EMTS User Guide
Version 2.10

1 Introduction

The Ethernet Multi-User Transputer System (EMTS) allows workstations on a network to share transputer resources. The transputers are hosted by one or more IBM PCs attached to the network. Each PC can host multiple B004-compatible "target" transputer systems, each of which is available to a separate workstation user. Access to the transputers is provided using the industry standard TCP/IP protocol suite running over Ethernet[1].

The system is fully compatible with the Inmos `iserver`, meaning that it supports standard transputer software such as the Inmos D700E (TDS3), the Inmos D7205 (Occam toolset), the Inmos D7214 (ANSI C toolset), etc. Later versions of the EMTS will support transputer development tools that use the `afserver`.

This guide is intended for people who wish to use the EMTS. Please refer to the **EMTS Installation Guide** for information on how to install the EMTS software at your site.

The next section gives a brief overview of the EMTS software. Section 3 explains how to use the software and includes full documentation on the command-line options and environment variables that affect the operation of the system. Section 4 describes the extensions that have been added to the server in order to provide access to X windows pseudo-terminals and operating system processes (if supported by your workstation). Section 5 describes the implications and limitations of using the system.

2 Overview

The EMTS provides network access to a collection of transputer resources. The EMTS *stub* program (`estub`) runs on an IBM PC and hosts a transputer resource (one or more *target* transputer systems). An EMTS *server* program (`eserver` or `iserver`, see below) runs on an IBM PC or a Sun workstation and allows the user to access a remote target transputer system. Refer to Figure 1 for an illustration of the system. The stub transparently relays data between server(s) and target(s). An EMTS *who* program (`ewho`) reports status information on who is using the transputer resources. In all cases, the IBM PC may be running either MS-DOS or Unix System V Release 4.

Two servers are provided with the distribution: the EMTS `eserver` and the Inmos `iserver`. The `iserver` is the Inmos version 1.42h which has been linked with the EMTS link module (`emtslink`) to allow it to access remote transputers. The `eserver` is an enhanced transputer server program that can be used in place of the `iserver`. Refer to Section 4 for further information on the EMTS server extensions. Refer to Section 3 of the **EMTS Installation Guide** for information on the system requirements in order to support the EMTS server extensions.

---

1 *Other network technologies, such as Token Ring, can be supported if required.*

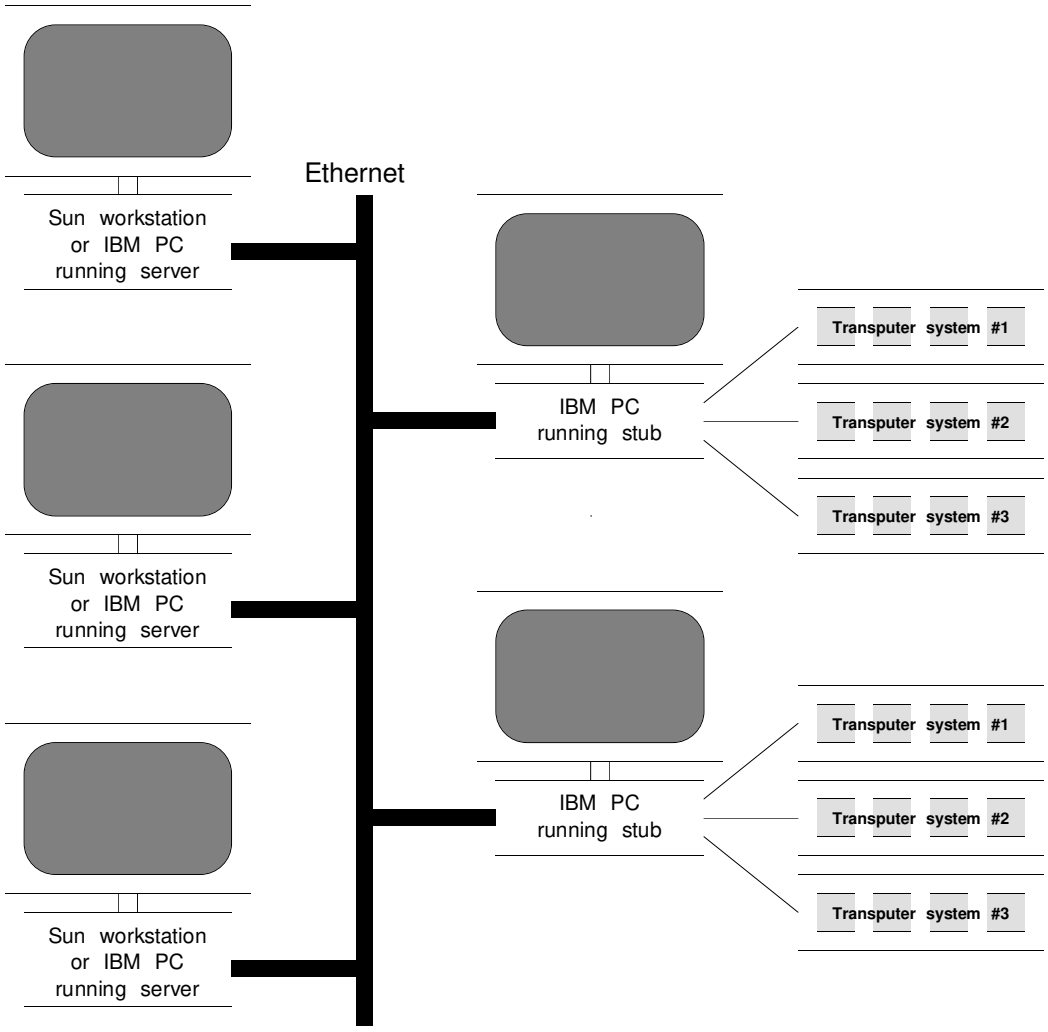

Figure 1: An EMTS installation

3 Using the EMTS

This section explains how to use the EMTS once it has been installed on your PC(s) and workstation(s). It describes the command-line options and environment variables that affect the behaviour of the stub and server[2]. This information is summarised in the Unix on-line manual pages for the `estub`, the `eserver` and `ewho` programs. These manual pages are also available in PostScript format.

2 *References to "server" in this section apply to the EMTS* `eserver` *program only; refer to Inmos documentation for information on the* `iserver`.

3.1 General points

3.1.1 Usage conventions

The stub and server programs obey the same usage conventions as the Inmos tools. Invoking the programs without any options causes them to print help information and then exit. All options are introduced by a two-letter keyword beginning with 's' and prefixed by the '-' switch character (or the '/' character under MS-DOS). The options can be given in any order with or without intervening spaces.

3.1.2 Environment variables

In the absence of certain command-line options, the programs attempt to read equivalent environment variables to assign the values of various parameters. In the presence of these same command-line options, the values of the corresponding environment variables are ignored (ie command-line values always override environment settings).

3.1.3 Information, Warning, Debug and Error Messages

By default, the stub and server print no information, warning or debugging messages to the screen. Error messages are always printed (to the standard error stream). Information, warning and debugging messages can be selectively enabled using command-line switches. Setting the `verbose` switch causes information and warning messages to be generated (there is no way of generating only warning messages). Setting the `debug` switch causes debug messages to be generated.

3.1.4 Error handling

If the server detects an error during a stub-server session, it terminates returning an `iserver`-compatible exit code to the operating system. It also prints a unique abort code and a brief explanatory message to the standard error stream.

If the stub detects an error *before it enters its serving loop*, it terminates returning an appropriate exit code to the operating system. It also prints a unique abort code and a brief explanatory message to the standard error stream. If the stub detects an error *after it enters its serving loop*, it sends one of a small number of error packets to the server (causing the server to terminate with a generic error message) and terminates the session. It also prints a unique error code and a brief explanatory message to the standard error stream. It does not terminate, since there may be other valid stub-server sessions in progress. In this way, an error on one stub-server session does not affect any other stub-server sessions.

The format of a server or stub *abort* message is as follows:

```
<progname>: Abort <procid>:<abortid> - <abortmessage>
```

where `<progname>` is the name of the program (`eserver` or `estub`), `<procid>` is a number identifying the program procedure from which the abort has originated, `<abortid>` is a unique number identifying the reason for the abort, and `<abortmessage>` is a brief message indicating the reason for the abort.

The format of a stub *error* message is as follows:

```
<progname>: Error <procid>:<errorid> - <errormessage>
```

where `<progname>` is the name of the program (`estub`), `<procid>` is a number identifying the program procedure from which the error has originated, `<errorid>` is a unique number identifying the reason for the error, and `<errormessage>` is a brief message indicating the reason for the error.

3.1.5 Terminating the stub and server

The stub can be terminated by sending an interrupt signal to the program. The server terminates if it detects an error condition, if it is instructed to terminate by a client transputer process (see your Inmos documentation), or if it is sent an interrupt signal from the operating system. To interrupt the server, the user can type `CTRL-C` (or `CTRL-BREAK` under MS-DOS) at the keyboard (if the standard input stream is connected to the keyboard) or use the Unix `kill` command (see your Unix documentation). When the program catches an interrupt signal, it invokes an interrupt handler if its standard input and standard error streams are connected to a terminal (otherwise it terminates immediately). The interrupt handler gives the user the opportunity to confirm the interrupt (causing the program to terminate), to cancel the interrupt (causing the program to continue), to run a subshell (typing `exit` to the subshell causes the program to resume), to toggle the status of the `verbose` switch, and to toggle the status of the `debug` switch.

3.1.6 Redirecting input or output

For both the stub and server, the shell normally arranges that the standard input stream be connected to the keyboard, and that the standard output and error streams be connected to the screen. In some circumstances, it may be convenient to redirect these streams using your shell's i/o redirection conventions.

For example, to log the output[3] of the stub to a file called `c:\tmp\stub.log`, you would use:

```
C:\> estub {options} > c:\tmp\stub.log
```

To log the standard output stream of the server to a file called `/tmp/server.log` and to leave the standard error stream connected to the controlling terminal, you would use:

---

3 *There is no way to separate the standard output and standard error streams using MS-DOS.*

```
example%  eserver  {options}  >  /tmp/server.log
```

To run the server in the background, directing the standard output stream to a file called `/tmp/server.log`, the standard error stream to the console, and causing the server to exit immediately on receiving an interrupt signal, you would use (in C shell notation):

```
example%  (eserver  {options}  >  /tmp/server.log)  >&  /dev/console  &
```

3.2 Using the stub

3.2.1 Stub command-line options

The stub accepts the following command-line options:

`-si`		information mode
`-sd`		debug mode
`-ss`		enter serving loop
`-se[n]`		check error flag of target 0 [n]
`-sr[n]`		reset target 0 [n]
`-st[n]`	`<linkname>`	assign named link to target 0 [n]
`-sc[n]`	`<filename>`	copy specified file to target 0 [n]
`-sb[n]`	`<filename>`	boot target 0 [n] with specified file

If the target identifier suffix `[n]` is omitted, the option applies to target 0. If the target identifier suffix `[n]` is supplied, it may either be the `*` character (in which case the option applies to all targets - illegal for the `-st` option) or a single digit (in which case the option applies to the specified target).

A limit on the number of target boards that can be handled by the stub is set at compile-time; invoke the stub program without any options to find out this maximum (which will always be between 1 and 9). Each target is referenced by a target identifier `n`, where `n` ranges from 0 to the maximum number of targets minus 1.

The link name to assign to target `n` can be specified using the environment variable `EMTS_TARGETn` (section 3.2.2.1) instead of the `-st[n]` option. If no `-st` options are present on the command-line and there are no equivalent environment variables, the stub uses the default link name of #150 (under MS-DOS) or `/dev/b004` (under Unix) and handles only a single target (target 0).

Each option is described below.

3.2.1.1 The -si option

By default, the stub prints no information or warning messages to the screen. Supplying the `-si` option puts the stub in `verbose` mode causing it to print a small number of information messages and warning messages (if any) to the standard error stream.

3.2.1.2 The -sd option

By default, the stub prints no debugging messages to the screen. Supplying the `-sd` option puts the stub in `debug` mode causing it to print copious debugging messages to the standard error stream. This option is not normally required.

3.2.1.3 The -ss option

The `-ss` option causes the stub to enter its serving loop in which it:

- listens for incoming TCP connection set-up requests from servers;
- listens for incoming UDP status request packets;
- listens for SP request packets from any of the target transputer systems and relays these requests to the appropriate server;
- listens for SP reply packets from any of its servers and relays these to the appropriate transputer target.

Note that this option is implicit in the `-sb` option (section 3.2.1.8).

3.2.1.4 The -se[n] option

The `-se[n]` option causes the stub to test the transputer error flag on target 0 (if `n` is not supplied), on target `n` (if `n` is a single digit), or on all targets (if `n` is the `*` character). This option is not normally required since there is an equivalent option for the server (section 3.3.1.4).

3.2.1.5 The -sr[n] option

The `-sr[n]` option causes the stub to reset target 0 (if `n` is not supplied), target `n` (if `n` is a single digit), or all targets (if `n` is the `*` character). This option is not normally required since there is an equivalent option for the server (section 3.3.1.6). Note that this option is implicit in the `-sb` option (section 3.2.1.8).

3.2.1.6 The -st[n] <linkname> option

The `-st[n] <linkname>` option causes the stub to open the specified link and to associate the specified link name with the target identifier 0 (if `n` is not supplied) or the target identifier `n` (if `n` is a single digit). Under MS-DOS, `<linkname>` is interpreted as the hexadecimal i/o

address of the target system which may optionally be prefixed by a # character. Under Unix, <linkname> is interpretted as the name of a device file of the form /dev/b004[xy] which maps onto the i/o address specified at kernel link time. If the stub is required to handle multiple target transputer systems, there may be multiple instances of this option in order to specify the link name for each target. This option does not have to be present if you have set an equivalent environment variable or if you have only a single target transputer system accessible via the default link name (#150 under MS-DOS and /dev/b004 under Unix).

3.2.1.7 The -sc[n] <filename> option

The -sc[n] <filename> option causes the stub to copy the named file to target 0 (if n is not supplied), to target n (if n is a single digit), or to all targets (if n is the * character). This option is not normally used required since it is implicit in the -sb option (section 3.2.1.8) and because there is an equivalent option for the server (section 3.3.1.12).

3.2.1.8 The -sb[n] <filename> option

The -sb[n] <filename> option causes the stub to boot target 0 (if n is not supplied), target n (if n is a single digit), or all targets (if n is the * character) with the named file. It is equivalent to specifying the -sr -ss -sc[n] <filename> options. This option may be useful if it is required to boot a target with a bootable file from the stub host's filing system instead of from the server host's filing system. Normally, however, this option is not required since files will be loaded from the server (section 3.3.1.13).

3.2.2 Stub environment variables

3.2.2.1 EMTS_TARGETn

The link name of a target transputer system can be specified using an environment variable instead of using the command-line. For values of n from 0 to the maximum target identifier, if the link name of target n is not specified on the command-line the stub attempts to read the value of the environment variable EMTS_TARGETn.

3.2.2.2 EMTS_INFOn

The value of the environment variable EMTS_INFOn should be a brief description of the target transputer system n. This string is returned to users of the ewho program (secion 3.4)

3.2.3 Example stub invocations

To instruct the stub to display help information, invoke the stub without any options:

```
C:\>  estub
```

To instruct the stub to run in verbose mode, to open the default link, to map (implicitly) the link to the target identifier 0, and to enter its serving loop waiting for connections:

```
C:\>  estub  -si  -ss
```

To instruct the MS-DOS stub to handle three target transputer systems at addresses #200, #250 and #300, to boot the target at #200 with the file `c:\bin\mtdemo.bah`, to boot the other two targets with the file `c:\bin\xdemo.bah`, to reset all targets (implicitly), and to enter its serving loop (implicitly):

```
C:\>  estub  -st0  #200  -st1  #250  -st2  #300  -sb*
c:\bin\xdemo.bah  -sb0  c:\bin\mtdemo.bah
```

To instruct the Unix stub to open the specified links, to map device file `/dev/b00400` to target identifer `0`, to map device file `/dev/b00401` to target identifier `1`, to run in `verbose` mode and to enter its serving loop:

```
example%  estub  -st0  /dev/b00400  -st1  /dev/b00401  -si  -ss
```

3.3 Using the server

3.3.1 Server command-line options

The server accepts the following command-line options (which are a superset of those accepted by the `iserver`):

`-si`		information mode
`-sd`		debug mode
`-ss`		serve transputer
`-se`		monitor transputer error flag
`-sm`		use MT protocol on link
`-sr`		reset the transputer
`-sa`		analyse and peek transputer
`-sp`	`<n>`	set peek buffer size to n kbytes
`-sl`	`<linkname>`	use specified link name
`-st`	`<targetid>`	use specified target
`-sn`	`<stubname>`	use specified stub
`-sc`	`<filename>`	copy specified file to target transputer
`-sb`	`<filename>`	boot transputer with specified file

The name of the stub machine must be specified using either the `EMTS_STUBNAME` environment variable (section 3.3.2.2), the `TRANSPUTER` environment variable (section 3.3.2.1), or the `-sn` option.

The target transputer system to select may be specified using either the `EMTS_TARGET` environment variable (section 3.3.2.3), the `TRANSPUTER` environment variable (section 3.3.2.1), or the `-st`

option. The target identifier is a single digit that is used by the stub to reference a particular transputer target transputer system. If no target identifier is specified, the stub attempts to select an arbitrary free target on behalf of the server.

Each option is described below.

3.3.1.1 The -si option

By default, the server prints no information or warning messages to the screen. Supplying the `-si` option puts the server in `verbose` mode causing it to print a small number of information messages and warning messages (if any) to the standard error stream. This option is implicit in the `-sb` option (section 3.3.1.13).

3.3.1.2 The -sd option

By default, the server prints no debugging messages to the screen. Supplying the `-sd` option puts the server in `debug` mode causing it to print copious debugging messages to the standard error stream. This option is not normally required.

3.3.1.3 The -ss option

The `-ss` option causes the server to enter its serving loop in which it receives SP request packets from the stub, processes each request, and transmits reply packets back to the stub. Packets are either processed sequentially (the default) or in parallel (if the client has elected to exploit the multi-threaded functionality of the server, section 4.4). This option is implicit in the `-sb` option (section 3.3.1.13).

3.3.1.4 The -se option

The `-se` option causes the server to instruct the stub to monitor the transputer error flag of the selected target. If the stub detects that the error flag has been set by the transputer, the server is informed causing it to terminate with a suitable error message. The user may then wish to invoke a debugger.

3.3.1.5 The -sm option

The -sm option causes the server to instruct the stub to use the `MT` protocol (section 4.4.3) instead of the `SP` protocol on the link to the target. This option should only be used with special programs that have been written to exploit the multi-threaded functionality of the server.

3.3.1.6 The -sr option

The `-sr` option causes the server to reset its selected target. This option is implicit in the `-sb` option (section 3.3.1.13).

3.3.1.7 The -sa option

The `-sa` option causes the server to instruct the stub to first analyse the selected target and then to peek 8 kbytes (by default) of transputer memory into a server buffer. The buffer can then be accessed by a debugger. This option is required in preparation for running the debugger.

3.3.1.8 The -sp <n> option

The `-sp <n>` option causes the server to allocate a peek buffer of size `<n>` Kbytes.

3.3.1.9 The -sl <linkname> option

The `-sl <linkname>` option instructs the server to connect to the server specified by the `<stubname>` part of the `<linkname>`, and use the target transputer system specified by the `<targetid>` part of the `<linkname>` (if specified). The synatax of `<linkname>` is `<stubname>[:<targetid>]`. This option can be used to override the value of the `TRANSPUTER` environment variable (section 3.3.2.1). This option is provided for compatibility with the Inmos `iserver`.

3.3.1.10 The -st <targetid> option

The `-st <targetid>` option instructs the server to select the specified target transputer system. This option is not required if you have only one target transputer system, if you have set an equivalent environment variable, or if you are happy to allow the stub to select an appropriate free target on behalf of the server. This option overrides the value of the `<targetid>` part of the `TRANSPUTER` environment variable (section 3.3.2.1), the value of the `EMTS_TARGET` environment variable (section 3.3.2.3), and the `<targetid>` part of the `<linkname>` as defined by the `-sl` option (section 3.3.1.9).

3.3.1.11 The -sn <stubname> option

The `-sn <stubname>` option instructs the server to connect to the named stub machine. This option is not required if you have set an equivalent environment variable. `<stubname>` is either the name of the stub host as registered in the `hosts` data base, or it is the IP address of the stub host in the standard "dot" notation. This option overrides the value of the `<stubname>` part of the `TRANSPUTER` environment variable (section 3.3.2.1), the value of the `EMTS_STUBNAME` environment variable (section 3.3.2.2), and the `<stubname>` part of the `<linkname>` as defined by the `-sl` option (section 3.3.1.9).

3.3.1.12 The -sc <filename> option

The `-sc <filename>` option causes the server to copy the named file to the selected target. This option is implicit in the `-sb` option (section 3.3.1.13).

3.3.1.13 The -sb <filename> option

The `-sb <filename>` option causes the server to boot the selected target with the named file. It is equivalent to specifying the `-sr -ss -sc <filename>` options.

3.3.2 Server environment variables

3.3.2.1 TRANSPUTER

The `<linkname>` can be specified by the `TRANSPUTER` environment variable instead of on the command-line. This environment variable is supported for compatibility with the Inmos `iserver`.

3.3.2.2 EMTS_STUBNAME

The name of the stub machine can be specified by the `EMTS_STUBNAME` environment variable instead of on the command-line. If the stub name is not specified on the command-line, the server attempts to read the value of the environment variable `EMTS_STUBNAME`. `<stubname>` is either the name of the stub host as registered in the `hosts` data base, or it is the IP address of the stub host in the standard "dot" notation.

3.3.2.3 EMTS_TARGET

The target identifier to use can be specified by the `EMTS_TARGET` environment variable instead of on the command-line. If the target identifier is not specified on the command-line, the server attempts to read the value of the environment variable `EMTS_TARGET`.

3.3.3 Example server invocations

To instruct the server to display help information, invoke the server without any options:

```
example%  eserver
```

To instruct the server to run in `verbose` mode, to select target 1, to connect to the stub at "saturn", and to enter its serving loop waiting for request packets from the target:

```
example%  eserver  -si  -st  1  -sn  saturn  -ss
```

Alternatively, use:

```
example%  eserver  -si  -sl  saturn:1  -ss
```

To instruct the server to select an arbitrary free target (implicitly), to reset the target (implicitly), to boot the target with the file `/usr/local/bin/mtdemo.bah`, to monitor the transputer error flag, to use the MT protocol on the link, and to enter its serving loop (implicitly), assuming that the name of the stub is specified by an environment variable:

```
example%  eserver  -sb  /usr/local/bin/mtdemo.bah  -se  -sm
```

3.4 Using the ewho program

`ewho` is a program that returns information on who is using the EMTS. If `ewho` is invoked without any options, it sends a broadcast request to all EMTS stubs on the network. Alternatively, if a `<stubname>` is specified on the command-line, `ewho` sends a request to only the specified stub.

3.4.1 Ewho command-line options

`ewho` understands the following command-line syntax:

```
ewho  [<options>]  [<stubname>]
```

`ewho` accepts the following command-line options:

`-b <address>`	use specified broadcast address
`-t <timeout>`	use specified timeout for broadcast requests
`-q`	do not print results, only set return code

Each option is described below.

3.4.1.1 The -b <address> option

This option sets the IP broadcast address to use for contacting machines on the network. This option can be used to override the value of the `EMTS_BROADCAST` environment variable (section 3.4.2.1). This option is ignored if an explicit `<stubname>` is also specified.

3.4.1.2 The -t <timeout> option

This option specifies the time (in seconds) to wait for reply packets in response to a broadcast request. The default is to wait forever. This option can be used to override the value of the `EMTS_TIMEOUT` environment variable (section 3.4.2.2). This option is ignored if an explicit `<stubname>` is also specified.

3.4.1.3 The -q option

This option surpresses printing of the result of the status enquiry, causing `ewho` to exit silently with a suitable exit code. `ewho` returns `-1` to the shell if no free targets can be found, or `n` to the shell to indicate that the target transputer system with identifier `n` is available for use. This result is not very meaningful unless an explicit `<stubname>` is also specified.

3.4.2 Ewho environment variables

3.4.2.1 EMTS_BROADCAST

The IP address (in "dot" notation) to use for broadcast can be set using the `EMTS_BROADCAST` environment variable instead of the `-b` command-line option.

3.4.2.2 EMTS_TIMEOUT

The timeout value (in seconds) to use for broadcast can be set using the `EMTS_TIMEOUT` environment variable instead of the `-t` command-line option.

3.4.3 Example ewho invocations

To broadcast status requests using the IP broadcast address `145.123.56.0`, printing the result to the standard output stream and terminating after 2 seconds:

```
example% ewho -b 145.123.56.0 -t 2
```

To poll the stub host machine `jupiter` and set the return code to be equal to the numeric target identifier of the first free target transputer system available at `jupiter`, or `-1` if no free targets are available:

```
examples% ewho -q jupiter
```

The following is an example of the information that is returned:

```
Stub at jupiter:
Tgt  User@host      Since                                  Target information
  0  brendan@pluto  Wed Mar 10 09:05:55 1993              B004, T800, 4Mb
  1  ***************  AVAILABLE  ***************           B008, 4 T800s
  2  chris@neptune  Wed Mar 10 08:00:23 1993              No information
```

4 Server extensions[4]

The full range of server extensions is only available to a Sun workstation running SunOS 4.1.x with access to the X or OpenWindows windowing environments. The `system` extension is available under Unix. The `xterm` extension is available under Unix and X11. The `multi-threading` extension is available under SunOS 4.1.x.

---

4 *References to "server" in this section apply to the EMTS* `eserver` *program only; these server extensions are not available with the Inmos* `iserver`*.*

Invoke your server without any command-line options to determine which extensions were enabled at compile-time. If `server extensions` are enabled, this means that you have access to the `system` and `xterm` extensions described below. If `multi-threading` is enabled, this means that you have access to the `multi-threading` extension described below.

4.1 General description

The server provides the following extensions to the Inmos `iserver`:

- Support for client-initiated `xterm` terminal emulator processes when the server is running under MIT's X11 or Sun's OpenWindows windowing environments. Each `xterm` process gives its client a private pseudo-terminal for terminal i/o allowing multiple clients to share the resources of a single physical terminal.

- Support for client-initiated `system` processes. A `system` process is a client-specified operating system process, the standard input and standard output streams of which are connected to the client transputer process.

- Multi-threading. In order to provide the optimal environment for the above two features, the server has been programmed to use multiple threads and non-blocking system calls when support for this is provided by the host operating system. Each request packet from a client transputer process is handled by a separate server thread. If the request cannot be serviced immediately (eg it might be a blocking i/o request), the server suspends the thread until it becomes runnable. Other requests can still be processed by the server. If the (multi-process) transputer client has been programmed to exploit the multi-threaded nature of the server, the server can be accessed by other processes within the client.

Access to the first two extensions is provided using the standard Inmos i/o libraries together with a number of additional EMTS library procedures for handling `xterm` keyboard input (section 4.2.3). Access to the multi-threaded functionality of the server is provided by using a multi-threaded server communications protocol and by adding a very simple harness to the transputer client. The mechanisms are described below.

4.2 Support for xterm processes

4.2.1 Overview

The motivation for providing this extension was to allow multiple processes in a multi-process transputer client to have access to multiple pseudo-terminals (in separate windows) in a window-based display. Previous transputer server programs (including the `iserver`) handle only a single (physical) terminal. Multiple terminal i/o streams from a transputer client must be multiplexed onto this one terminal. This has two major disadvantages:

- Terminal output from multiple client processes is interleaved on a single display;

- A blocking terminal read request from one client process blocks out all other i/o requests.

Allowing a client process to open a private pseudo-terminal (implemented by the `xterm` terminal emulator program) solves the first problem. Providing the addition of multiple threads together with support for non-blocking reads solves the second problem (section 4.4).

4.2.2 Programming guide

Access to the `xterm` server extension has been provided using the standard Inmos i/o libraries together with a couple of additional EMTS procedures for handling keyboard input (section 4.2.3). To open an `xterm` window, a client transputer process opens a "file" with the special name:

```
XTERM:[options to xterm]
```

Note that case is important. If no optional options are supplied, the trailing `:` character is optional. The client is handed back a file descriptor. Writing to the file descriptor causes output to appear in the `xterm` window. Reading from the file descriptor causes input to be read from the keyboard if the window has the keyboard focus. Closing the file descriptor causes the window to be closed.

Refer to your `xterm` documentation for details of the possible options to `xterm`. Note that you can specify a display other than your local workstation; this allows a transputer client to open an `xterm` window on any X server (Sun or otherwise) on your network. For example, to use display 0 on the X server "jupiter", you should open a "file" called:

```
XTERM:-display jupiter:0
```

You must have permission to use the specified X server (see your `xhost` documentation). Furthermore, `xterm` must be accessible via your shell's directory search `PATH`.

Note that the EMTS server does not check that an X server is actually running on your specified machine. The effects of your `xterm` process failing to contact an X server are not defined by the EMTS (but your client process will almost certainly fail!). Refer to the source of the `xdemo` example program (section 4.2.4) for the sort of tests that could be made to verify the presence of a valid X server.

4.2.3 Library extensions

4.2.3.1 Keyboard handling

The EMTS server supports two extensions to the Inmos SP protocol:

- the addition of an `EMTS.READCH` (cf `SP_GETKEY`) message to allow non-echoed reading of a character from an `xterm` window (`SP_GETKEY` reads from the standard input stream);

- the addition of an `EMTS.POLLCH` (cf `SP_POLLKEY`) message to allow polling for terminal input from an `xterm` window (`SP_POLLKEY` polls the standard input stream).

Two corresponding library procedures (mt.readch and mt.pollch) are provided in the EMTS library \examples\bin\emtslib.lib for the TA class of transputers. The occam source \examples\occam\emtslib.occ together with the corresponding header file \examples\occam\emtslib.inc are also provided.

4.2.3.2 Stream i/o

A number of additional library routines are also provided with the EMTS library to support the Inmos stream-based input/output model:

- mt.keystream.from.kbd.readch (cf so.keystream.from.kbd) generates a keystream from an xterm window (or the standard input stream) using the EMTS.READCH protocol extension. This procedure should only be used if the multi-threading server extension is enabled (otherwise it will block the server).

- mt.keystream.from.kbd.pollch (cf so.keystream.from.kbd) generates a keystream from an xterm window (or the standard input stream) using the EMTS.POLLCH protocol extension. This procedure can be used if the multi-threading server extension is disabled.

- mt.scrstream.to.ANSI (cf so.scrstream.to.ANSI) generates an ANSI scrstream to an xterm window (or the standard output stream).

4.2.4 The xdemo example program

An example program \examples\bin\xdemo.bah is provided with the distribution and is runnable on the TA class of transputers. The occam source \examples\occam\xdemo.occ is also provided. The program illustrates use of the xterm server extension.

The program comprises a single process which checks for a valid X server, opens an xterm window, performs a number of terminal i/o operations, and then closes the window. The program accepts an optional argument string which is passed to the xterm process.

The following examples assume that the stub name has been set using an environment variable.

To run the program without the optional argument string, you would use:

```
example% eserver -se -sb xdemo.bah
```

The following invocation passes an argument string to the xterm process:

```
example% eserver -se -sb xdemo.bah "-title xdemo -display jupiter:0.0"
```

4.2.5 The mtstream example program

An example program `\examples\bin\mtstream.bah` is provided with the distribution and is runnable on the TA class of transputers. The occam source `\examples\occam\mtstream.occ` is also provided. The program illustrates use of the stream i/o model using the EMTS library procedures.

The program consists of a `root` process which prints a few messages to the screen and then terminates the program. It runs in parallel with an `mt.driver` process (section 4.4.4), an `mt.key-stream.from.kbd.{readch, pollch}` process and an `mt.scrstream.to.ANSI` process.

To run the program, you would use (assuming that the stub name has been set using an environment variable):

```
example% eserver -se -sm -sb mtstream.bah
```

4.3 Support for system processes

4.3.1 Overview

The motivation for providing this extension was to allow a transputer process to interact with an operating system process in a natural fashion. For example, you may want a transputer process to process data generated by an operating system process, or vice versa. The transputer process reads from the standard output stream and writes to the standard input stream of the `system` process.

4.3.2 Programming guide

Access to the `system` server extension has been provided using the standard Inmos i/o libraries. To spawn a `system` process, a client transputer process opens a "file" with the special name:

```
SYSTEM:[<sep>:]<prog>[options to <prog> [separated by <sep>]]
```

Note that case is important. `<sep>` is an optional group of one or more characters, any subset of which is interpreted by the server as an option separator when parsing the string (see below). `<prog>` is the name of the program to be run by the spawned process.

The client is handed back a file descriptor. Writing to the file descriptor causes data to be written to the standard input stream of the `system` process. Reading from the file descriptor causes data to be read from the `system` process. Closing the file descriptor causes the `system` process to be terminated.

By default, the server uses a single space as the option separator when parsing the string. However, there may be occasions when a single space needs to part of an option (in which case a single space cannot be used as the option separator). Therefore, the server allows

a custom separator of up to 10 characters to be specified in the string. Any subset of the separator characters is interpreted as an option separator.

For example, to run the Unix command `/bin/ls -al`, you would open a "file" like:

```
SYSTEM:/bin/ls -al
```

To run the Unix command `ls -al > /tmp/ls.log`, you could open a "file" like:

```
SYSTEM:%:csh%-c%/bin/ls -al > /tmp/ls.log
```

using `%` as the option separator.

Alternatively, you could open the "file":

```
SYSTEM:csh -c /bin/ls<TAB>-al<TAB>><TAB>/tmp/ls.log
```

where `<TAB>` is the tab character.

4.3.3 The system example program

An example program `\examples\bin\system.bah` is provided with the distribution and is runnable on the TA class of transputers. The occam source `\examples\occam\system.occ` is also provided. The program illustrates use of the `system` server extension.

The program comprises a single process which reads the command-line for the name of the program to run (plus optional arguments) and then spawns a `system` process to run the program. It reads the standard output stream of the process and prints it to the terminal.

The following examples assume that the stub name has been set using an environment variable.

To run the Unix command `ls -al`, you would use:

```
example% eserver -se -sb system.bah "ls -al"
```

To run the Unix command `ls -al > /tmp/ls.log`, you could use:

```
example% eserver -se -sb system.bah "%:csh%-c%/bin/ls -al > /tmp/ls.log"
```

Note that you must put the option string in quotes if it contains any `-` characters. This is because of the way the example program parses the command-line. For example, the following would not work:

```
example% eserver -se -sb system.bah %:ls%-al
```

Furthermore, you must have at least one space or tab character in a quoted string that you pass to the example program. This is because of the way the server parses the command-line. For example, the following would not work either:

```
example% eserver -se -sb system.bah "%:ls%-al"
```

4.4 Multi-threading

4.4.1 Overview

The Inmos `iserver` is a single-threaded transputer server program. An incoming request packet has to be serviced and a reply packet transmitted before the next request packet can be processed. A request that causes the server to block (eg waiting for a key to be pressed on the keyboard) will delay any other transputer client processes that require access to the server.

The transputer i/o libraries employ a "single-threaded request-reply" protocol for communication with the server. A transputer client sends a request packet to the server and then waits until it receives a reply packet before proceeding. If a process is multiplexing request packets on behalf of several clients, a blocking i/o request from one client will block out all other clients because the multiplexor will not proceed until it receives the reply packet from the server. This communications paradigm is consistent with the single-threaded nature of the `iserver`.

4.4.2 Multi-threaded server

Because the `iserver` supports only a single (physical) terminal, the single-threaded nature of the server and transputer i/o libraries is not too serious a limitation. The EMTS `eserver`, on the other hand, supports multiple (pseudo) terminals in the form of `xterm` windows. A single-threaded `eserver` would mean that a blocking i/o request to one pseudo-terminal would block client access to all other pseudo-terminals (and other i/o resources such as the filing system). Likewise, a blocking i/o request to a `system` process would block the entire server.

For this reason, the `eserver` has been programmed to use multiple threads and to support non-blocking system calls. Each request packet is handled by a separate thread. If an i/o request on a file descriptor cannot be serviced immediately, only the thread handling the request is blocked; all other threads are unaffected.

4.4.3 The MT protocol

In order to exploit the multi-threaded functionality of the server, the single-threaded request-reply protocol used by transputer clients has to be replaced by a multi-threaded request-reply protocol. This communications paradigm allows multiple server transactions to be in progress simultaneously. If one server transaction is blocked, other transactions can still proceed.

The EMTS software uses the MT (multi-threaded) protocol for multi-threaded communications with the server. The MT protocol is a very simple extension to the Inmos SP protocol: every packet

includes an additional byte (client identifier) that identifies the owner of the packet. The MT packet format is shown below:

```
<header>:<body>:<client  identifier>
```

The `<header>` is the standard 2-byte SP header which encodes the length of the body. The `<body>` of the packet contains up to 1024 bytes of user-defined data consistent with the SP protocol. The `<client  identifer>` is a 1-byte identifer used to define the owner of the packet. The MT packet format is equivalent to:

```
<SP  packet>:<client  identifier>
```

where `<SP  packet>` is data in the SP packet format.

4.4.4 The MT driver process

A sample `mt.driver` procedure is available in the library `\examples\bin\emtslib.lib` provided with the distribution. It is compatible with the TA class of transputers. The source of the library `\examples\occam\emtslib.occ` is also provided. The driver multiplexes SP request packets from multiple client processes transmitting them to the server as MT packets with the appropriate client identifier. The driver demultiplexes MT reply packets from the server, sending them to the appropriate client (determined by the client identifer) as SP packets. From the client process's point of view, the MT driver looks like a standard SP multiplexor.

The sample MT driver uses the channel array index on which an SP request arrives as the client identifier for the MT packet. If you have a transputer network in which processes on different transputers require access to the server using the MT protocol, you need to run an MT driver on each of the transputers and multiplex their MT channels onto the server MT channel using an MT multiplexor process running on the root transputer. In this case, you would need to define unique client identifiers for each client process in your network and map between these client identifers and appropriate channels using (custom) MT multiplexor and MT driver processes.

4.4.5 Benefits of the MT protocol

The MT protocol is only of benefit to those transputer programs that need to exploit the multi-threaded functionality of the server. For example, if you have a multi-process program in which multiple processes open private `xterm` windows and request keyboard input from each, you may find it useful to use the MT protocol. If, on the other hand, multiple processes in your program only write output to their private `xterm` windows, the MT protocol is probably unnecessary. The `xterm` and `system` extensions are still available to your transputer clients regardless of whether or not they are programmed to use the MT protocol.

4.4.6 Using the MT protocol

Because the MT packet format is different from the SP packet format, the MT protocol is not compatible with the SP protocol. By default, the stub uses the SP protocol when communicating

with a target. In order to support transputer programs that use the MT protocol, you must start the server with the `-sm` command-line option (section 3.3.1.5).

4.4.7 The mtdemo example program

An example program `\examples\bin\mtdemo.bah` is provided with the distribution and is runnable on the TA class of transputers. The occam source `\examples\occam\mtdemo.occ` is also provided. The program illustrates use of the `xterm` and `multi-threading` server extensions.

The body of the program comprises three processes which run in parallel. `Client0` prints a message and prompts for keyboard input. This is repeated a number of times before the process terminates the program. `Client1` and `client2` each print a message to the screen every second.

The initial section of the program parses the command line looking for options of the form:

`[-client0] [client0args] [-client1] [client1args] [-client2] [client2args] [-x]`

`Client0` is started by supplying the `-client0` command-line option; likewise for `client1` and `client 2`. If the `-x` option is given, each client opens a private `xterm` window. Each client then reads keyboard input from its private terminal and writes screen output to its private terminal. Because the program uses the MT protocol, the blocking nature of `client0` does not affect the other two clients.

If the `-x` option is not given, each client reads keyboard input from the server's controlling terminal, and writes output to the server's controlling terminal. Output from the three processes is multiplexed onto the screen.

If the `client0args` option is given, string is passed to the `client0 xterm` process (only if the `-x` option is given). The `client1args` and `client2args` options have the same effect on the `client1` and `client2 xterm` processes (only if the `-x` option is given).

The following examples assume that the stub name has been set using an environment variable.

To start the server on your Sun and boot the target with the example program, issue:

`example%` **`eserver -se -sm -sb mtdemo.bah -client0 -client1 -client2 -x`**

This should cause three `xterm` windows to appear on your screen. Follow the instructions generated by `client0` and the program will terminate correctly.

You might like to supply some optional options by using a command such as:

```
example% eserver -se -sm -sb mtdemo.bah -client0 "-title client0" -client1 "-display jupiter:0.0" -x
```

4.4.8 The mtshell example program

An example program `\examples\bin\mtshell.bah` is provided with the distribution and is runnable on the TA class of transputers. The occam source `\examples\occam\mtshell.occ` is also provided. The program has been contrived to illustrate the use of all three server extensions.

The body of the program comprises two processes which run in parallel. The `xterm.reader` process reads input from an `xterm` window and writes it to the standard input stream of a `system` process running a Unix C shell. The `shell.reader` process reads from the standard output stream of the shell process and writes it to the `xterm` window. The program exploits the multi-threaded nature of the server.

The following example assumes that the stub name has been set using an environment variable.

To start the server on your Sun and boot the target with the example program, issue:

```
example% eserver -se -sm -sb mtshell.bah
```

This should cause a single `xterm` window to appear on your screen. Commands typed in the window are processed by a Unix shell; the results are written back to your window. Type `CTRL-D` or `exit` to terminate the program.

5 System Implications and Limitations

5.1 Performance

The performance of the EMTS cannot match that of the standard `iserver` configuration in which the server runs on the same machine that hosts the transputer target. This is because every SP request and reply packet has to be relayed between server and stub over Ethernet. EMTS performance is influenced by:

- the speed of your PC;
- the speed of your PC network interface;
- the number of target transputer systems that the stub is serving[5];

---

5 *The stub uses a strict round-robbin scheduling policy when communicating with multiple targets to ensure that no one target gets more than its fair share of processing time.*

- the load on your network;
- the speed of your server machine;
- the speed of your server's network interface.

Transputer programs that are heavily i/o intensive or that are highly interactive may not run satisfactorilly on the EMTS if they use the standard transputer i/o libraries. For example, the performance of transputer-hosted editors (the F editor or the TDS3 editor) is less than ideal on slow workstations; every cursor key press suffers the round-trip delay of an Ethernet packet going from server to stub to server before it is echoed to the screen. Having said this, the transputer-hosted editors are still quite usable with the EMTS (the F editor was used to prepare the example programs on a Sun4), although server-hosted editors may be preferred for serious use.

Transputer programs that exploit the multi-threaded nature of the `eserver` can achieve *improved* i/o performance under some circumstances. This is because multiple i/o requests can be queued on the same file descriptor thereby keeping the server continuously busy and removing the effects of network latency due to the request-reply round-trip delay.

Note also that the standard Inmos i/o library routines use a maximum buffer size of 512 bytes, even though the SP protocol supports 1024 byte packets. The performance of programs that do a large amount of file i/o would benefit from using a larger buffer size.

Transputer applications that use the Inmos stream-based input/output model typically run in parallel with a process that polls the server for keyboard input. This can have drastic performance repercussions in the EMTS environment since the keyboard is effectively being polled across the network. Better performance can be achieved with a multi-threaded server by using `EMTS.READCH` or `SP_GETKEY` instead of `EMTS.POLLCH` or `SP_POLLCH`. Refer to section 4.2.3.2.

5.2 Multiple users

The EMTS allows multiple users to share the transputer resources of a single (networked) PC. Each user either implicitly or explicitly selects one of the target transputer systems hosted by the PC at start of day. If all targets are busy when a user starts the server, the server will terminate with an appropriate error message.

5.3 Debugging in a multi-user environment

If you are running the EMTS in a multi-user environment, you may run into a problem when it comes to debugging a target. The problem is as follows. If you do not explicitly select a target, the stub selects an arbitrary free target on your behalf (running the server in `verbose` mode causes it to print a message informing you which target you are using). If your program crashes, the stub terminates the session and your server exits with a suitable error message. If you then wish to debug your program, you must connect *to the same target*; if you do not

explicitly select the same target, the stub may allocate you a different free target making it impossible for you to debug your program.

A possible solution to this problem is to ensure that each user of a multi-user system always connects to a globally agreed (unique) target. This is not possible, however, if there are more potential users than targets.

5.4 MS-DOS support

The server does not support MS-DOS server extensions.

5.5 Server version numbers

The EMTS server version number returned as the result of an `SP_ID` message will be equal to or greater than 100. Version `n.m0` of the server will return `nm0 + k`. If `server extensions` are enabled, the least significant digit (`k`) will be 1. If `multi-threading` is also enabled, the least significant digit (`k`) will be 2. If neither are enabled, the least significant digit (`k`) will be 0. For example, version 1.00 of the server with `server extensions` enabled will return 101.