.

"Common ECI Scenarios" on page 54 contains examples of the CICS_ExternalCall() and describes which calls and control block values are needed for a particular use.

Return Codes

ECI_NO_ERROR

The call to CICS_ExternalCall() was successful.

ECI_ERR_INVALID_DATA_LENGTH

The eci_commarea length is invalid and is caused by one of the following

eci_commarea length < 0

    `eci_commarea` length > 32500

    `eci_commarea` length = 0 and

    `eci_commarea` is not a NULL pointer

    `eci_commarea` length > 0 and

    `eci_commarea` is a NULL pointer

`ECI_ERR_INVALID_EXTEND_MODE`

The value of `eci_extend_mode` is either invalid, or not appropriate to the value of `eci_call_type`.

`ECI_ERR_NO_CICS`

Either the ECI Client is unavailable, or the server is not available.

`ECI_ERR_CICS_DIED`

The Sun MTP Server that is servicing an extended unit-of-work is no longer available. When this occurs, it is not possible to determine if the resource changes were committed or backed out. Log this event to aid in manual recovery of the resources involved.

`ECI_ERR_NO_REPLY`

A reply solicitation has asked for a reply when none is available.

`ECI_ERR_RESPONSE_TIMEOUT`

The request has timed out before completion. When this occurs, it is not possible to determine if resource changes were committed or backed out. Log this event to aid in manual recovery of the resources involved.

`ECI_ERR_TRANSACTION_ABEND`

The request to which this call refers has caused the Sun MTP transaction to abend. The abend code is returned in the `eci_abend_code` field of the `ECI_PARMS` structure.

`ECI_ERR_LUW_TOKEN`

The `eci_luw_token` field contains an invalid value.

`ECI_ERR_SYSTEM_ERROR`

An internal error has occurred.

`ECI_ERR_NULL_WIN_HANDLE`

The `eci_window_handle` field specified does not refer to an existing window.

`ECI_ERR_NULL_SEM_HANDLE`

The `eci_sem_handle` field specified is NULL.

`ECI_ERR_NULL_MESSAGE_ID`

The `eci_message_id` field is 0.

ECI_ERR_INVALID_CALL_TYPE

The `eci_call_type` field contains an invalid value.

ECI_ERR_ALREADY_ACTIVE

An attempt was made to continue a unit-of-work while an existing request for that unit-of-work was active.

ECI_ERR_RESOURCE_SHORTAGE

There were insufficient system resources to satisfy the request.

ECI_ERR_NO_SESSIONS

The application attempted to create a new unit-of-work when the ECI Client had reached its maximum allowed number of concurrent units-of-work.

ECI_ERR_INVALID_DATA_AREA

Either the pointer to the `ECI_PARMS` structure is NULL, or the `eci_commarea` pointer is NULL when it is not expected to be NULL.

ECI_ERR_INVALID_VERSION

The `eci_version` field is not one of `ECI_UNIKIX_VERSION_1`, `ECI_UNIKIX_VERSION_1A`, `ECI_VERSION_1` or `ECI_VERSION_1A`.

ECI_ERR_UNKNOWN_SERVER

The `eci_system_name` field contains an unknown system.

ECI_ERR_CALL_FROM_CALLBACK

A call to `CICS_ExternalCall()` was made from inside a callback routine. This is not allowed.

ECI_ERR_INVALID_TRANSID

The value in `eci_transid` is different from the one specified in previous calls in the same unit-of-work.

ECI_ERR_SECURITY_ERROR

The `eci_userid`/`eci_password` pair that was supplied were invalid.

ECI_ERR_MAX_SYSTEMS

The number of systems that the ECI Client can talk to concurrently is reached.

ECI_ERR_ROLLEDBACK

When attempting to commit a unit-of-work, the Sun MTP region was unable to do so, and rolled the changes back.

CICS_EciListSystems()

The `CICS_EciListSystems()` function enables the programmer to obtain a list of remote systems at which ECI requests can be directed. The list of systems returned is the set of systems defined in the `KIXCLI.INI` file (see "Adding Systems to the `KIXCLI.INI` File" on page 10). A system appearing in the list does not mean that a connection exists to the remote system; it only indicates that the system is defined in the `KIXCLI.INI` file.

On successful return from this function, the array pointed to by the List parameter contains details of the systems; the `Systems` parameter is updated to reflect the number of systems that are available.

The systems returned by this function allow an application to provide a valid `Systems` parameter to the `CICS_ExternalCall()` function. The only valid values for the `Systems` parameter are those returned by this call and a special value consisting of eight spaces.

Format

```
cics_sshort_t CICS_EciListSystems(cics_char_t *NameSpace,
cics_ushort_t *Systems,
CICS_EciSystem_t *List);
```

where:

`NameSpace`	Must be set to a NULL pointer; it is ignored.
`Systems`	Pointer to a number. On entry, it contains the number of `CICS_EciSystem_t` structures that can fit into the area of storage pointed to by the List parameter. On return, it contains the number of systems that exist, regardless of how many were requested.
`List`	Pointer to an area of storage that contains details of the systems defined. It only contains the information found in the `Systems` area.

The following table describes the fields within the `CICS_EciSystem_t` structure.

TABLE 8-4   `CICS_EciSystem_t` Structure Fields

Field	Type	Description
`SystemName`	`char[9]`	Name of the remote system, which is a NULL- terminated character string obtained from the `KIXCLI.INI` file. Also used in the `CICS_ExternalCall()` function to perform remote requests.
`Description`	`char[61]`	Textual description of the system obtained from the `KIXCLI.INI` file.

Return Codes

ECI_NO_ERROR

The call to `CICS_EciListSystems()` was successful.

ECI_ERR_INVALID_DATA_LENGTH

The value of `Systems` is such that the amount of storage in the `List` parameter exceeds 32767 bytes.

ECI_ERR_NO_CICS

The ECI Client is unavailable.

ECI_ERR_SYSTEM_ERROR

An internal error has occurred.

ECI_ERR_INVALID_DATA_AREA

The `List` parameter is NULL, and the `Systems` parameter is non-zero.

ECI_ERR_CALL_FROM_CALLBACK

A call to `CICS_EciListSystems()` was made from inside a callback routine. This is not allowed.

ECI_ERR_NO_SYSTEMS

The number of systems defined in the `KIXCLI.INI` file is 0.

ECI_ERR_MORE_SYSTEMS

The number of systems in the `KIXCLI.INI` file is greater than the number requested in the `List` parameter. In this case, the value in the `List` parameter is the number of systems that exist.

KixCli_QueryFD()

This function is only available on Solaris and enables the programmer to determine the file descriptor (FD) that is being used for communications between the application process and the Client process. It is an extension to enable the writing of efficient ECI and EPI applications.

When asynchronous API calls are made, there is no way for the application to know when a reply is available. The application process is single threaded, so the only way that the ECI/EPI APIs can know when data is ready is for the application is to make calls into the APIs. In addition, the application does not know when data is ready to be received.

The `KixCli_QueryFD()` function allows an application to obtain the FD that is used for communications. Giving this to the application allows it to use the system `select()` function call to determine when there is work for the ECI/EPI APIs to

perform. After an application finds that there is incoming data on the FD, it should make a call into the ECI/EPI APIs to enable the data to be read and processed. The type of call does not matter; the purpose of the call is to allow the data to be read and any callbacks to occur. For example, the application could call `KixCli_QueryFD()` again, which has no real effect, but allows the callbacks to occur.

The supplied samples, `ECIEX2` and `EPIEX2`, demonstrate the use of this call to enable the processing of curses I/O while concurrently performing ECI and EPI requests.

Format

```
cics_sshort_t KixCli_QueryFD( int *pFD );
```

where:

> pFD
>
> > Pointer to an FD that is returned by this call.

Return Codes

> ECI_NO_ERROR
>
> > The call to `KixCli_QueryFD()` was successful.
>
> ECI_ERR_FROM_CALLBACK
>
> > The call to `KixCli_QueryFD()` was made while processing a callback.
>
> ECI_ERR_NO_CICS
>
> > Could not contact the Sun MTP Client.

Common ECI Scenarios

This section describes the sample applications supplied with Sun MTP Client and demonstrate some methods for using ECI. Only those relevant to your platform are provided.

ECI provides a variety of mechanisms to create and use units-of-work. The first thing you must do when designing an application is to decide on the requirements for reply notification.

Note – Since some methods only apply to certain platforms, read the guidelines in "Application Design" on page 43 before using this section.

When deciding which of these scenarios most closely matches your requirements, consider the programs that you are going to execute on the Sun MTP server.

- Is there a single program to run?
- Do you need to control whether the program commits the work, or whether it backs it out?
- Do you need to run many programs to form a logical unit-of-work?

Answering these questions should enable you to chose the appropriate programming model.

Performing a One-Shot DPL

The most frequently required functionality available from ECI is to perform a unit-of-work that consists of a single DPL to a Sun MTP region. It is normally used for a single set of updates or a single set of queries directed at a single region. Many applications require this functionality.

Performing a One-shot Asynchronous DPL Using Message Notification

Note – This is only available on Windows platforms.

Performing an asynchronous DPL consists of three actions:

1. The application must first call CICS_ExternalCall(), telling it to perform the work, and informing it how to notify the application program of any reply.

2. It must wait until it receives notification that the request is complete.

3. It must call CICS_ExternalCall() again to perform a reply solicitation call to obtain the results of the request.

The first part of the process gets ECI to perform the request. This is done with the `CICS_ExternalCall()` function with the `ECI_PARMS` block set with the values listed in TABLE 8-5. Set all other parameters to NULL.

TABLE 8-5 `ECI_PARMS` Values for One-Shot Asynchronous DPL Using Message Notification

Field	Value
eci_call_type	ECI_ASYNC_NOTIFY_MSG
eci_program_name	Name of the program to execute as specified in the PPT of the Sun MTP region.
eci_userid	User ID under which to execute the program.
eci_password	Password corresponding to the `eci_userid`.
eci_commarea	Pointer to the space allocated within the application for the data to pass to the Sun MTP region.
eci_commarea_length	Length of the data block pointed to by the `eci_commarea` field.
eci_extend_mode	ECI_NO_EXTEND
eci_message_qualifier	User-defined value that you can specify so that, upon receipt of notification messages, the application can determine to which unit-of-work the notification corresponds.
eci_version	UNIKIX_ECI_VERSION_1
eci_system_name	Name of the system that is the destination for this request. This name is either one of the names returned from the `CICS_EciListSystems()` call, or set it to all NULLs, in which case, the default server is used (as defined in `KIXCLI.INI`).
eci_window_handle	Window handle of a window that receives posted messages upon completion of this request.
eci_message_id	ID of the message that is sent to the `eci_window_handle` window upon completion of this request.

After this call completes, the ECI Client fills the `eci_luw_token` field in the `ECI_PARMS` structure. This field contains an ECI Client LUW token, which serves no purpose for this type of request.

The `eci_call_type` value specified in TABLE 8-5 means that a message with the identifier `eci_message_id` is posted to the window specified in the `eci_window_handle` field.

Format

wParam	Return code of the asynchronous call.
lParam	Low order 16 bits contain the `eci_message_qualifier` specified on the asynchronous call.

On receipt of this message, the application should perform the ECI request to obtain the results of the DPL. This is done by another call to CICS_ExternalCall(), but this time with the parameter block set with the values shown in TABLE 8-6. Set all other fields to NULL.

TABLE 8-6   ECI_PARMS Value for Obtaining a Specific Reply

Field	Value
eci_call_type	ECI_GET_SPECIFIC_REPLY
eci_commarea	Pointer to space allocated within the application for the data that the Sun MTP region returns. This data must be the same length as the original data sent on the DPL.
eci_commarea_length	Length of the data block pointed to by the eci_commarea field.
eci_message_qualifier	Message qualifier that was specified on the original DPL call.
eci_version	ECI_UNIKIX _VERSION_1

After this call completes, the comm area specified in the eci_commarea field contains the returned data. The unit-of-work, as far as the Sun MTP server is concerned, is complete; any changes are committed.

This call may have modified the following field:

eci_abend_code   If the transaction running this unit-of-work abended, the abend code is placed in this field.

After this reply is received, the unit-of-work is complete.

Performing a One-Shot Asynchronous DPL Using Semaphore Notification

Note – This is only available on Windows platforms.

Performing an asynchronous DPL consists of three actions:

1. The first part of the process gets ECI to perform the request. This is done with the CICS_ExternalCall() function with the ECI_PARMS block set up with the values listed in TABLE 8-7. Set all other parameters to NULL.

TABLE 8-7   `ECI_PARMS` Values for One-Shot Asynchronous DPL Using Semaphore Notification

Field	Description
`eci_call_type`	`ECI_ASYNC_NOTIFY_SEM`
`eci_program_name`	Name of the program to execute as specified in the PPT of the Sun MTP region.
`eci_userid`	User ID under which to execute the program.
`eci_password`	Password corresponding to the `eci_userid`.
`eci_commarea`	Pointer to space allocated within the application for the data to pass to the Sun MTP region.
`eci_commarea_length`	Length of the data block pointed to by the `eci_commarea` field.
`eci_extend_mode`	`ECI_NO_EXTEND`
`eci_message_qualifi er`	User-defined value that you can specify so that, upon receipt of notification messages, the application can determine to which unit-of-work the notification corresponds.
`eci_version`	`UNIKIX_ECI_VERSION_1`
`eci_system_name`	Name of the system that is the destination for this request. This name is either one of the names returned from the `CICS_EciListSystems()` call, or set it to all NULLs, in which case, the default server is used (as defined in `KIXCLI.INI`).
`eci_sem_handle`	Handle of a system Event that is set when the request is complete.

After this call completes, the ECI Client fills the eci_luw_token field in the `ECI_PARMS` structure, which contains an ECI Client LUW token. This serves no purpose for this type of request.

The `eci_call_type` value specified in TABLE 8-7 means that the Event specified in the `eci_sem_handle` parameter is set when the request completes. An application can use the Windows functions `WaitForMultipleObjects()` and `WaitForSingleObject()` to wait for this to happen.

When an application receives notification of completion, it should perform the ECI request to obtain the results of the DPL. This is done by another call to `CICS_ExternalCall()`, but this time with the parameter block set with the values shown in TABLE 8-6. Set all other fields to NULL.

After this call completes, the comm area specified in the `eci_commarea` field contains the returned data. The unit-of-work, as far as the Sun MTP server is concerned, is complete, and any changes made are committed.

This call may have modified the following field:

`eci_abend_code`   If the transaction running this unit-of-work abended, then the abend code is placed in here.

After this reply is received, the unit-of-work is complete.

Performing a One-Shot Asynchronous DPL Using Callback Notification

Performing an asynchronous DPL consists of three actions:

1. The application must first call `CICS_ExternalCall()`, telling it to perform the work, and informing it how to notify the application program of any reply.

2. It must wait until it receives notification that the request is complete.

3. It must call `CICS_ExternalCall()` again to obtain the results of the request.

There are two areas of added complexity for callback notification:

- It is not possible to issue ECI calls while inside a callback. This means that all you should do inside a callback is to notify some other part of the program that there is a reply ready, and that it should obtain it.

- The context of the callback and the conditions under which it is performed for your implementation differs for the various platforms.

For Windows, the callback is called on a different thread than the rest of the application. Thus, the callback can happen at any point in the processing of the applications other threads. Therefore, it is important that any work done inside the callback is thread-safe.

For the Solaris environment, the Sun MTP Client communicates with an application via named pipes. Since the application is single-threaded, data can only be read from the pipe when the application calls into the ECI interface. It is only at this point that any callbacks can be performed. Therefore, notification callbacks are only performed while processing other ECI functions.

These limitations can cause problems when designing an application on Solaris that works effectively. In order to simplify this process, you can use the `KixCli_QueryFD()` function, which returns a standard UNIX file descriptor that is the pipe that is used for the communications. An application can use the `select()` function call to wait until there is information for the ECI to process. When the application is told that there is data, it should call into the ECI interface to get any necessary callbacks processed. The supplied samples show how you can use this method to enable `curses` applications to work cleanly with the ECI. The first part of

the process is to get ECI to perform the request. This is done with the `CICS_ExternalCall()` function with the `ECI_PARMS` block set up with the values listed in TABLE 8-8. Set all other parameters to NULL.

TABLE 8-8 `ECI_PARMS` Values for One-Shot Asynchronous DPL Using Callback Notification

Field	Value
eci_call_type	ECI_ASYNC
eci_program_name	Name of the program to execute as specified in the PPT of the Sun MTP region.
eci_userid	User ID under which to execute the program.
eci_password	Password corresponding to the `eci_userid`.
eci_commarea	Pointer to the space allocated within the application for the data to pass to the Sun MTP region.
eci_commarea_length	Length of the data block pointed to by the `eci_commarea` field.
eci_extend_mode	ECI_NO_EXTEND
eci_message_qualifier	User-defined value you can specify so that, upon receipt of notification messages, the application can determine to which unit-of-work the notification corresponds.
eci_version	UNIKIX_ECI_VERSION_1
eci_system_name	Name of the system that is the destination for this request. This name is either one of the names returned from the `CICS_EciListSystems()` call, or it set to all NULLs, in which case the default server is used (as defined in `KIXCLI.INI`).
eci_callback	Pointer to a function to call when a response is ready.

After this call completes, the ECI Client fills the `eci_luw_token` field in the `ECI_PARMS` structure, which contains an ECI Client LUW token. This serves no purpose for this type of request.

The `eci_call_type` value specified in TABLE 8-8 means that the callback specified in the `eci_callback` parameter is executed when the request completes.

When an application receives notification of completion, it should perform the ECI request to obtain the results of the DPL. This is done by another call to `CICS_ExternalCall()`, but with the parameter block set with the values shown in TABLE 8-6. Set all other fields to NULL.

After this call completes, the memory location specified in the `eci_commarea` field contains the returned data. The unit-of-work, as far as the Sun MTP server is concerned, is complete; any changes made are committed.

The following field may have been modified by this call:

eci_abend_code   If the transaction running this unit-of-work abended, then the
                        abend code is placed in here.

After this reply is received, the unit-of-work is complete.

Performing a One-Shot Synchronous DPL

Performing a synchronous DPL consists of a single action. The application must
make a single call to CICS_ExternalCall(), telling it to perform the work. The
call to CICS_ExternalCall() is done with the ECI_PARMS block set with the
values listed in TABLE 8-9. Set all other parameters to NULL.

TABLE 8-9   ECI_PARMS Values for One-Shot Synchronous DPL

Field	Value
eci_call_type	ECI_SYNC
eci_program_name	Name of the program to execute as specified in the PPT of the Sun MTP region.
eci_userid	User ID under which to execute the program.
eci_password	Password corresponding to the eci_userid.
eci_commarea	Pointer to space allocated within the application for the data to pass to the Sun MTP region.
eci_commarea_length	Length of the data block pointed to by the eci_commarea field.
eci_extend_mode	ECI_NO_EXTEND
eci_version	UNIKIX_ECI_VERSION_1
eci_system_name	Name of the system that is the destination for this request. This name is either one of the names returned from the CICS_EciListSystems() call, or it set it to all NULLs, in which case, the default server is used (as defined in KIXCLI.INI).

After this call completes, the memory location specified in the eci_commarea field
contains the returned data. The unit-of-work, as far as the Sun MTP server is
concerned, is complete, and any changes made are committed.

The call may have modified the following field:

eci_abend_code   If the transaction running this unit-of-work abended, then the
                        abend code is placed in here.

After this call returns, the unit-of-work is complete.

Starting a Multiple Part Unit-of-Work

The method for starting a multiple part unit-of-work is similar to that of starting a unit-of-work consisting of a single program. For asynchronous, the application must perform the same steps for issuing the request, waiting for notification of the results, and obtaining the results. The most significant difference is that after obtaining the results, the unit-of-work is not over, and it can be extended to the users requirements.

Therefore, to initiate a long running unit-of-work, follow the same steps as a one-shot DPL, with the exception that you must set the `eci_extend_mode` parameter on the initial call to `CICS_ExternalCall()` to `ECI_EXTEND` instead of `ECI_NO_EXTEND`. On return from the initiating call to `CICS_ExternalCall()`, the `eci_luw_token` contains a value that must be supplied to later calls. You should store this value to facilitate the completion of the unit-of -work.

After the reply is received, the unit-of-work is ready to be extended or completed.

Caution – At this point, an Sun MTP transaction server process is tied-up awaiting a reply. Therefore, it is unavailable to run other Sun MTP transactions. Do not leave a transaction processor in this state any longer than necessary.

Continuing a Long Running Unit-of-Work

To continue a long running unit-of-work, you must have initiated a unit-of-work, using methods similar to those described in "Starting a Multiple Part Unit-of-Work" on page 62. You must retain the `eci_luw_token` returned by the call to `CICS_ExternalCall()` that initiated the unit-of-work to use in these extensions to the unit-of-work.

The steps required for continuing a unit-of-work are the same as those for initiating one. A call to `CICS_ExternalCall()` must be made to run a program. These is either synchronous or asynchronous. The only significant difference between continuing a unit-of-work and initiating one is that the `eci_luw_token` field in the `ECI_PARMS` structure must contain the value that was returned on the original initiating program call. If the `eci_no_extend` of the latest DPL initiating call is set to `ECI_EXTEND`, then after the reply is received, the unit-of-work is complete, and the `eci_luw_token` that was stored away is no longer valid. If the `eci_extend_mode` is `ECI_EXTEND`, the unit-of-work is ready to have more action applied to it after the reply is received.

Explicitly Syncpointing a Unit-of-Work

To syncpoint a unit-of-work, you must have initiated a unit-of-work, using the methods described in "Starting a Multiple Part Unit-of-Work" on page 62. You must retain the `eci_luw_token` returned by the call to `CICS_ExternalCall()` that initiated the unit-of-work for use in these extensions to the unit-of-work.

The steps required for syncpointing a unit-of-work are the same as those for initiating one. You must make a call to `CICS_ExternalCall()` to syncpoint the unit-of-work, and if necessary, the application then waits until a reply is ready; then, it gets the reply. The significant difference is that the `eci_luw_token` field in the `ECI_PARMS` structure must contain the value that was returned on the original program call.

The first part of the process is to get ECI to perform the request. This is done with the `CICS_ExternalCall()` function with the `ECI_PARMS` block set up with the values given in TABLE 8-10. Set all other parameters to NULL.

TABLE 8-10   `ECI_PARMS` Values for Syncpointing a Unit-of-Work

Field	Value
`eci_call_type`	`ECI_ASYNC_NOTIFY_MSG`, `ECI_ASYNC_NOTIFY_SEM`, or `ECI_SYNC`
`eci_commarea`	NULL pointer because no program is to be run.
`eci_commarea_length`	0 because no program is to be run.
`eci_extend_mode`	`ECI_COMMIT`
`eci_luw_token`	Token returned by the initial ECI call in this unit-of-work.
`eci_message_qualifier`	User-defined value you can specify so that, upon receipt of notification messages, the application can determine to which unit-of-work the notification corresponds. This value does not have to be the same as the value for the original request.
`eci_version`	`ECI_UNIKIX_VERSION_1`
`eci_window_handle`	Window handle of a window that received posted messages upon completion of this request. (For `ECI_ASYNC_NOTIFY_MSG`.)
`eci_message_id`	ID of the message that is sent to the `eci_window_handle` window upon completion of this request. (For `ECI_ASYNC_NOTIFY_MSG`.)
`eci_sem_handle`	Handle of a system Event that is set when the request is complete. (For `ECI_ASYNC_NOTIFY_SEM`.)

After this call is complete and the reply received, if necessary, the unit-of-work, as far as the Sun MTP server is concerned, is complete, and any changes made are committed.

Rolling Back a Unit-of-Work

In order to roll back a unit-of-work, you must have initiated a unit-of-work, using methods similar to those described in "Starting a Multiple Part Unit-of-Work" on page 62. You must retain the `eci_luw_token` returned by the call to `CICS_ExternalCall()` that initiated the unit-of-work for use in these extensions to the unit-of-work.

The basic steps required for rolling back a unit-of-work are the same as those for syncpointing one. A call to `CICS_ExternalCall()` must be made to roll back the unit-of-work, the application then waits until a reply is ready, then it receives the reply.

The first part of the process is to get ECI to perform the request. This is done with the `CICS_ExternalCall()` function with the `ECI_PARMS` block set up with the values given in TABLE 8-11. Set all other parameters to NULL.

TABLE 8-11   `ECI_PARMS` Values for Rolling Back a Unit-of-Work

Field	Value
eci_call_type	ECI_ASYNC_NOTIFY_MSG
eci_commarea	NULL pointer because no program is to be run.
eci_commarea_length	0 because no program is to be run.
eci_extend_mode	ECI_BACKOUT
eci_luw_token	Token returned by the initial ECI call in this unit-of-work.
eci_message_qualifier	User-defined value you can specify so that, upon receipt of notification messages, the application can determine to which unit-of-work the notification corresponds. This value does not have to be the same as the value for the original request.
eci_version	ECI_UNIKIX_VERSION_1
eci_window_handle	Window handle of a window that receives posted messages upon completion of this request. For `ECI_ASYNC_NOTIFY_MSG`.
eci_message_id	ID of the message that is sent to the `eci_window_handle` window upon completion of this request. For `ECI_ASYNC_NOTIFY_MSG`.
eci_sem_handle	Handle of a system Event that is set when the request is complete. For `ECI_ASYNC_NOTIFY_SEM`.

After this call is complete and the reply is received, if necessary, the unit-of-work, as far as the Sun MTP server is concerned, is over; any changes made have been rolled back.

Interrogating Connections to a Remote System

Using the ECI, it is possible to make a call to interrogate the ECI Client as to whether there is a connection to a specified Sun MTP region. To do this, the application must call CICS_ExternalCall() with the parameters set as shown in TABLE 8-12. Set all other parameters to NULL.

TABLE 8-12   ECI_PARMS Values for ECI_STATE_ASYNC Call

Field	Value
eci_call_type	ECI_STATE_ASYNC_MSG
eci_commarea	Pointer to a space allocated within the application for a structure of type ECI_STATUS.
eci_commarea_length	Length of the data block pointed to by the eci_commarea field (size of ECI_STATUS).
eci_extend_mode	ECI_STATE_IMMEDIATE
eci_message_qualifier	User-defined value you can specify so that, upon receipt of notification messages, the application can determine to which unit-of-work the notification corresponds.
eci_version	ECI_UNIKIX_VERSION_1
eci_window_handle	Window handle of a window that receives posted messages upon completion of this request.
eci_message_id	ID of the message that is sent to the eci_window_handle window upon completion of this request.

After ECI has completed the request, a notification message is posted to the eci_window_handle window. On receipt of this message, the application should solicit the results of the request using a call to CICS_ExternalCall() with the values listed in TABLE 8-13 set within the ECI_PARMS structure.

TABLE 8-13   ECI_PARMS Values for STATE_ASYNC_MESSAGE Reply Solicitation

Field	Value
eci_call_type	ECI_GET_SPECIFIC_REPLY
eci_commarea	Pointer to a space allocated within the application for a structure of type ECI_STATUS.

Field	Value
eci_commarea_length	Length of the data block pointed to by the eci_commarea field; size of ECI_STATUS.
eci_message_qualifier	Message qualifier that was specified on the original call and returned as part of the notification function.
eci_version	ECI_UNIKIX_VERSION_1

The ECI_STATUS structure now contains the values. For the meanings of these fields, see TABLE 8-2.

In addition to this basic status request, it is possible to have ECI notify the application when the status of a particular system changes. or example, you can use this to alert you when a system is ready for work. Set the following parameters to issue a call to CICS_ExternalCall():

eci_call_type	ECI_STATE_ASYNC_MSG
eci_extend_mode	ECI_STATE_CHANGED

After this is done, a notification is sent when the system state changes from that specified in the eci_commarea fields.

Using Callbacks

You can call ECI to generate callbacks instead of Windows messages. "Performing a One-Shot DPL" on page 55 through "Interrogating Connections to a Remote System" on page 65 detail how to generate notification via Windows messages using the eci_window_handle parameter of the ECI_PARMS structure.

To generate callbacks instead of Windows messages, set the eci_window_handle parameter to NULL, and set the eci_callback parameter as a pointer to a callback function. After this is set up and an ECI request completes, the callback function is called. The callback function receives a single value as its parameter, the eci_message_qualifier field, as specified on the original call.

Within the callback function, the application must not call any ECI functionality. If it does, these calls return the ECI_ERR_IN_CALLBACK return code. Because of this restriction, the callback function must find some way to communicate the notification to the main section of the Windows program. This is normally done by posting a Windows message.

Sun MTP ECI Interface Enhancements

Sun MTP Client has four different versions of the ECI API. These are obtained by setting the `eci_version` field of the ECI parameter block to any one of the following:

`ECI_VERSION_1`	IBM compatible version of the ECI API Version 1.
`ECI_VERSION_1A`	IBM compatible version of the ECI API Version 1A.
`ECI_UNIKIX_VERSION_1`	Same interface as `ECI_VERSION_1` but with enhancements to the reply message formats, as described in "Reply Message Formats" on page 67.
`ECI_UNIKIX_VERSION_1A`	Same interface as `ECI_VERSION_1A` but with enhancements to the reply message formats, as described in "Reply Message Formats" on page 67.

Reply Message Formats

When an application makes an asynchronous ECI call and has requested notification by a message, a message is delivered to the specified window as follows:

`ECI_VERSION_1` and `ECI_VERSION_1A`

wParam	The return code of the asynchronous call.
lParam	The four-character abend code of the asynchronous call. (This may be four spaces if no abend occurred).

`ECI_UNIKIX_VERSION_1` and `ECI_UNIKIX_VERSION_1A`

wParam	The return code of the asynchronous call.
lParam	The low-order 16 bits contain the `eci_message_qualifier` specified on the asynchronous call.

The message qualifier is usually more important information than the abend code. You can also obtain the abend code by performing an `ECI_GET_SPECIFIC_REPLY`, specifying the message qualifier.

External Presentation Interface (EPI)

The External Presentation Interface (EPI) for Sun MTP is an API that allows a non-CICS application program to appear to Sun MTP as one or more standard 3270 terminals. The EPI application communicates with Sun MTP as if it is a normal 3270 terminal and allows you to

- Log on terminals
- Start transactions
- Send and receive standard 3270 data streams to and from those transactions

The EPI application is responsible for the presentation of the 3270 data it receives. It may present the data to a device by emulating a 3270 terminal or by any other means appropriate to the user.

This chapter describes EPI as C program function specifications and the data structures they require.

EPI Examples

Examples that illustrate how to use EPI from the C programming language are found in the $INSTROOT\EXAMPLES directory, where $INSTROOT indicates the name of the directory where the Sun MTP Client was installed on your machine. The directory also contains a README file explaining its use and the necessary Sun MTP server COBOL source code.

Developing an EPI Application

EPI is a set of functions in a library that can be called from any program. These routines are a C language interface.

- For Windows
  - The ECI/EPI code is in `CCLAPI32.DLL`
  - Link applications with `CCLWIN32.LIB`
- For Solaris
  - The ECI/EPI code is in `libcclapi.so`
  - Link applications with `libcclapi.so`

All applications must include `cics_epi.h` to obtain the definitions and prototypes described in this manual.

Initializing and Terminating EPI

The EPI functions, `CICS_EpiInitialize()` and `CICS_EpiTerminate()`, initialize and terminate EPI, respectively.

The `CICS_EpiInitialize()` function must be called once per task to initialize the EPI interface. All other EPI calls are invalid before the initialization function is complete. Initialization is complete when the `CICS_EpiInitialize()` call returns with a good return code.

The `CICS_EpiTerminate()` function must be called when a process has completed using EPI, typically just before the task terminates. This terminate function cleanly terminates EPI.

Future compatibility between EPI versions is provided by the `CICS_EpiInitialize()` function, which requires a parameter that defines the version of EPI for which this application is coded.

Adding and Deleting EPI Terminals

After the application completes the initialization, it can install one or more EPI terminals, which appear to Sun MTP as 3270 terminals. The application must call the `CICS_EpiAddTerminal()` function once for each terminal to be added. This function returns a value that is used to identify it uniquely within the application.

To apply any EPI functions using this terminal, the application passes this `TermIndex` to the EPI functions.

When the application no longer needs a terminal, delete it by calling the `CICS_EpiDelTerminal()` function. After the deletion completes successfully, the `TermIndex` value becomes free and can be reused by EPI. However, the deletion does not complete until the application receives the `CICS_EPI_EVENT_END_TERM` event.

The delete function fails if the terminal is currently running a transaction. In this case, the function returns an error code.

To track information on a per terminal basis, the application can use a simple array indexing scheme., which uses the `TermIndex` value to maintain the corresponding terminal information.

Starting Transactions

The application can start transactions against any installed terminal. From the Sun MTP perspective, the transaction appears as if a terminal user entered a transaction on the screen and pressed an AID key.

The application calls the `CICS_EpiStartTran()` function to start a transaction. The `CICS_EpiStartTran()` parameters are the name of the transaction and the initial 3270 data associated with the transaction.

If the transaction name is not given, EPI determines the name of the required transaction from up to the first four data characters in the 3270 data following the AID sequence. This simplifies EPI applications that provide true 3270 terminal emulation. The initial 3270 data is not normally empty. It contains at least the AID byte for the 3270 key that causes the terminal input.

For information about the format of the 3270 datastream, refer to the *IBM 3270 Information Display System Data Stream Programmers Reference*.

Processing Events

A variety of events can affect an added terminal. The events are the result of actions within the remote system, not actions of the terminal user. When events occur, the EPI application is informed by EPI.

On receipt of event notification, the EPI application should make calls to the `CICS_EpiGetEvent()` function to retrieve a single event from the queue for processing. This function returns a data structure, `CICS_EpiEventData_t`, with information to indicate the event that occurred and any associated data. It also

indicates if there are more events waiting in the queue. Refer to "EPI Events" on page 82 for more information on the possible events and the information in the `CICS_EpiEventData_t` data structure.

If there are more events available for this terminal, the application should continue to call `CICS_EpiGetEvent()` until the event queue is empty. Failure to do so prevents the application from being notified of future events for the terminal.

The application should try to process events as quickly as possible. This synchronization prevents conditions in which the EPI state and the application state do not agree. When this mismatch occurs, the application receives an unexpected error return code when it tries to issue EPI functions.

Event notification varies on the supported platforms. The following sections describe these differences.

Event Notification on Windows

On Windows NT, event notification is performed through a callback mechanism. When there are events ready for a particular terminal, the callback routine specified on the `CICS_EpiAddTerminal()` call is initiated. This callback is on a separate thread owned by EPI; no EPI work should be performed from this thread. Normally, the callback routine must inform the main code that there is something ready using messages and events.

Event Notification on Solaris

There are special considerations required to get the callback mechanism to work properly on Solaris. Sun MTP Client communicates with an application via named pipes. Since Sun MTP Client is single threaded, data can only be read from the pipe when an application calls into the ECI/EPI interface. It is only at this point that it is possible to perform callbacks. Thus, event notification callbacks are only performed while processing other ECI/EPI functions. This limitation can cause problems when designing an application that works effectively.

To simplify this process, use the `KixCli_QueryFD()` function, which returns a standard UNIX file descriptor (FD) that is the pipe used for communications. An application can use the `select()` function call to wait until there is information for the ECI/EPI to process. When the application is told that there is data, it can call into the ECI/EPI to process any callbacks.

The supplied sample, `EPIEX2`, shows how to fit this mechanism into a `curses` application that performs multiple concurrent units-of-work, while simultaneously servicing user input.

Sending and Receiving Data

EPI generates either a `CICS_EPI_EVENT_SEND` or a `CICS_EPI_EVENT_CONVERSE` event when a transaction sends data to a terminal. The `CICS_EPI_EVENT_SEND` event does not require a reply from the terminal. The `CICS_EPI_EVENT_CONVERSE` event does require a reply from the terminal.

The EPI application replies by calling the `CICS_EpiReply()` function to provide the response data. Use this function to respond to the `CICS_EPI_EVENT_CONVERSE` event only; calling this function in response to any other event returns an error.

For information about the format of the 3270 datastream, refer to the *IBM 3270 Information Display System Data Stream Programmers Reference*.

EPI Constants and Data Structures

EPI include files provide constants and data structures for using EPI. The following sections describe the constants, standard data types, and data structures.

Constants

EPI defines the following constants for use in EPI applications (`#define` statements in C):

`CICS_EPI_SYSTEM_MAX`	8
`CICS_EPI_DESCRIPTION_MAX`	60
`CICS_EPI_NETNAME_MAX`	8
`CICS_EPI_TRANSID_MAX`	4
`CICS_EPI_ABEND_MAX`	4
`CICS_EPI_DEVTYPE_MAX`	16
`CICS_EPI_ERROR_MAX`	60
`CICS_EPI_TERM_INDEX_NONE`	0xFFFF

Standard Data Types

EPI defines the following data types for use in EPI applications:

`cics_char_t`	character type
`cics_sbyte_t`	signed 8-bit integer
`cics_sshort_t`	signed 16-bit integer
`cics_slong_t`	signed 32-bit integer
`cics_ubyte_t`	unsigned 8-bit integer
`cics_ushort_t`	unsigned 16-bit integer
`cics_ulong_t`	unsigned 32-bit integer
`cics_ptr_t`	general pointer type
`cics_shandle_t`	general 16-bit handle
`cics_lhandle_t`	general 32-bit handle

Data Structures

EPI defines the following data structures for use in EPI applications

CICS_EpiSystem_t

This data structure contains the name and description of a remote Sun MTP region. The `CICS_EpiListSystems()` function returns information of this type.

C Definition

```
typedef struct
{
        cics_char_t    SystemName[CICS_EPI_SYSTEM_MAX+1];
        cics_char_t    Description[CICS_EPI_DESCRIPTION_MAX+1];
} CICS_EpiSystem_t;
```

where:

SystemName   Name of an Sun MTP region. Pass this string as a parameter to the
                 `CICS_EpiAddTerminal()` function to identify the region to which the
                 terminal should attach. This string is padded with NULL bytes and is
                 terminated by an extra NULL byte.

Description   Brief description of the system. This string is padded with NULL bytes
                 and is terminated by an extra NULL byte.

CICS_EpiDetails_t

Pass a pointer to this structure to the `CICS_EpiAddTerminal()` function. When the
terminal install is complete, the structure contains details about the terminal.

C Definition

```
typedef struct
{
        cics_char_t    SystemName[CICS_EPI_SYSTEM_MAX+1];
        cics_char_t    Description[CICS_EPI_DESCRIPTION_MAX+1];
        cics_char_t    NetName[CICS_EPI_NETNAME_MAX+1];
        cics_sshort_t  NumLines;
        cics_sshort_t  NumColumns;
        cics_ushort_t  MaxData;
        cics_sshort_t  ErrLastLine;
        cics_sshort_t  ErrIntensify;
        cics_sshort_t  ErrColor;
        cics_sshort_t  ErrHilight;
        cics_sshort_t  Hilight;
        cics_sshort_t  Color;
        cics_sshort_t  Printer;
} CICS_EpiDetails_t;
```

where:

SystemName	Name of a Sun MTP region. Pass this string as a parameter to the `CICS_EpiAddTerminal()` function to identify the region to which the terminal should attach. This string is padded with NULL bytes and is terminated by an extra NULL byte.
Description	Brief description of the system. This string is padded with NULL bytes and is terminated by an extra NULL byte.
NetName	An 8-character string that specifies the VTAM-style `NetName` by which the terminal gets installed. This string is padded with NULL bytes and is terminated by an extra NULL byte.
NumLines	Contains the number of lines supported by the terminal.
NumColumns	Contains the number of columns supported by the terminal.
MaxData	Contains the maximum possible size of the data Sun MTP sends to the terminal.
ErrLastLine	Non-zero if the terminal displays error messages on its last line.
ErrIntensify	Non-zero if the terminal displays intensified error messages.
ErrColor	Contains the 3270 attribute defining the color value for displaying error messages.
ErrHilight	Contains the 3270 attribute defining the highlight value for displaying error messages.
Hilight	Non-zero if the terminal supports extended highlighting.
Color	Non-zero if the terminal supports color.
Printer	Non-zero if the terminal is a printer.

CICS_EpiEventData_t

Pass a pointer to this structure (with the `Data` and `Size` fields set) to the `CICS_EpiGetEvent()` function. If the function returns successfully, it returns this structure with details about the event. Not all fields are valid for all events; any invalid fields are set to zero or NULL bytes.

C Definition

```
typedef struct
{
      u_short_t           TermIndex;
      CICS_EpiEvent_t     Event;
      CICS_EpiEnd_t       EndReason;
      char                TransId[CICS_EPI_TRANSID_MAX+1];
      char                AbendCode[CICS_EPI_ABEND_MAX+1];
      u_byte_t *          Data;
      u_short_t           Size;
} CICS_EpiEventData_t;
```

where:

TermIndex	Index number for the terminal against which this event occurred.
Event	Specifies the event. See "EPI Events" on page 82 for a description of the event indicators.
EndReason	Termination reason, if the event is a CICS_EPI_EVENT_END_TERM.
TransId[]	Four-character string that specifies the transaction name. The string is padded with spaces and is terminated by a NULL byte.
AbendCode[]	Four-character string that specifies the ABEND code. The string is padded with spaces and is terminated by a NULL byte.
Data	Pointer to a buffer that contains any terminal data stream associated with the event.
Size	Contains the maximum size of the buffer addressed by Data. When the CICS_EpiGetEvent() function returns, this field contains the actual length of the data returned.

CICS_EpiSysError_t

Pass a pointer to this structure as a parameter to the CICS_EpiGetSysError() function. If the function returns successfully, it returns this structure with details about the cause of a system error.

C Definition

```
typedef struct
{
        u_long_t    Cause;
        u_long_t    Value;
        char        Msg[CICS_EPI_ERROR_MAX+1];
} CICS_EpiSysError_t;
```

where:

Cause	Operating environment specific value that indicates the cause of the last error.
Value	Operating environment specific value that indicates the nature of the last error.
Msg	Text message that may describe the last error. This string is padded with NULL bytes; EPI terminates it with an extra NULL byte. If no message is available, the string is all NULLS.

The possible values for the Cause field are

CICS_EPI_SYSERROR_UNEXPECTED_DATASTREAM

A datastream was received from the Sun MTP region that Sun MTP Client could not decode.

CICS_EPI_SYSERROR_NO_MEMORY

Sun MTP Client encountered memory allocation problems.

CICS_EPI_SYSERROR_DUPLICATE_NETNAME

The NetName specified in the add terminal command was already in use.

CICS_EPI_SYSERROR_UNKNOWN_NETNAME

NetName specified in the add terminal command was unknown to the end system.

CICS_EPI_SYSERROR_UNKNOWN_DEVTYPE

DevType specified in the add terminal command was unknown to the end system.

CICS_EPI_SYSERROR_INVALID_TPNAME

The end system did not recognize the terminal installation transaction.

CICS_EPI_SYSERROR_UNEXPECTED_ERROR

An internal function has produced an unexpected error.

CICS_EPI_SYSERROR_UNKNOWN_SYSTEM

   System specified in the add terminal command is not defined.

CICS_EPI_SYSERROR_TERMINAL_OUT_OF_SERVICE

   Could not install the terminal into the end system because its definition is set
   out of service.

CICS_EPI_SYSERROR_SYSTEM_UNAVAILABLE

   System specified in the add terminal command is unavailable at this time.

CICS_EPI_SYSERROR_INTERNAL_LOGIC_ERROR

   Internal logic error in Sun MTP Client.

CICS_EPI_SYSERROR_AUTOINSTALL_FAILED

   Autoinstall process failed on the Sun MTP region.

CICS_EPI_SYSERROR_TERM_INSTALL_FAILED

   The terminal install process failed on the Sun MTP region.

CICS_EpiNotify_t

This type is used as a parameter to the CICS_EpiAddTerminal() function. It
defines a function to be called when there is an EPI event outstanding.

C Definition

```
typedef void (CICS_EpiCallback_t)(cics_ushort_t Trm);
typedef CICS_EpiCallback_t *CICS_EpiNotify_t;
```

CICS_EpiEvent_t

This type defines the event codes that EPI generates. For additional information
about each event code, see "EPI Events" on page 82.

C Definition

```
typedef cics_ushort_t CICS_EpiEvent_t;
```

Values

   CICS_EPI_EVENT_SEND

   CICS_EPI_EVENT_CONVERSE

```
CICS_EPI_EVENT_END_TRAN

CICS_EPI_EVENT_START_ATI

CICS_EPI_EVENT_END_TERM
```

CICS_EpiEnd_t

This type defines the possible reason codes for a `CICS_EPI_EVENT_END_TERM`
event. For more information about each reason code, see
"`CICS_EPI_EVENT_END_TERM`" on page 84.

C Definition

```
typedef cics_ushort_t CICS_EpiEnd_t;
```

Values

```
CICS_EPI_END_SIGNOFF

CICS_EPI_END_SHUTDOWN

CICS_EPI_END_OUTSERVICE

CICS_EPI_END_UNKNOWN

CICS_EPI_END_FAILED
```

CICS_EpiATIState_t

This type defines the possible values to pass to the `CICS_EpiATIState()` function.
This function also returns these values when the function completes.

C Definition

```
typedef cics_ushort_t CICS_EpiATIState_t;
```

Values

```
CICS_EPI_ATI_ON
```

 EPI processes Asynchronous Transaction Initiation (ATI) requests in the normal
 manner.

```
CICS_EPI_ATI_HOLD
```

 EPI queues all ATI requests until `CICS_EPI_ATI_ON` is set.

CICS_EPI_ATI_QUERY

The ATI request status of EPI is unchanged. Use this value to query the current setting.

CICS_EpiSenseCode_t

This type defines the possible values to pass to the `CICS_EpiSenseCode()` function. The `CICS_EpiSenseCode()` function performs no function, and is provided for compatibility only.

C Definition

```
typedef cics_ushort_t CICS_EpiSenseCode_t;
```

Values

CICS_EPI_SENSE_OPCHECK

EPI detects an error in the 3270 datastream.

CICS_EPI_SENSE_REJECT

EPI detects an invalid 3270 command.

CICS_EpiWait_t

This type defines the possible values to pass to the `CICS_EpiGetEvent()` function. The value has no meaning but must be one of the defined values.

C Definition

```
typedef cics_ushort_t CICS_EpiWait_t;
```

Values

CICS_EPI_WAIT

CICS_EPI_NOWAIT

EPI Events

The EPI application is responsible for collecting and processing all EPI events. When an event occurs against a terminal, EPI informs the application that there is an event outstanding.

When event notification is received, the EPI application calls `CICS_EpiGetEvent()` to retrieve the `CICS_EpiEventData_t` structure. This structure indicates the event that occurred and contains details about the event.

The following are the possible EPI events:

CICS_EPI_EVENT_SEND

A Sun MTP transaction sent some 3270 data to a terminal and is not expecting a reply. However, the data should display. This is typically the result of an `EXEC CICS SEND` type command.

The following fields are complete in the `CICS_EpiEventData_t` structure for this event:

Event	CICS_EPI_EVENT_SEND
Data	Buffer pointed to by this field that contains the data sent by the transaction. The first two bytes are the 3270 command and the WCC (Write Control Character).
Size	Length of the data in the Data buffer.

CICS_EPI_EVENT_CONVERSE

A Sun MTP transaction sent some 3270 data to a terminal and is expecting a reply. This is the result of an `EXEC CICS RECEIVE` type command or an `EXEC CICS CONVERSE` type command. Here, the application uses the `CICS_EpiReply()` call to return the reply data to Sun MTP.

The type of reply expected by Sun MTP depends on the 3270 command order in the first byte of the supplied `Data` field. If the 3270 command order is a `Read Buffer` command, Sun MTP expects the reply immediately. If the command is a `Read Modified` or `Read Modified All`, Sun MTP expects the reply when the user next presses an AID key. This event may occur with or without any associated data. If there is no data, the `Size` field is set to zero.

The following fields are completed in the `CICS_EpiEventData_t` structure for this event:

Event	CICS_EPI_EVENT_CONVERSE
Data	Buffer pointed to by this field that contains the data sent by the transaction.
Size	Length of the data in the `Data` buffer. If zero, it indicates that no data was sent, but a reply is expected.

CICS_EPI_EVENT_END_TRAN

A Sun MTP transaction against a terminal is ended. If the transaction ends abnormally, the event may indicate the `AbendCode`. If the transaction finishes normally, the `AbendCode` field is four spaces. If the transaction is pseudo-conversational, the `TransId` field contains the name of the next transaction required. The application should start this transaction when the user next presses an AID key (use the `CICS_EpiStartTran()` function).

An application may only begin a transaction if no other transaction is running. The `CICS_EPI_EVENT_END_TRAN` event signals that EPI is in a state to accept a new transaction.

The following fields are in the `CICS_EpiEventData_t` structure for this event:

Event	CICS_EPI_EVENT_END_TRAN
TransId	If the transaction is pseudo-conversational, this is the name of the next transaction to start. The name is four characters and is terminated with an extra NULL byte. If there is no next transaction, the field is all NULL bytes.
AbendCode	If the transaction ends abnormally, this field may contain the AbendCode, which is four characters terminated with an extra NULL byte. If the transaction finishes successfully, it contains four spaces and a NULL byte.

CICS_EPI_EVENT_START_ATI

An Asynchronous Transaction Initiation (ATI) transaction is started against this terminal. If the terminal receives an ATI request while it is running another transaction, the request is held by EPI until the end of the current transaction. When the current transaction ends, the ATI transaction starts for the terminal and EPI generates this event to inform the application.

The CICS_EPI_EVENT_START_ATI event indicates an ATI transaction is started and the EPI state no longer allows transactions to start. If the application calls the CICS_EpiStartTran() function after EPI generates the CICS_EPI_EVENT_START_ATI event but before it is received, the start transaction function fails.

The following fields are in the `CICS_EpiEventData_t` structure for this event:

Event	CICS_EPI_EVENT_START_ATI
TransId	Name of the ATI transaction started. It is four characters and terminated with an extra NULL byte.

CICS_EPI_EVENT_END_TERM

This event indicates that a terminal no longer exists. The TermIndex for this terminal is no longer valid after this event is retrieved.

The following fields are in the `CICS_EpiEventData_t` structure for this event:

Event

   CICS_EPI_EVENT_END_TERM

EndReason

Contains one of the following reasons why the terminal was ended or no longer exists

`CICS_EPI_END_SIGNOFF`

Terminal is signed-off; possibly the result of calling `CICS_EpiDelTerminal()`.

`CICS_EPI_END_SHUTDOWN`

Sun MTP is shutting down.

`CICS_EPI_END_OUTSERVICE`

Terminal is switched to out-of-service.

`CICS_EPI_END_UNKNOWN`

Unexpected error occurred.

`CICS_EPI_END_FAILED`

Failure to delete a terminal occurred.

EPI Functions

The EPI library includes the following functions that you can call from a C program:

CICS_EpiInitialize()

The `CICS_EpiInitialize` function initializes EPI. You should call this function once per process; any other EPI calls before this function are invalid.

Format

```
cics_sshort_t CICS_EpiInitialize(cics_ulong_t Version);
```

where:

Version

   Version of the EPI library for which the application is coded. This provides future compatibility with EPI library versions.

   For this version of EPI, set this parameter to the constant `CICS_EPI_VERSION_101`.

Return Codes

CICS_EPI_NORMAL

   Successful completion.

CICS_EPI_ERR_FAILED

   EPI is unable to initialize the interface.

CICS_EPI_ERR_VERSION

   EPI cannot support the version requested.

CICS_EPI_ERR_IS_INIT

   EPI is already initialized for this process.

CICS_EpiTerminate()

This function terminates EPI. Call this function once per process, usually just before the process terminates. Any EPI calls made after this function are invalid. Calling this function does not delete any EPI terminals defined by the application. The application must issue the `CICS_EpiDelTerminal()` calls before terminating the interface.

Format

```
cics_sshort_t CICS_EpiTerminate(void);
```

Return Codes

CICS_EPI_NORMAL

Successful completion.

CICS_EPI_ERR_NOT_INIT

Initialization is not completed.

CICS_EPI_ERR_FAILED

Unable to terminate EPI.

CICS_EpiListSystems()

This function obtains a list of possible candidate systems (Sun MTP regions) to which a terminal can be attached.

Format

```
cics_sshort_t CICS_EpiListSystems  (cics_char_t    *NameSpace,
cics_ushort_t                                       *Systems,
CICS_EpiSystem_t                                    *List);
```

where:

Namespace	parameter is ignored but should be set to a NULL pointer.
Systems	Set to the maximum number of CICS_EpiSystem_t structures that EPI may return. When the function returns, this field contains the actual number of systems found.
List	Array of CICS_EpiSystem_t data structures. The function fills in these structures and returns them to the application. The caller provides the storage for the array and sets the Systems parameter to indicate the maximum size of the array.

Return Codes

CICS_EPI_NORMAL

Successful completion.

CICS_EPI_ERR_NOT_INIT

Initialization is not completed; call CICS_EpiInitialize().

CICS_EPI_ERR_FAILED

Unable to find candidate systems.

CICS_EPI_ERR_NO_SYSTEMS

> EPI fails to find a candidate system. The return value in the `Systems` field is zero.

CICS_EPI_ERR_MORE_SYSTEMS

> Not enough space in the array to store all the candidate systems. The actual number of systems is found in the `Systems` field. Use this to reallocate enough storage for all the candidate systems and retry the function.

CICS_EpiAddTerminal()

This function installs a new terminal. It returns a `TermIndex` value to use on all further terminal-specific EPI calls. The index number is the next available small integer, starting at zero. The application must maintain a mapping between index numbers and terminals to process per-terminal information.

Format

```
cics_sshort_t CICS_EpiAddTerminal (cics_char_t      *NameSpace,
                                   cics_char_t      *System,
                                   cics_char_t      *NetName,
                                   cics_char_t      *DevType,
                                   CICS_EpiNotify_t  NotifyFn,
                                   CICS_EpiDetails_t *Details,
                                   cics_ushort_t    *TermIndex);
```

where:

Namespace	Parameter is ignored but should be set to a NULL pointer.
System	Name of the system to which the terminal is to be attached.This name should match one of the systems returned from the `CICS_EpiListSystems()` function call. If a string of 0 length is specified, the default system is used as specified in the `KIXCLI.INI` configuration file.This is a pointer to a NULL terminated string.
NetName	VTAM-style net name for the terminal found in the Sun MTP TCT. This is a pointer to a string that is padded with NULL bytes and terminated with an extra NULL byte. If this parameter is set to a NULL pointer, a default is used. If this parameter is provided, it must be unique and must match one of the net names in the terminal definitions listed in the system's terminal database. The terminal is auto-installed if no `NetName` is specified.

DevType	Pointer to a string that specifies the model terminal. This string is padded with NULL bytes and terminated with an extra NULL byte. This parameter is ignored if the `NetName` parameter is not NULL.
	If this field is set to a NULL pointer, the default model IBM-3278-2 is used.
	The names for the terminal models are system specific, but for Sun MTP, the valid models are:
	IBM-3278-2, IBM-3278-2-E
	IBM-3278-4, IBM-3278-4-E
	IBM-3278-5, IBM-3278-5-E
	IBM-3287
	The `-E` suffix indicates that the terminal has extended visual attributes.
NotifyFn	Address of a routine to call when an EPI event occurs for this terminal.
Details	Pointer to a structure that EPI fills with details about the terminal that was installed.
	An application can safely use and discard this storage on receipt of the `CICS_EPI_WIN_INSTALLED` message.
TermIndex	Index number of the terminal that was installed. Use this index number on all other EPI functions that are specific to this terminal. The index generated is the first available integer starting from 0.

Return Codes

CICS_EPI_NORMAL

Successful completion.

CICS_EPI_ERR_NOT_INIT

Initialization is not completed.

CICS_EPI_ERR_FAILED

Unable to install the terminal.

CICS_EPI_ERR_SYSTEM

Name specified in the `System` parameter does not exist.

CICS_EPI_ERR_MAX_TERMS

EPI maximum number of terminals supported is reached.

The following return codes are returned by `CICS_EpiGetSysError()` if `CICS_EpiAddTerminal()` fails while performing its asynchronous work:

CICS_EPI_ERR_SYSTEM

Name specified in the `System` parameter does not exist.

CICS_EPI_ERR_MAX_TERMS

EPI maximum number of terminals supported is reached.

CICS_EpiDelTerminal()

This function deletes an installed EPI terminal. If the terminal was autoinstalled, its definition is deleted. However, the application cannot delete a terminal if the terminal is currently running a transaction. If the application calls this function while the terminal is processing a transaction, the function fails. When the current transaction finishes, the application can successfully call this function. The application should not consider the terminal completely deleted until it receives a good return code and the CICS_EPI_EVENT_END_TERM event.

Format

```
cics_sshort_t CICS_EpiDelTerminal(cics_ushort_t TermIndex);
```

where:

  TermIndex

    Index number of the terminal to delete.

Return Codes

  CICS_EPI_NORMAL

    Successful completion.

  CICS_EPI_ERR_NOT_INIT

    Initialization is not completed.

  CICS_EPI_ERR_FAILED

    Unable to delete the terminal.

  CICS_EPI_ERR_BAD_INDEX

    TermIndex parameter does not represent a valid terminal.

  CICS_EPI_ERR_TRAN_ACTIVE

    A transaction is currently running against the terminal.

CICS_EpiStartTran()

This function starts a transaction for an installed terminal. If the transaction begins, no other start requests are accepted by EPI until the CICS_EPI_EVENT_END_TRAN event is generated.

The application may get an unexpected return code of
CICS_EPI_ERR_ATI_ACTIVE. This can occur even if the application believes a
terminal is not currently running a transaction. It means that an ATI request was
started against the terminal and a CICS_EPI_EVENT_START_ATI event has been
raised, but the application has not processed this event.

Format

```
cics_sshort_t CICS_EpiStartTran (cics_ushort_t  TermIndex,
                                 cics_char_t    *TransId,
                                 cics_ubyte_t   *Data,
                                 cics_ushort_t  Size);
```

where:

TermIndex	Index number of the terminal to run the transaction.
TransId	Transaction code to run against the terminal. This field is a pointer to a string that is padded with spaces and terminated with an extra NULL byte.
	If the field is a NULL pointer, EPI attempts to extract the correct transaction code from the Data buffer. It does this by examining the first byte of the data and skipping any 3270 control codes to find the start of the user data. It copies up to the next four characters until either another 3270 control code is found or no more data is available. The result is padded with spaces to four characters.
	To start a continuing portion of a pseudo-conversational transaction, this field must contain the TransId returned when the CICS_EPI_EVENT_END_TRAN event occurred.
Data	Pointer to the initial 3270 data buffer associated with the transaction. This parameter cannot be a NULL pointer; it must contain at least the 3270 AID key that caused the terminal read.
Size	Number of bytes in the Data buffer.

Return Codes

CICS_EPI_NORMAL

Successful completion.

CICS_EPI_ERR_NOT_INIT

Initialization is not completed.

CICS_EPI_ERR_FAILED

Unable to start the transaction.

CICS_EPI_ERR_BAD_INDEX

   TermIndex parameter does not represent a valid terminal.

CICS_EPI_ERR_TTI_ACTIVE

   A TTI transaction is already active for this terminal.

CICS_EPI_ERR_ATI_ACTIVE

   An ATI transaction is active for this terminal.

CICS_EPI_ERR_NO_DATA

   No initial data is provided.

CICS_EpiReply()

This function sends data from a terminal to a Sun MTP transaction. It is used only for replying to a CICS_EPI_EVENT_CONVERSE event.

Format

```
cics_sshort_t CICS_EpiReply  (cics_ushort_t  TermIndex,
                              cics_ubyte_t   *Data,
                              cics_ushort_t  Size);
```

where:

TermIndex	Index number of the terminal that is sending the data.
Data	Pointer to a buffer of 3270 data to be sent to the transaction. This field cannot be a NULL pointer; it must contain at least the 3270 AID byte that causes the terminal read.
Size	Size of the Data buffer.

Return Codes

CICS_EPI_NORMAL

   Successful completion.

CICS_EPI_ERR_NOT_INIT

   Initialization is not completed.

CICS_EPI_ERR_FAILED

   Unable to send the reply data.

CICS_EPI_ERR_BAD_INDEX

   TermIndex parameter does not represent a valid terminal.

CICS_EPI_ERR_NO_CONVERSE

   No CICS_EPI_EVENT_CONVERSE event is outstanding for the terminal.

CICS_EPI_ERR_NO_DATA

   No reply data is provided.

CICS_EpiATIState()

This function allows the application to query and change the terminal handling of ATI requests. If ATI requests are enabled (CICS_EPI_ATI_ON) and an ATI request is issued, EPI automatically starts the request as soon as the terminal is available. If ATI requests are held (CICS_EPI_ATI_HOLD), any ATI requests are queued and started when ATI requests are enabled.

EPI always begins in the CICS_EPI_ATI_HOLD state. The application can change the handling of ATI requests when it is ready to allow ATI processing and provided the terminal definition allows ATI requests.

Format

```
cics_sshort_t CICS_EpiATIState   (cics_ushort_t   TermIndex,
                 CICS_EpiATIState_t   *ATIState);
```

where:

TermIndex	Index number of the terminal.
ATIState	One of CICS_EPI_ATI_ON, CICS_EPI_ATI_HOLD, CICS_EPI_ATI_QUERY, as defined in "CICS_EpiATIState_t" on page 80.On entry to this function, this field contains the required new ATI state. On exit from this function, this field contains the previous state.

Return Codes

CICS_EPI_NORMAL

   Successful completion.

CICS_EPI_ERR_NOT_INIT

   Initialization is not completed; call CICS_EpiInitialize().

CICS_EPI_ERR_FAILED

    Unable to set or query the ATI state.

CICS_EPI_ERR_BAD_INDEX

    `TermIndex` parameter does not represent a valid terminal.

CICS_EPI_ATI_STATE

    An invalid `ATIState` was given.

CICS_EpiSenseCode()

Call this function when an application detects an error on the datastream sent to it.

Note – This function is retained only for compatibility and is ignored.

Format

```
cics_sshort_t CICS_EpiSenseCode (cics_ushort_t       TermIndex,
                                 CICS_EpiSenseCode_t  SenseCode);
```

where:

TermIndex	Index number of the terminal.
SenseCode	Sense Code failure reason, one of `CICS_EPI_ATI_OPCHECK` or `CICS_EPI_ATI_REJECT`, as defined in "CICS_EpiSenseCode_t" on page 81.

Return Codes

CICS_EPI_NORMAL

    Successful completion.

CICS_EPI_ERR_NOT_INIT

    Initialization is not completed.

CICS_EpiGetEvent()

This function retrieves an event from the event queue for processing.

Format

```
cics_sshort_t CICS_EpiGetEvent        (cics_ushort_t  TermIndex,
              CICS_EpiWait_t           Wait,
              CICS_EpiEventData_t     *Event);
```

where:

TermIndex	Index number of the terminal for which to obtain an event. This can be the constant CICS_EPI_TERM_INDEX_NONE to indicate that the next event for any registered terminal is to be returned. In this case, the application should examine the TermIndex field in the returned CICS_EpiEventData_t structure to determine the relevant terminal.
Wait	Ignored, but must be set to either of the following: CICS_EPI_WAIT CICS_EPI_NOWAIT
Event	Pointer to a structure that contains details of the event that occurred. The Data field of the structure points to the data buffer that is updated by any terminal datastream for the event. The Size field indicates the maximum size of the data buffer and is updated to contain the actual length of the data returned.

Return Codes

CICS_EPI_NORMAL

Successful completion.

CICS_EPI_ERR_NOT_INIT

Initialization is not completed.

CICS_EPI_ERR_FAILED

Unable to get next event.

CICS_EPI_ERR_BAD_INDEX

The TermIndex parameter does not represent a valid terminal.

CICS_EPI_ERR_WAIT

Wait parameter is invalid.

CICS_EPI_ERR_NO_EVENT

No events are outstanding for this terminal.

CICS_EPI_ERR_MORE_DATA

`Data` buffer is insufficient to contain the terminal's data; the data is truncated.

CICS_EPI_ERR_MORE_EVENTS

An event is successfully obtained, but more events against this terminal are outstanding.

CICS_EpiGetSysError()

This function obtains detailed error information about the last error that occurred. Error information is saved by EPI when any EPI command failed with a return code of `CICS_EPI_ERR_FAILED`

If an EPI function gave this return code, call this function specifying the `TermIndex` relevant to the original EPI request. The value returned in the `SysErr` parameter further describes the return code from any other EPI function.The values are operating system and environment specific and are explained in the documentation provided for each environment.

Format

```
cics_sshort_t CICS_EpiGetSysError(cics_ushort_t      TermIndex,
                                  CICS_EpiSysError_t *SysErr);
```

where:

TermIndex	Index number of the terminal for which to obtain the additional error information. This can be the constant `CICS_EPI_TERM_INDEX_NONE` to indicate that further error information is required from the `CICS_EpiInitialize()`, `CICS_EpiTerminate()`, `CICS_EpiListSystems()`, or `CICS_EpiAddTerminal()` functions.
SysErr	Pointer to a `CICS_EpiSysError_t` structure that contains the system error information.The values in the `Cause` and `Value` fields of the `SysErr` structure can be used to further qualify the return code for an EPI function.
	The `Msg` field returns an operating environment-specific text message describing the error that occurred, if available.

Return Codes

CICS_EPI_NORMAL

   Successful completion.

CICS_EPI_ERR_NOT_INIT

   Initialization is not completed.

CICS_EPI_ERR_BAD_INDEX

   TermIndex parameter does not represent a valid terminal.

CICS_EPI_ERR_FAILED

   Unable to obtain the SysErr information.

Other

   Return code from the command that caused the last error. Refer to the sections
   on the original calls to decide on the meaning of these values. This is especially
   relevant to the CICS_EpiAddTerminal() function.

When the values returned in the SysErr parameter are relevant, the following
values for the Cause field have the listed meanings.

CICS_EPI_SYSERROR_UNEXPECTED_DATASTREAM

   Sun MTP Client received a datastream from Sun MTP that it could not decode.

CICS_EPI_SYSERROR_NO_MEMORY

   Sun MTP Client was unable to obtain memory for its internal processing.

CICS_EPI_SYSERROR_DUPLICATE_NETNAME

   NetName specified on the CICS_EpiAddTerminal() call is already being
   used.

CICS_EPI_SYSERROR_UNKNOWN_NETNAME

   NetName specified on the CICS_EpiAddTerminal() call is unknown to the
   Sun MTP region to which you are communicating.

CICS_EPI_SYSERROR_UNKNOWN_DEVTYPE

   DevType specified on the CICS_EpiAddTerminal() call is unknown to the
   Sun MTP region to which you are communicating.

CICS_EPI_SYSERROR_INVALID_TPNAME

   The terminal install transaction CTIN is not available on the remote Sun MTP
   region.

CICS_EPI_SYSERROR_UNEXPECTED_ERROR

   An internal error has occurred.

CICS_EPI_SYSERROR_UNKNOWN_SYSTEM

   System specified was unknown to Sun MTP Client.

CICS_EPI_SYSERROR_TERM_OUT_OF_SERVICE

Terminal corresponding to the specified `NetName` is set out of service.

CICS_EPI_SYSERROR_SYSTEM_UNAVAILABLE

Cannot access the specified system at the present time.

CICS_EPI_SYSERROR_INTERNAL_LOGIC_ERROR

An internal error has occurred.

CICS_EPI_SYSERROR_AUTOINSTALL_FAILED

Terminal autoinstall failed for an unknown reason.

CICS_EPI_SYSERROR_TERM_INSTALL_FAILED

Terminal install failed for an unknown reason.

CICS_EpiInquireSystem()

This function supplies limited information about a terminal. When a `TermIndex` is specified, this function returns the name of the system to which the terminal is attached.

Format

```
cics_sshort_t CICS_EpiInquireSystem (cics_ushort_t  TermIndex,
                                     cics_char_t    *System);
```

where:

TermIndex	Index number of the terminal for which system information is required.
System	Pointer to a buffer that contains the name of the system to which the `TermIndex` is connected.

Return Codes

CICS_EPI_NORMAL

Successful completion.

CICS_EPI_ERR_NOT_INIT

Initialization is not completed. Call `CICS_EpiInitialize()`.

CICS_EPI_ERR_BAD_INDEX

`TermIndex` parameter does not represent a valid terminal.

```
CICS_EPI_ERR_FAILED
```
Unable to obtain system information.

KIXTERM.INI

This appendix describes the initialization file, KIXTERM.INI, which contains definitions for color and the keyboard that are used by the ECI/EPI Client terminal emulator. The file contains three sections

Section in File	Contents
[keys]	Key mappings, described in "Key Mappings" on page 102
[system_colors]	Color mapping from RGB values to symbolic names, described in "Defining the Normal and Light Colors for a Terminal" on page 104
[colors]	Color mappings from symbolic names to 3270 values, described in "Mapping Colors" on page 105
[general]	All other initialization settings, described in "Resetting a Keyboard" on page 105.

In addition, the initialization file contains comments, the format of which is described in "Identifying File Comments" on page 101.

Identifying File Comments

Identify a comment by preceding the comment text with a semicolon, for example:

```
; ---------------------------
; 3270 color designation
; ---------------------------
```

Key Mappings

The [keys] section of the KIXTERM.INI file contains the key mappings.

Format

 3270 key = System Key [+ *Modifier key*]

where:

3270 key	Defined by the terminal emulator as a key that performs a specific 3270 operation. All 3270 keys have a symbolic representation. See TABLE A-1.
System Key	Defined as a symbolic representation of a key on a PC keyboard that is supported by the terminal emulator. See TABLE A-2 for the definitions.
Modifier key	Allow 3270 keys to be represented by key combinations. For example, the following defines the reset key as Alt R reset = R + Alt See TABLE A-3 for the definitions.

The following table shows the 3270 keys.

TABLE A-1   KIXTERM.INI 3270 Keys

Symbolic Name	3270 Operation
backspace	Deletes the character to the left of the cursor in unprotected fields and moves the cursor 1 position left.
backtab	Moves the cursor to the first character position in the current field, if it is unprotected and it is not in the first character position. Otherwise, the cursor is moved to the previous unprotected field.
clear	Clears the screen and moves the cursor to the home position. An AID is transmitted.
cursorleft	Moves the cursor 1 position to the left.
cursordown	Moves the cursor 1 line down.
cursorright	Moves the cursor 1 position to the right.
cursorup	Moves the cursor 1 line up.
delete	Deletes the character under the cursor.
enter	Transmits the enter AID.

Symbolic Name	3270 Operation
eraseeof	Erases all characters from the current cursor position to the end of the field.
eraseinput	Erases all unprotected fields.
insert	Toggles the terminal between insert and overtype mode.
newline	Moves the cursor to the first unprotected field on a line.
pa1 through pa3	Transmits the appropriate PA key AID.
pf1 through pf24	Transmits the appropriate PF key AID.
printscreen	Prints the current display on the local printer.
reset	Resets the keyboard lock.
tab	Moves the cursor to the first character position of the next unprotected field.

The following table shows the system keys.

TABLE A-2   KIXTERM.INI System Keys

Symbolic Name	System Key
A through Z	Keys A through Z
0 through 9	Keys 0 through 9
backspace	Backspace key
delete	Delete key
down	Down arrow
end	End key
escape	Esc key
f1 through f24	Function keys
home	Home key
insert	Insert key
left	Left arrow
rightctrl	Right control key
newline	Newline (enter) key
numpad0 through numpad9	Keys 0 through 9 on the numeric keypad
numpad+	+ key on the numeric keypad
numpad-	- key on the numeric keypad
numpad*	* key on the numeric keypad

TABLE A-2 KIXTERM.INI System Keys *(Continued)*

Symbolic Name	System Key
numpad/	/ key on the numeric keypad
numpad.	. key on the numeric keypad
pagedown	Page down key
pageup	Page up key
pause	Pause key
printscreen	Print screen key
right	Right arrow
scroll_lock	Scroll lock key
tab	Tab key
up	Up arrow

The following table shows the modifier keys.

TABLE A-3 KIXTERM.INI Modifier Keys

Symbolic name	System Key
Shift	Shift key
Control	Left Control key
Ctrl	Left Control key
Alt	Right Alt Key

Defining the Normal and Light Colors for a Terminal

The `system_color` mapping allows the definition of the normal and light forms of color used by the terminal. TABLE A-4 lists the system colors.

Format

system_color = RGB(*red*, *green*, *blue*)

where *red*, *green*, and *blue* are numbers between 0 and 255 that represent the amount of color used in the resulting system color. For example:

```
light_white = RGB(255,255,255)
```

Mapping Colors

Color mapping allows you to map from symbolic names to 3270 values. The color mapping takes two forms, which define the color representation used by the emulator

- When displaying 3270 fields that contain color attributes. For example:

  ```
  3270 color = System color
  ```

- When displaying fields that do not contain any color attributes. In this form, the color of a field is determined by the field attribute. For example, the following statement causes all normal unprotected fields to appear as light green, by default:

  ```
  normal_unprotected = light_green
  ```

The following table defines the symbolic representation of 3270 colors, system colors, and field attributes:

TABLE A-4   KIXTERM.INI Colors

3270 Colors	System Colors	Field Attributes
Default	red / light_red	normal_unprotected
red	green / light_green	normal_protected
green	blue / light_blue	intensified_unprotected
blue	pink / light_pink	intensified_protected
pink	cyan / light_cyan	
turquoise	yellow / light_yellow	
yellow	white / light_white	
white	black	

Resetting a Keyboard

The general section contains initialization options that do not relate to key or color mapping. In this release of the emulator, only one entry is valid:

```
kbd_reset = end_of_transaction
```

The `kbd_reset` option causes a keyboard reset to occur every time a transaction ends. The option is incorporated into the emulator because it makes the terminal easier to use. This action is not part of the standard 3270 architecture and, if it is not required, you can remove it from the `KIXTERM.INI` file by commenting it out with a semi-colon:

```
; kdb_reset = end_of_transaction
```

If you comment it out, you may have to reset the keyboard if a transaction locks the keyboard.

Messages

This chapter describes how to examine Sun MTP Client messages, defines the categories and form used for messages, and the meaning of Sun MTP Client and 3270 terminal emulator messages.

Examining Messages

Sun MTP Client issues various status and error messages. You can examine these messages in one of two ways:

- On the Sun MTP Client Messages Display
- Reviewing the message log file `KIXCLI.MSG`

Note that only the last 100 messages are displayed on the Sun MTP Client Messages Display.

Message Format

All messages have the same form regardless of their category.

Format

UCM####*a Text of message*

where:

UCM	Error generated by the Sun MTP Client
EMU	Error generated by the 3270 terminal emulator
####	Sun MTP error number.
a	Letter indicating the severity of the error. I: Informative message W: Warning message E: Non-fatal error occurred F: Fatal error occurred T: Transaction abend occurred that did not affect Sun MTP.
Text of message	Message issued by Sun MTP. In some cases, the name of subroutine that originated the message is given in brackets before the message text. In these cases, the message appears as follows: UCM####*a* [*subroutine_name*] *Message* For convenience, the messages are listed in numerical order. Within the Text of message, the following variable conventions are used *%c*: Replaced in the message by a single character. *%d*: Replaced in the message by a by a decimal value. *%s*: Replaced in the message by a string of characters. *%r*: Replaced in the message by the name of a subroutine. *%x*: Replaced in the message by a hexadecimal value

Sun MTP Client Messages

The message log file, KIXCLI.MSG, is written to the directory specified by the MsgDir parameter in the KIXCLI.INI configuration file. This message log contains all the messages for the most recent execution of Sun MTP Client. Each message includes a date and time stamp. KIXCLI.MSG is overwritten each time Sun MTP Client is started. This section lists the messages returned by Sun MTP Client.

UCM0001I: Sun MTP Client startup is complete

Description: Informational message.

UCM0002W: Unable to create timer

UCM0003E: Unable to resolve TCP host %s for system %s

UCM0004W: Unable to connect to TCP host %s for system %s

UCM0005E: Lost contact with TCP host %s for system %s:

UCM0006I: TCP/IP transport support loaded
   Description: Informational message.

UCM0007I: Connected to system %s as ApplID %s
   Description: Informational message.

UCM0008E: Redefinition of system %s; the first definition is being used

UCM0009W: No systems found

UCM0010I: Winsock used is %s
   Description: Informational message.

UCM0011E: The definition of system %s has insufficient parameters
   Solution: Supply the necessary parameters.

UCM0012E: The transport %s for system %s is invalid

UCM0013E: The port %s for system %s is not numeric

UCM0014E: Default system %s does not exist. Using %s

UCM0015W: No default system specified. Using %s

UCM0016E: Alias %s is too long; it is being ignored

UCM0017W: Trace is enabled. TraceMask is 0x%4.4x

UCM0018W: Alias %s has port 0 specified. The default port %d will be used

UCM0019W: Invalid MaxRequests value of %s specified

UCM0020I: Setting MaxRequests to %d
   Description: Informational message.

UCM0021W: Invalid MaxSystems value of %s specified

```
UCM0022I: Setting MaxSystems to %d
```
   Description: Informational message.

```
UCM0023E: The TCP address %s for system %s is longer than the
maximum %d characters:
```

```
UCM0024W: The description for system %s is longer than the
maximum %d characters
```

```
UCM0025I: A request has been made to cease communication with
system %s
```
   Description: Informational message.

```
UCM0026E: Resource shortage while in function %s
```

```
UCM0027E: Specified TraceDir is invalid. Using default TraceDir
```

```
UCM0028E: Specified MsgDir %s is invalid. Using %s
```

```
UCM0029E: Trace file could not be opened. Trace is disabled
```

```
UCM0030E: Client Install Transaction CCIN is invalid in
system %s
```

```
UCM0031E: Unexpected return code received while decoding a CCIN
reply from system %s
```

```
UCM0032E: Client Install Transaction CCIN could not be executed
on system %s
```

```
UCM0033W: LU6.2 Error Reply received from system %s.
Sense data is {%2.2x,%2.2x,%2.2x,%2.2x}
```

```
UCM0040I: Copyright © 2001 by Sun Microsystems, Inc.
```
   Description: Informational message.

```
UCM0041I: -------------------------------------------
```
   Description: Informational message.

```
UCM0042W: MaxSystems has been reached. Connection to system %s
has been aborted.
```

```
UCM0043E: The Sun MTP Client could not be contacted.
```

UCM0044I: kixctl [-s] [-l] [-m] [-D] [-c <system>]
[-d <system>] [-t <mask>]
   Description: Informational message.

UCM0045E: kixctl [-s] [-l] [-m] [-D] [-c <system>]
[-d <system>] [-t <mask>]

UCM0046I: A connection request has been issued for system '%s'
   Description: Informational message.

UCM0047E: System name '%s' is not valid

UCM0048E: System name '%s' could not be connected

UCM0049I: A disconnection request has been issued for system
'%s'
   Description: Informational message.

UCM0050E: System '%s' could not be disconnected

UCM0051E: Contact with the Sun MTP Client has been lost

UCM0052E: Remote system %s sent invalid block
header %2.2x%2.2x%2.2x%2.2x%2.2x%2.2x%2.2x%2.2x

UCM0053I: A request to shutdown the Sun MTP Client has been
issued
   Description: Informational message.

UCM0054I: Sun MTP Client shutdown is complete
   Description: Informational message.

UCM0055E: A reply from the Sun MTP Client could not be obtained

UCM0056I: A dump is being taken into %s
   Description: Informational message.

UCM0057E: A request to dump could not be performed

UCM0058E: The Sun MTP Client could not create pipe %s

UCM0059I: Destroying terminated application pipe %s
   Description: Informational message.

UCM0060E: Could not delete file %s

UCM0061E: Could not create file %s

UCM0062E: The dump request could not be processed

UCM0063E: A dump request has been issued

UCM0064E: The TCP/IP transport support library could not be
loaded

UCM0065I: SNA transport support loaded

Description: Informational message.

UCM0066W: An SNA conversation could not be allocated.
primary_rc = 0x%4.4x, secondary_rc = 0x%8.8lx

UCM0067E: The SNA Remote LU name %s for system %s is longer
than the maximum %d characters

UCM0068E: The SNA Local LU name %s for system %s is longer than
the maximum %d characters

UCM0069E: The SNA Mode name %s for system %s is longer than the
maximum %d characters

UCM0070E: The Remote LU Alias %s for system %s is invalid

UCM0071E: The Mode Name %s for system %s is invalid

UCM0072W: The local SNA Transaction Program could not perform a
TP_START. primary_rc = 0x%4.4x, secondary_rc = 0x%8.8lx

UCM0073E: The Local LU Alias %s for system %s is invalid

UCM0074E: The SNA transport support library could not be loaded

UCM0075E: WinAPPCStartup() failed with return code 0x%8.8x

UCM0076E: The SNA PU2.1 node could not be contacted

UCM0077E: The underlying network subsystem is not ready for
network communication

UCM0078E: A connection attempt to system %s failed due to a
transport initialization failure

UCM0079E: An attempt to run the terminal install transaction
CTIN failed on system %s

UCM0080I: %s BuildStamp %s
  Description: Informational message.

Emulator Messages

This section describes the messages generated by the Sun MTP Client 3270 terminal emulator and printer emulator.

EMU0001E Unknown error

EMU0002E Terminal not initialized

EMU0003E Bad Index

EMU0004E Failed

EMU0005E Unexpected datastream

EMU0006E Out of memory

EMU0007E Duplicate netname

EMU0008E Unknown netname
  Description: Netname is not defined.

  Solution: Define or correct the netname.

EMU0009E Unknown devtype

EMU0010E Terminal install failed

EMU0011E Unexpected error

EMU0012E Unknown system
  Description: System is not defined.

  Solution: Define or correct the name of the system.

EMU0013W Terminal out of service

```
EMU0014E System unavailable

EMU0015E Internal Logic error

EMU0016E Auto Install failed

EMU0017E Terminal Install error

EMU0018E EPI version not supported

EMU0019E Is Init

EMU0020E No Systems

EMU0021E No more terminal resources

EMU0022E System Error

EMU0023E Transaction active

EMU0024E TTI active

EMU0025E No server connection

EMU0026E Invalid data length

EMU0027E Invalid datastream

EMU0028E Terminal install error

EMU0029E Closing terminal

EMU0032W Are you sure you want to quit?

EMU0033E Terminal Initialization Error: 0x%4.4x

EMU0034E No connections available

EMU0040I Printer installed as netname: %s

EMU0041W Printer not installed

EMU0042I Printer file: %s
```
Description: Informational message.

```
EMU0043I Printer Command: %s
```
   Description: Informational message.

```
EMU0044E Cannot execute command: 0x%4.4x
```

```
EMU0045E Cannot open file: %s
```

```
EMU0046E Cannot open temporary file: %s
```
   Description: System cannot locate the named command or file.

   Solution: Check the syntax for the command and verify that you are using the correct file name, then resubmit.

Glossary

3270 SNA device	A terminal device that displays an IBM SNA 3270 datastream.
API	See Application Programing Interface
Application Programming Interface (API)	A predefined interface used by application programs. The API consists of a routine name and its associated arguments and adheres to the syntax of the associated application program language.
asynchronous processing	A bi-directional process that allows a mainframe to start transactions on Sun MTP or allows Sun MTP to start transactions on the mainframe while continuing to process.
Basic Mapping Support (BMS)	Macro instructions, which are used as input to the Sun MTP BMS Assembler to create physical and symbolic definition map files.
BMS	See Basic Mapping Support.
CICS	See Customer Information Control System.
conversational transaction	One in which dialog with the user (typically a `SEND/RECEIVE` sequence) is carried on while the transaction is active.
Customer Information Control System (CICS)	A general purpose transaction processing environment.
Distributed Program Link (DPL)	Method of intersystem communication in which a program on one region can synchronously link to a program on another region.

Distributed Transaction Processing (DTP)	The distribution of processing between transactions that communicate synchronously with one another over intersystem or interregion links.
DPL	See Distributed Program Link.
DTP	See Distributed Transaction Processing.
ECI	See External Call Interface.
EPI	See External Presentation Interface.
environment variables	Variables that define the location of program files and applications. Both clients and the server use environment variables.
External Call Interface (ECI)	API for writing programs that allow non-CICS application programs to call a CICS program that follows the rules for Distributed Program Link (DPL) and is running on a server
External Presentation Interface (EPI)	API for writing programs that allow a non-CICS application program to appear to Sun MTP as one or more standard 3270 terminals. The EPI application communicates with Sun MTP as if it is a real 3270 terminal.
File permissions (or modes)	Control access to a file as defined by the operating system.
Intersystem Communication (ISC)	Communication between separate systems by means of SNA networking facilities or by means of the application-to-application facilities of an SNA access method.
ISC	See Intersystem Communication
logical unit (LU)	In SNA, a port through which an end user accesses the SNA network to communicate with another end user and through which the end user accesses the functions provided by system services control points (SSCPs).
logical unit-of-work	A logical unit-of-work is all the processing in the server that is needed to establish a set of updates to recoverable resources.
LU	See logical unit.
LU6.2	Logical unit type that supports general communication between programs in a distributed processing environment.
SNA	See Systems Network Architecture.
sockets	A mechanism for interprocess communication that allows the use of different network protocols.

SSCP	See System Services Control Point.
System Services Control Point (SSCP)	In SNA, the focal point within an SNA network for managing the configuration, coordinating network operator and problem determination requests, and providing directory support and other session services for end users of the network.
Systems Network Architecture (SNA)	The logical structure, formats, protocols, and operational sequences for transmitting information units through, and controlling the configuration and operation of networks.
TCP/IP	A suite of network protocols on which the Internet is based. Transmission Control Protocol (TCP) is a protocol that provides a reliable, full-duplex datastream. Internet Protocol (IP) is a protocol that provides the packet delivery services for TCP. The TCP protocol interacts with the IP, not the user process.
TN3270 protocol	An extension of the traditional TCP/IP Telnet protocol that allows non-ASCII, block mode devices, such as IBM-3270s, and applications, such as EBM, to communicate via TCP/IP. Also includes TN3270E.
TN3270 Server (`unikixtnemux`)	Allows Sun MTP to support 3270 emulators running on PCs, Macintosh, and UNIX machines using the TCP/IP - TN3270 protocol. Also supports TN3270E.
Sun MTP region	A set of UNIX processes, files, and environment variables that define separate CICS applications on a system.
transaction routing	Allows terminals connected to one Sun MTP/CICS region to run transactions on another Sun MTP/CICS region either on the same or a different machine.
`unikixtnemux`	See TN3270 Server.

Index