CBP Runtime Module

(CRAY T90™ Series)

HDM-071-A

Cray Research Proprietary

Cray Research, Inc.

Record of Revision

REVISION	DESCRIPTION
	September 1995. Original printing.
A	March 1996. This revision corresponds to the MT-T2.2.0 offline diagnostic release.

Any shipment to a country outside of the United States requires a letter of assurance from Cray Research, Inc.

Requests for copies of Cray Research, Inc. publications should be directed to:

CRAY RESEARCH, INC.
Customer Service Logistics
1100 Lowater Road
Chippewa Falls, WI 54729

Comments about this publication should be directed to:

CRAY RESEARCH, INC.
Service Publications and Training
890 Industrial Blvd.
Chippewa Falls, WI 54729

CBP RUNTIME MODULE

Figures

Tables

Overview

The Command Buffer Parser (CBP) application contains different runtime modules used to troubleshoot the different types of hardware that CBP supports. This document describes the CRAY T90 series CBP runtime module that enables you to automate troubleshooting tasks performed by the Mainframe Maintenance Environment (MME) environment 1, MME environment 2, Logic Monitor Environment (LME), and System Configuration Environment (SCE) for CRAY T90 series mainframes.

This document includes the procedure to start CBP with the CRAY T90 series CBP runtime module, the command buffer programs that are available for use with the CRAY T90 series CBP runtime module, and the CBP commands that are specific to the CRAY T90 series CBP runtime module.

For general information about the CBP application, including descriptions of the user interface, CBP programming, and general-purpose commands, refer to the *Command Buffer Parser User Guide*, publication number HDM-076-0.

Starting the CRAY T90 Series CBP Runtime Module

Choose **Utilities –> Command Buffer** in the MME environment 1 base window, MME environment 2 base window, or LME base window to start CBP with the CRAY T90 series CBP runtime module.

NOTE: You cannot start the CRAY T90 series CBP runtime module from MME environment 0. The CRAY T90 series CBP runtime module does not support MME environment 0.

Command Buffer Programs and Files

You can access several command buffer programs specific to the CRAY T90 series CBP runtime module with the Dir: menu button in the CBP: Load/Save window. Figure 1 shows this menu structure.

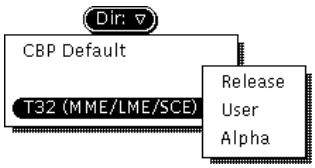

Figure 1. Dir Menu Structure

The following menu options are available:

Menu Option	Description
Release	CRAY T90 series command buffer programs that are included in the current offline diagnostic release.
User	CRAY T90 series command buffer programs that you modify or create and then save.
Alpha	CRAY T90 series command buffer programs that are not officially released.

CRAY T90 Series CBP Commands

The CRAY T90 series CBP runtime module includes active CBP commands that perform functions specific to the CRAY T90 series MME, SCE, and LME applications.

The following conventions are used in the command descriptions:

- *Italic* type indicates a variable.
- Square brackets [ ] indicate an optional entry.
- A vertical bar | indicates a choice.

Table 1 lists the `include` statements necessary for some CBP operations.

Table 1. CBP `include` Statements

`include` Statement	Description
`#include "t32/rel/cmd/mme_std.h"`	This include file contains standard function definitions.
`#include "t32/rel/cmd/mme_std_def.h"`	This include file contains the standard variable definitions.
`#include "t32/rel/cmd/T90_chips.h"`	This include file contains the definitions needed to reference specific chips and test points using the LME commands available from within CBP.
`#include "t32/rel/cmd/T90.h"`	This include file contains the constant definitions for the CRAY T90 series CBP command set (LME, MME, and SCE).
`#include "t32/rel/cmd/T90_mods.h"`	This include file contains the definitions needed to reference specific modules using LME commands available from within CBP.

MME-specific Command Descriptions

Table 2 lists the MME-specific commands that you can use in command buffer programs to manipulate the MME interface and to automate CRAY T90 series mainframe testing. Descriptions of the options for each command follow the table.

Table 2. MME-specific Commands

Command	Description
`MMEAlloc( )`	Sets up resource allocation options
`MMEAssign( )`	Assigns a control point to a specific CPU or to all CPUs
`MMECtrlptGet( )`	Returns status information about a specific executing control point
`MMECtrlptInfo( )`	Returns configuration information about a specific control point
`MMECtrlptList( )`	Returns the next control point in the list of loaded control points, the next active control point, or the next control point that has detected an error
`MMECtrlptSectionInfo( )`	Returns configuration information about a specific control point section
`MMECtrlptSetup( )`	Sets configuration information for a specific control point or for all current control points
`MMEDeassign( )`	Deassigns a specific CPU from the control point to which it is assigned or deassigns all CPUs assigned to a specific control point
`MMEError( )`	Returns a value that indicates whether a specific control point or any control point has an error
`MMEGet( )`	Returns current MME status information
`MMEGo( )`	Starts a specific control point or all loaded control points
`MMEHalt( )`	Stops a specific control point or all executing control points
`MMELoad( )`	Loads a control point
`MMEPartitionGet( )`	Returns information about the partition in which MME is running
`MMERead( )`	Reads a block of data from mainframe memory
`MMEReload( )`	Reloads a specific control point or all control points
`MMEReset( )`	Resets MME to the current environment or environment 1 or 2
`MMETimeout( )`	Returns a value that indicates whether a specific control point or all control points timed out
`MMEUnload( )`	Unloads a specific control point or all control points
`MMEWait( )`	Waits a specified number of seconds and returns a value that indicates whether any control points reached time-out values or found an error
`MMEWrite( )`	Writes a block of data to mainframe memory

tag Parameter Description

The *tag* parameter, which is used for several MME-specific CBP commands, refers to an integer value called a control point tag that was returned by a previous `MMELoad( )` or `MMECtrlptList( )` command. Use this value in an MME-specific CBP command to specify which control point you want the command to reference.

MMEAlloc( )

`MMEAlloc(CACHE,`*cpu*`|ALL,ENABLE|DISABLE);`

This command enables or disables the scalar cache for the CPU specified by *cpu* or for all CPUs if you specify `ALL`.

`MMEAlloc(CPUMODE,AUTO|MANUAL);`

This command enables or disables automatic CPU assignment when you load a control point.

`MMEAlloc(IOICM,`*cpu*`|ALL,ENABLE|DISABLE);`

This command enables or disables the interrupt on correctable memory error (ICM) option for the CPU specified by *cpu* or for all CPUs if you specify `ALL`; this option sets or clears the ICM flag in the exchange package for the specified CPU(s).

`MMEAlloc(IOCPU,`*cpu*`);`

This command sets the I/O CPU (the CPU path used to write to memory and read from memory) to *cpu*; *cpu* can be a constant or variable.

`MMEAlloc(IOIRP,`*cpu*`|ALL,ENABLE|DISABLE);`

This command enables or disables the interrupt on register parity error (IRP) option for the CPU specified by *cpu* or for all CPUs if you specify `ALL`; this option sets or clears the IRP flag in the exchange package for the specified CPU(s).

`MMEAlloc(IOIUM,`*cpu*`|ALL,ENABLE|DISABLE);`

This command enables or disables the interrupt on uncorrectable memory error (IUM) option for the CPU specified by *cpu* or for all CPUs if you specify `ALL`; this option sets or clears the IUM flag in the exchange package for the specified CPU(s).

`MMEAlloc(MEMDELAY,`*cpu*`|ALL,0|4|16|63);`

This command delays sending the signal to common memory that indicates that a CPU is ready to accept another word of data. You can set a delay of 0, 4, 16, or 63 clock periods for the CPU specified by *cpu* or for all CPUs if you specify `ALL`.

For more information about this delay, refer to the "Memory Input Ports Maintenance Functions" description in Section 4 of the *Triton Maintenance System Engineering Note*, publication number PRN-0957.

`MMEAlloc(MEMMODE,BOTTOM_UP|RANDOM|TOP_DOWN);`

This command sets the memory mode to bottom up, random, or top down.

`MMEAlloc(MEMMODE,PARTITION_DOWN,COUNT,`*count*`);`

This command sets the memory mode to top down with the number of partitions specified by *count*.

`MMEAlloc(MEMMODE,PARTITION_DOWN,SIZE,`*size*`);`

This command sets the memory mode to top down with partitions that have the size specified by *size*.

`MMEAlloc(MEMMODE,PARTITION_UP,COUNT,`*count*`);`

This command sets the memory mode to bottom up with the number of partitions specified by *count*.

`MMEAlloc(MEMMODE,PARTITION_UP,SIZE,`*size*`);`

This command sets the memory mode to bottom up with partitions that have the size specified by *size*.

`MMEAlloc(SBCDBD,`*cpu*`|ALL,ENABLE|DISABLE);`

This command enables or disables the single-byte correction/ double-byte detection (SBCDBD) feature for the CPU specified by *cpu* or for all CPUs if you specify `ALL`.

`MMEAlloc(SECDED,`*cpu*`|ALL,ENABLE|DISABLE);`

This command enables or disables the single-error correction/ double-error detection (SECDED) feature for the CPU specified by *cpu* or for all CPUs if you specify `ALL`.

`MMEAlloc(SPARECHIP,DEFAULT);`

This command resets the current spare-chip table to the spare-chip table specified in SCE.

`MMEAlloc(SPARECHIP,MERGE,`*upper_code*`,`*lower_code*`);`

This command takes the bad bit codes that you have specified and merges them with the spare-chip table that is specified in the SCE configuration. Table 3 shows the bits you can select with the *upper_code* and *lower_code* parameters.

`MMEAlloc(SPARECHIP,OVERRIDE,`*upper_code*`,`*lower_code*`);`

This command overrides the spare-chip table specified in SCE. This command writes a spare-chip table that uses the bad bits you specify with the *upper_code* and *lower_code* parameters. Table 3 shows the bits you can select with the *upper_code* and *lower_code* parameters.

Table 3. Available Upper and Lower Bad Bit Codes

Bad Bit Code	Bits Selected for Upper Bad Bit Code	Bits Selected for Lower Bad Bit Code
000	48 and 32	16 and 0
001	49 and 33	17 and 1
002	50 and 34	18 and 2
003	51 and 35	19 and 3
004	52 and 36	20 and 4
005	53 and 37	21 and 5
006	54 and 38	22 and 6
007	55 and 39	23 and 7
010	56 and 40	24 and 8
011	57 and 41	25 and 9
012	58 and 42	26 and 10
013	59 and 43	27 and 11
014	60 and 44	28 and 12
015	61 and 45	29 and 13
016	62 and 46	30 and 14
017	63 and 47	31 and 15
020	73 and 70	67 and 64
021	74 and 71	68 and 65

Table 3. Available Upper and Lower Bad Bit Codes (continued)

Bad Bit Code	Bits Selected for Upper Bad Bit Code	Bits Selected for Lower Bad Bit Code
022	75 and 72	69 and 66
037	None	None

MMEAssign( )

`MMEAssign(`*tag*`,`*cpu*`|ALL);`

This command assigns the CPU specified by *cpu* to the control point referenced by *tag*. If you specify `ALL`, all CPUs are assigned to the control point.

For more information about the *tag* parameter, refer to “tag Parameter Description” on page 7.

MMECtrlptGet( )

NOTE: There are two ways to access the values that the `MMECtrlptGet( )` commands return:

- You can set a variable to the result of the `MMECtrlptGet( )` command. For example:

```
base = MMECtrlptGet(cp1,BASE_ADDR);
limit = MMECtrlptGet(cp1,LIMIT_ADDR);
```

- You can include the variables in the `MMECtrlptGet( )` command. For example:

```
MMECtrlptGet(cp1,BASE_ADDR,base,LIMIT_ADDR,
limit);
```

`MMECtrlptGet(`*tag*`,BASE_ADDR);`

This command returns the base address (the address where the control point is loaded in mainframe memory) of the control point referenced by *tag*.

For more information about the *tag* parameter, refer to “tag Parameter Description” on page 7.

`MMECtrlptGet(`*tag*`,CPU_SELECT);`

This command returns a mask of the CPUs that are assigned to the control point referenced by *tag*.

For more information about the *tag* parameter, refer to “tag Parameter Description” on page 7.

`MMECtrlptGet(`*tag*`,CURRENT_CPU);`

This command returns the number of the CPU that is currently running the control point referenced by *tag*.

For more information about the *tag* parameter, refer to “tag Parameter Description” on page 7.

`MMECtrlptGet(`*tag*`,CURRENT_SECTION);`

This command returns the number of the section that is currently executing for the control point referenced by *tag*.

For more information about the *tag* parameter, refer to “tag Parameter Description” on page 7.

`MMECtrlptGet(`*tag*`,ERROR_COUNT);`

This command returns the current global error count for the control point referenced by *tag*.

For more information about the *tag* parameter, refer to “tag Parameter Description” on page 7.

`MMECtrlptGet(`*tag*`,IS_ACTIVE);`

This command returns a 1 if the control point referenced by *tag* is active and returns a 0 if the control point is not active.

For more information about the *tag* parameter, refer to “tag Parameter Description” on page 7.

`MMECtrlptGet(`*tag*`,LIMIT_ADDR);`

This command returns the limit address for the control point referenced by *tag*.

For more information about the *tag* parameter, refer to “tag Parameter Description” on page 7.

`MMECtrlptGet(`*tag*`,PASS_COUNT);`

This command returns the current global pass count for the control point referenced by *tag*.

For more information about the *tag* parameter, refer to “tag Parameter Description” on page 7.

`MMECtrlptGet(`*tag*`,SECTION_ENV(`*section_number*`));`

This command returns a bit mask of the environments in which the section runs for the section specified by *section_number* of the control point referenced by *tag*.

If bit 0 of the returned bit mask is set to 1, the section runs in environment 1. If bit 1 of the returned bit mask is set to 1, the section runs in environment 2. If both bits of the returned bit mask are set to 1, the section runs in environments 1 and 2.

If bit 0 of the returned bit mask is set to 0, the section does not run in environment 1. If bit 1 of the returned bit mask is set to 0, the section does not run in environment 2.

For more information about the *tag* parameter, refer to “tag Parameter Description” on page 7.

`MMECtrlptGet(`*tag*`,SECTION_MODE(`*section_number*`));`

This command returns the mode of the section specified by *section_number* for the control point referenced by *tag*.

This command returns `SCPU_ONLY` to indicate that the section runs in a single CPU, `MCPU_ONLY` to indicate that the section runs in multiple CPUs, or `SCPU_MCPU` to indicate that the section runs in a single CPU or in multiple CPUs.

For more information about the *tag* parameter, refer to “tag Parameter Description” on page 7.

`MMECtrlptGet(`*tag*`,SECTION_NAME(`*section_number*`));`

This command returns the name of the section specified by *section_number* for the control point referenced by *tag*.

For more information about the *tag* parameter, refer to “tag Parameter Description” on page 7.

`MMECtrlptGet(`*tag*`,SECTION_SELECT);`

This command returns a mask that indicates the sections that are selected for the control point referenced by *tag*.

For more information about the *tag* parameter, refer to “tag Parameter Description” on page 7.

MMECtrlptInfo( )

`MMECtrlptInfo(`*tag*`,` *section_select*`,` *cpu_select*`,` *pass_count*`,` *error_count*`);`

This command returns information about the control point referenced by *tag*. All parameters following *tag* are used to return values.

The `MMECtrlptInfo( )` command returns the section select mask, CPU select mask, pass count, and error count to the variables in the *section_select*, *cpu_select*, *pass_count*, and *error_count* positions. You may use any valid variable to receive these values. If you do not need a return value, set the appropriate parameter to 0 or `NULL`.

MMECtrlptList( )

`MMECtrlptList(`*tag*`);`

This command returns the control point tag for the next control point in the list of loaded control points. You can use the returned control point tag in other MME-specific commands to manipulate the control point.

You can start at a specific control point in the control point list by specifying a control point tag with the *tag* parameter. When you use the *tag* parameter, this command returns the control point tag of the next control point in the list of loaded control points. If you specify 0 or `NULL` for the *tag* parameter, this command returns the control point tag for the first control point in the list of loaded control points.

The following code example shows how you could use this command with the `while( )` command to step through the list of loaded control points.

```
td = NULL;
while ((td = MMECtrlptList(td)) != NULL)
{
    base = MMECtrlptGet(td, BASE_ADDR);
    lim = MMECtrlptGet(td, LIMIT_ADDR);
    /* commands that use these values */
}
```

NOTE: The list of control points contains the control points in the order in which they were loaded.

`MMECtrlptList(`*tag*`,ACTIVE);`

This command returns the control point tag for the next active control point in the list of loaded control points. You can use this control point tag in other MME-specific commands to manipulate the control point.

You can start at a specific control point in the control point list by specifying a control point tag with the *tag* parameter. When you use the *tag* parameter, this command returns the control point tag of the next active control point. If you specify 0 or `NULL` for the *tag* parameter, this command returns the control point tag for the first active control point.

NOTE: The list of control points contains the control points in the order in which they were loaded.

`MMECtrlptList(`*tag*`,ERROR);`

This command returns the control point tag for the next control point that has an error in the list of loaded control points. You can use this control point tag in other MME-specific commands to manipulate the control point.

You can start at a specific control point in the control point list by specifying a control point tag with the *tag* parameter. When you use the *tag* parameter, this command returns the control point tag of the next control point that has an error. If you specify 0 or `NULL` for the *tag* parameter, this command returns the control point tag for the first control point that has an error.

NOTE: The list of control points contains the control points in the order in which they were loaded.

MMECtrlptSectionInfo( )

`MMECtrlptSectionInfo(`*tag*`,`*section_select*`,`*cpu_select*`,`*pass_count*`,`*error_count*`);`

This command returns information about a specific section of a control point executing in a specific CPU. The *section_select* parameter specifies which section of the control point is referenced by *tag*. The *cpu_select* parameter specifies from which CPU to take the information.

The `MMECtrlptSectionInfo( )` command returns the pass count and error count to the variables in the *pass_count* and *error_count* positions. You may use any valid variable to receive these values. If you do not need a return value, set the appropriate parameter to 0 or `NULL`.

For more information about the *tag* parameter, refer to "tag Parameter Description" on page 7.

MMECtrlptSetup( )

`MMECtrlptSetup(`*tag*`|ALL,`*section_mask*`|ALL_SECTIONS, PASS,`*pass_count*`);`

This command sets the section select and the pass count for the control point referenced by *tag* or for all current control points.

The *section_mask* parameter contains a mask of sections to be selected; use 0 to indicate no change from the current sections selected and `ALL_SECTIONS` to select all sections of the control point. The control point executes until it reaches the pass count specified by the *pass_count* parameter.

For more information about the *tag* parameter, refer to "tag Parameter Description" on page 7.

`MMECtrlptSetup(`*tag*`|ALL,`*section_mask*`|ALL_SECTIONS, TIME,`*time_limit*`);`

This command sets the section select and the time limit for the control point referenced by *tag* or for all current control points.

The *section_mask* parameter contains a mask of sections to be selected; use 0 to indicate no change from the current sections selected and `ALL_SECTIONS` to select all sections of the control point. The *time_limit* parameter specifies the amount of time (in seconds) that you want the control point to execute.

For more information about the *tag* parameter, refer to “tag Parameter Description” on page 7.

MMEDeassign( )

`MMEDeassign(`*cpu*`);`

This command deassigns the CPU specified by *cpu* from the control point to which the CPU is assigned.

`MMEDeassign(ALL,`*tag*`);`

This command deassigns all CPUs assigned to the control point referenced by *tag* or deassigns all CPUs that are assigned to control points if you specify `ALL`.

For more information about the *tag* parameter, refer to “tag Parameter Description” on page 7.

MMEError( )

`MMEError(`*tag*`|ALL);`

This command returns a nonzero (true) value if the control point referenced by *tag* (or any control point if you specify `ALL`) has an error; this command returns a zero (false) value if the control points do not have errors.

For more information about the *tag* parameter, refer to “tag Parameter Description” on page 7.

MMEGet( )

```
MMEGet(ENV);
```

This command returns the environment in which MME is currently operating (environment 1 or 2).

For example, `curr_environ = MMEGet(ENV);` sets the `curr_environ` variable to 1 if MME is operating in environment 1.

MMEGo( )

`MMEGo(`*tag*`|ALL);`

This command starts the control point referenced by *tag* or all loaded control points if you specify `ALL`. If no conditions were previously set for the specified control points by the `MMECtrlptSetup( )` or `MMEGo( )` commands, the pass and error counts default to –1. This causes the control point to run indefinitely.

For more information about the *tag* parameter, refer to “tag Parameter Description” on page 7.

`MMEGo(`*tag*`|ALL,`*section_mask*`|ALL_SECTIONS,PASS|TIME,`*value*`);`

This command performs the `MMECtrlptSetup( )` command and then starts the control point referenced by *tag* or all loaded control points. [Refer to the description of the `MMECtrlptSetup( )` command on page 15 for more information.]

For more information about the *tag* parameter, refer to “tag Parameter Description” on page 7.

MMEHalt( )

`MMEHalt(`*tag*`|ALL[,NO_DUMP|EXCHANGE_DUMP|REGISTER_DUMP]);`

This command stops the control point referenced by *tag* or stops all control points if you specify `ALL`. The optional parameter specifies the halt dump mode; if you omit this parameter, no dump occurs.

For more information about the *tag* parameter, refer to “tag Parameter Description” on page 7.

MMELoad( )

`MMELoad(“`*filename*`”[,ALL]);`

This command loads the control point contained in the file specified by *filename*. This command returns an integer value called a *tag*, which you use with other commands to reference the control point. The optional `ALL` parameter assigns all unassigned CPUs to the control point.

For more information about the *tag* parameter, refer to “tag Parameter Description” on page 7.

You can use the following shortcut options to specify the directory that contains the file you want to load. Replace the *major* and *minor* variables with the highest common major and minor revision values from SCE.

Shortcut Options	Directory
`USR`	`/cri/cme/t32/usr/diag`
`DIAG`	`/cri/cme/t32/rel/diag.cp`*major.minor*
`ALPHA`	`/cri/cme/t32/alpha/diag.cp`*major.minor*
`UTIL`	`/cri/cme/t32/rel/util.cp`*major.minor*

To use the shortcut options, follow the convention of `MMELoad(“`*shortcut filename*`”);`. For example, `MMELoad(“USR/my01.t”);` will load `/cri/cme/t32/usr/diag/my01.t`.

`MMELoad(“`*filename*`”,`*base*`,`*limit*`);`

This command loads the control point contained in the file specified by *filename*. MME loads the control point at the address in memory specified by the *base* parameter. The *limit* parameter specifies the last memory address that MME will allocate to the control point. MME allocates all memory from *base* to *limit* for the control point.

MMEPartitionGet( )

`MMEPartitionGet(LOSP_IOS_MASK);`

This command checks the logical partition in which MME is running for LOSP channels that are enabled and connected to an external device (for example, an IOS). This command returns a bit mask in which bit 0 represents LOSP channel pair 100/101; bit 1 represents LOSP channel pair 102/103; etc.

`MMEPartitionGet(LOSP_LOOP_MASK);`

This command checks the logical partition in which MME is running for LOSP channels that are enabled and are in loopback (either internal or external) mode. This command returns a bit mask in which bit 0 represents LOSP channel pair 100/101; bit 1 represents LOSP channel pair 102/103; etc.

`MMEPartitionGet(NUM_USABLE_CPUS);`

This command checks the logical partition in which MME is running for available CPUs and returns the number of available CPUs.

`MMEPartitionGet(USABLE_CPUS_MASK);`

This command checks the logical partition in which MME is running for available CPUs. This command returns a mask in which bit 0 represents CPU 00; bit 1 represents CPU 01; etc.

`MMEPartitionGet(VHISP_MASK);`

This command checks the logical partition in which MME is running for VHISP channels that are enabled and connected to an external device [for example, an SSD solid-state storage device (SSD)]. This command returns a bit mask in which bit 0 represents VHISP 20; bit 1 represents VHISP 21; etc.

MMERead( )

`MMERead([`*tag*`,]`*buffer*`,`*address*`,`*length*`);`

This command reads a block of data from mainframe memory. The block begins at the word address specified by the *address* parameter. The *length* parameter specifies the number of 64-bit words to read. If you specify a *tag*, addresses are relative to the beginning of the control point referenced by *tag*; otherwise, the addresses are absolute.

The `MMERead( )` command puts the data into a buffer that you specify with the *buffer* variable. You must define the *buffer* variable as an array of byte, short, unsigned integer (uint), or long data. You must ensure that the buffer is large enough to hold the requested data.

For more information about the *tag* parameter, refer to “tag Parameter Description” on page 7.

MMEReload( )

```
MMEReload(tag|ALL);
```

This command reloads the control point referenced by *tag* or reloads all control points if you specify `ALL`.

For more information about the *tag* parameter, refer to “tag Parameter Description” on page 7.

MMEReset( )

```
MMEReset([1|2]);
```

This command resets MME in the current environment or resets MME to the specified environment.

NOTE: The CRAY T90 series CBP runtime module does not support environment 0.

MMETimeout( )

```
MMETimeout(tag,label);
```

This command returns a nonzero (true) value if the control point referenced by *tag* (or any control point) timed out; this command returns a zero (false) value if the control point(s) did not time out.

For more information about the *tag* parameter, refer to “tag Parameter Description” on page 7.

MMEUnload( )

```
MMEUnload(tag|ALL);
```

This command unloads the control point referenced by *tag* or all control points if you specify `ALL`.

For more information about the *tag* parameter, refer to “tag Parameter Description” on page 7.

MMEWait( )

```
MMEWait(limit);
```

This command waits for *limit* seconds and returns a logical false (zero) value if any control points reach their time limits or find an error. This command returns a logical true (nonzero) value if the control points reach their pass limits or if the user of your CBP program clicks on Continue in the CBP base window to stop the `MMEWait( )` command.

MMEWrite( )

```
MMEWrite([tag,]buffer,address,length);
```

This command writes the block of data that is stored in *buffer* to mainframe memory using the CPU specified by *cpu*. The *buffer* parameter must be an array of byte, short, unsigned integer (uint), or long data and must contain at least as much data as you specify in the *length* parameter.

The block begins at the word address specified by *address*. The *length* parameter specifies the length of the block (read from the buffer) in 64-bit words.

If you specify a *tag*, addresses are relative to the beginning of the control point referenced by *tag*; otherwise, addresses are absolute.

For more information about the *tag* parameter, refer to “tag Parameter Description” on page 7.

SCE-specific Command Descriptions

Table 4 lists the SCE-specific commands that you can use in command buffer programs to manipulate the SCE interface and to automate CRAY T90 series mainframe configuration. Descriptions of the options for each command follow the table.

Table 4. SCE-specific Commands

Command	Description
`SCEGet( )`	Gets configuration data from SCE
`SCEReset( )`	Resets SCE and applies the configuration

SCEGet( )

```
SCEGet(CRAY_CPUS_MASK);
```

This command returns a bit mask of all CP modules in the mainframe that use Cray Research, Inc. (CRI) floating-point number format. In the bit mask, bit 0 represents CPU 00; bit 1 represents CPU 01; etc.

NOTE: The bit mask does not indicate that a CPU is usable or that it is located in the logical partition in which MME is running.

```
SCEGet(IEEE_CPUS_MASK);
```

This command returns a bit mask of all CP modules in the mainframe the use IEEE floating-point number format. In the bit mask, bit 0 represents CPU 00; bit 1 represents CPU 01; etc.

NOTE: The bit mask does not indicate that a CPU is usable or that it is located in the logical partition in which MME is running.

```
SCEGet(SIMULATOR);
```

This command returns a logical nonzero (true) value if SCE is running with the mainframe instruction simulator (MSIM) and a logical zero (false) value if it is running with actual hardware.

```
SCEGet(SYSTEM_TYPE);
```

This command returns the type of system currently configured by SCE, which is one of the following values:

- `SYS_TESTER` for a CP tester
- `SYS_T94` for a CRAY T94 mainframe
- `SYS_T916` for a CRAY T916 mainframe
- `SYS_T932` for a CRAY T932 mainframe

SCEReset( )

```
SCEReset( );
```

This command resets SCE and reapplies the current configuration. This command also resets MME and LME.

LME-specific Command Descriptions

Table 5 lists the LME-specific commands that you can use in command buffer programs to manipulate the LME interface and to automate CRAY T90 series mainframe testing. Descriptions of the options for each command begin on page 26.

Table 5. LME-specific Commands

Command	Description
`LMEActive( )`	Indicates whether or not a logic monitor is active
`LMEAssign( )`	Assigns a parameter set to a module
`LMECreate( )`	Creates a new parameter set (parameters are set to default values)
`LMEDelete( )`	Deletes a parameter set
`LMEGetTP( )`	Returns one test point from the last test-point dump
`LMEGo( )`	Starts recording logic monitor data
`LMEHalt( )`	Stops recording logic monitor data
`LMEIBDump( )`	Dumps the instruction buffers for the specified CPU
`LMELoad( )`	Loads a parameter set from a file
`LMEModuleData( )`	Copies data from an LME buffer or instruction buffer to a buffer that you have defined
`LMERead( )`	Uses the specified CPU to read data from mainframe memory
`LMEReset( )`	Resets LME
`LMESetParams( )`	Modifies parameters
`LMETPDump( )`	Dumps all test points for a module
`LMEWait( )`	Delays command buffer execution for a specific period of time
`LMEWrite( )`	Uses the specified CPU to write data to memory

Available Module Parameter Values

CAUTION

The following list includes all module names that you can use in command buffer programs.

Because your system configuration may not include all of the available modules, ensure that you use only the modules that are available in your current system configuration. If you specify a module that is not included in the current system configuration, the CBP command will return invalid data or cause an error.

Many of the LME-specific commands include a *module* parameter. The following list describes the available values for the *module* parameter:

- The available CP module parameter values include `CPU00` through `CPU37`.
- The available IO module parameters include `IO00` through `IO03`.
- The available shared module parameters include `SHR00` and `SHR01`.
- The available network module parameter values are as follows:

NW_K1_EVEN	NW_C1_EVEN	NW_G1_EVEN	NW_01_EVEN
NW_K1_ODD	NW_C1_ODD	NW_G1_ODD	NW_O1_ODD
NW_K2_EVEN	NW_C2_EVEN	NW_G2_EVEN	NW_02_EVEN
NW_K2_ODD	NW_C2_ODD	NW_G2_ODD	NW_02_ODD
NW_K3_EVEN	NW_C3_EVEN	NW_G3_EVEN	NW_O3_EVEN
NW_K3_ODD	NW_C3_ODD	NW_G3_ODD	NW_O3_ODD
NW_K4_EVEN	NW_C4_EVEN	NW_G4_EVEN	NW_04_EVEN
NW_K4_ODD	NW_C4_ODD	NW_G4_ODD	NW_04_ODD

You can use mathematical operations on these values. For example, `CPU00 + 2` is equivalent to `CPU02`. You can also use these values in a loop to step through a group of values.

Refer to the `/cri/cme/t32/rel/cmd/T90_mods.h` file for specific module parameter values.

pset Parameter Description

The *pset* parameter stores an integer that an `LMELoad( )` or an `LMECreate( )` command returns. This integer, also called a parameter set descriptor, is used by other commands to reference a parameter set that was loaded in a previous `LMELoad( )` command. This value is used in CBP commands to specify which control point the commands reference.

Replace the *pset* variable with the name of an integer variable. The following commands show examples of how to use the *pset* parameter:

```
/* declare the parameter set descriptors */
int example1;
int example2;

/* example of how to create a new parameter set with default
values */
example1 = LMECreate();

/* example of how to load an existing parameter set */
example2 = LMELoad("USR/example.pset");
```

LMEActive( )

`LMEActive(`*module*`|ALL);`

This command returns a nonzero (true) value if the logic monitor for the module referenced by *module* is active. This command returns a zero (false) value if the logic monitor is not active.

If you specify `ALL`, this command returns a nonzero (true) value if a logic monitor for any module in the system is active. If no logic monitors are active, this command returns a zero (false) value.

NOTE: This command is not valid for network modules.

LMEAssign( )

`LMEAssign(`*pset*`,`*module*`|ALL|AVAILABLE);`

This command assigns the parameter set specified by *pset* to the module specified by *module*. If you specify `ALL`, this command assigns the parameter set to all of the modules. If you specify `AVAILABLE`, this command assigns the parameter set to all modules that are available (not currently assigned).

For more information about the *pset* parameter, refer to “pset Parameter Description” on page 26.

LMECreate( )

```
LMECreate( );
```

This command creates a new parameter set in which every parameter is set to its default value. This command returns an integer value referred to as a parameter set descriptor. Use this descriptor with other LME-specific commands to reference the parameter set.

For more information about the parameter set descriptor, refer to “pset Parameter Description” on page 26.

LMEDelete( )

```
LMEDelete(pset|ALL);
```

This command deletes the parameter set specified by *pset*; this command deletes all parameter sets if you specify `ALL`.

For more information about the *pset* parameter, refer to “pset Parameter Description” on page 26.

LMEGetTP( )

```
LMEGetTP(module,TPDUMP,chip,test_point);
```

This command returns the value of one test point from the most recent `LMETPDump( )` command for the module specified by *module*. The *chip* parameter indicates the chip on which the test point is located (for example, CI03). The *test_point* parameter specifies the test point (0200_8 through 0377_8).

For more information about the LME test-point dump command, refer to “LMETPDump( )” on page 36.

LMEGo( )

```
LMEGo(pset|ALL);
```

This command starts recording logic monitor data for the modules that are assigned to the parameter set specified by *pset*. If you specify `ALL`, this command starts recording logic monitor data for all of the assigned modules.

For more information about the *pset* parameter, refer to “pset Parameter Description” on page 26.

LMEHalt( )

```
LMEHalt(pset|ALL);
```

This command stops recording logic monitor data for the modules that are assigned to the parameter set specified by *pset*. If you specify `ALL`, this command stops recording logic monitor data for all of the assigned modules.

For more information about the *pset* parameter, refer to “pset Parameter Description” on page 26.

LMEIBDump( )

```
LMEIBDump(cpu);
```

This command dumps the instruction buffers for the CPU specified by *cpu*. You can use the CPU number or the module symbol to specify the CPU.

LMELoad( )

```
LMELoad(“filename”);
```

This command loads the parameter set in the file specified by *filename*. This command returns an integer called a parameter set descriptor. Use this descriptor with other LME-specific commands to reference the parameter set.

For more information about the *pset* parameter, refer to “pset Parameter Description” on page 26.

You can use the following shortcut options to specify the directory that contains the file you want to load.

Shortcut Options	Directory
USR	/cri/cme/t32/usr/lme/pset
REL	/cri/cme/t32/rel/lme/pset
ALPHA	/cri/cme/t32/alpha/lme/pset

To use the shortcut options, follow the convention of `LMELoad("`*shortcut/filename*`");`. For example, `LMELoad("USR/myparams");` will load `/cri/cme/t32/usr/lme/pset/myparams`.

LMEModuleData( )

`LMEModuleData(`*module*`,A|B|C|D|CURRENT,`*buffer*`);`

This command copies data from an LME buffer (A, B, C, D, or the current buffer) to a buffer that you have defined. The *module* variable specifies the module from which to obtain the data. The *buffer* variable indicates the buffer you have defined.

You must define the *buffer* variable as an array of byte, short, unsigned integer (uint), or long data. You must ensure that the buffer is large enough to hold 0100_8 64-bit words of data.

NOTE: This command is not valid for network modules.

`LMEModuleData(`*cpu*`,IBDUMP,[`*ibnum*`],`*buffer*`);`

This command copies the instruction buffer dump data into a buffer that you have defined. This command copies data from a previous `LMEIBDump( )` command. You may copy data from one or all instruction buffers.

The *cpu* variable specifies the CPU from which you want to obtain instruction buffer data. The optional *ibnum* variable specifies that you want to obtain data from a specific instruction buffer: *ibnum* is the number of the instruction buffer. If you do not specify *ibnum*, this command copies data from all the instruction buffers of the most recent dump.

The *buffer* variable indicates the buffer you have defined. You must define the *buffer* variable as an array of byte, short, unsigned integer (uint), or long data. You must ensure that the buffer can hold 0400_8 64-bit words of data if you want to copy all of the instruction buffers. You must ensure that the buffer can hold 040_8 64-bit words of data if you want to copy one instruction buffer.

LMERead( )

`LMERead(`*buffer*`,`*address*`,`*length*`,`*cpu*`);`

This command reads a block of data from mainframe memory using the CPU specified by *cpu*. The block begins at the word address specified by *address*. The length of the block is specified by *length*. This command copies the data to the *buffer* variable, which must be an array of byte, short, unsigned integer (uint), or long data. You must ensure that the buffer is large enough to hold the requested data.

LMEReset( )

`LMEReset( );`

This command resets LME.

LMESetParams( )

`LMESetParams(`*pset*`,BREAKPOINT_IP,`*ip_value*`);`

This command sets a breakpoint on the instruction parcel value specified by *ip_value* for the parameter set specified by *pset*.

For more information about the *pset* parameter, refer to “pset Parameter Description” on page 26.

`LMESetParams(`*pset*`,BREAKPOINT_PREG,`*p-reg_value*`);`

This command sets a breakpoint on the P register value specified by *p-reg_value* for the parameter set specified by *pset*.

For more information about the *pset* parameter, refer to “pset Parameter Description” on page 26.

`LMESetParams(`*pset*`,BREAKPOINT_TRIGGER);`

This command sets a breakpoint on the current trigger condition for the parameter set specified by *pset*.

For more information about the *pset* parameter, refer to “pset Parameter Description” on page 26.

`LMESetParams(`*pset*`,IDENT,"`*ident_string*`"[,"`*description_string*`"]);`

This command sets the parameter set label for the parameter set specified by *pset*. The *ident_string* parameter specifies the parameter set label and must be six or fewer characters.

You can also use this command to create a description of the parameter set by using the *description_string* parameter.

For more information about the *pset* parameter, refer to "pset Parameter Description" on page 26.

`LMESetParams(`*pset*`,MODTYPE,CPU|IEEE|SHR|IO|NW);`

This command sets the module type for the parameter set specified by *pset*.

For more information about the *pset* parameter, refer to "pset Parameter Description" on page 26.

`LMESetParams(`*pset*`,NW_CHIP_MCONN,`*chip*`,`*pin*`[,`*test_point*`]);`

This command selects an NW module chip to monitor through the specified network module maintenance connector pin, which is routed to an external scope point connected to the bulkhead. The *chip* parameter specifies the name of the network module chip (for example, LA01) on which the test point that you want to select is located. The *pin* parameter (0 through 3) specifies which maintenance connector pin should receive the test-point data of the specified chip. The *test_point* parameter specifies the test-point value to use for the selected chip.

If you omit the *test_point* parameter, this command uses the currently selected test point for the chip.

For more information about the *pset* parameter, refer to "pset Parameter Description" on page 26.

NOTE: The maintenance connector pins are numbered 0 to 3 on the half of the network module that is connected to an even section of memory and 4 to 7 on the half of the network module that is connected to an odd section of memory. This command sets the proper pin number based on which half of a network module is being referenced.

`LMESetParams(`*pset*`,NW_CHIP_PORT,`*chip*`,`*port*`[,`*test_point*`]);`

This command selects a network module chip to monitor through the network module port. (Data sent to the NW module ports can be monitored by connected CPUs using test point 0215 on the CJ chips.) The *chip* parameter specifies the name of the network module chip (for example, LA01) on which the test point that you want to select is located. The port parameter (0 through 7) specifies which network port should receive the test-point data of the specified chip. The *test_point* parameter specifies the test-point value to use for the selected chip.

If you omit the *test_point* parameter, this command uses the currently selected test point for the chip.

For more information about the *pset* parameter, refer to “pset Parameter Description” on page 26.

`LMESetParams(`*pset*`,RECORD_CHIP,`*chip*`[,`*test_point*`]);`

This command adds a chip to the list of chips whose test points will be recorded by the parameter set specified by *pset*. The test-point number for the chip specified by *chip* can optionally be set.

If you omit the *test_point* parameter, this command uses the currently selected test point for the chip.

For more information about the *pset* parameter, refer to “pset Parameter Description” on page 26.

`LMESetParams(`*pset*`,RECORD_CHIP_CLEAR);`

This command clears all test points that are currently set for recording in the parameter set specified by *pset*.

For more information about the *pset* parameter, refer to “pset Parameter Description” on page 26.

`LMESetParams(`*pset*`,RECORD_CHIP_SECONDARY,`*chip*`[,`*test_point*`]);`

This command selects the secondary output bit for the test point specified by *test_point*. The *chip* parameter specifies the chip on which the test point is located. The *pset* parameter specifies the parameter set.

If you omit the *test_point* parameter, this command uses the currently selected test point for the chip.

For more information about the *pset* parameter, refer to “pset Parameter Description” on page 26.

`LMESetParams(`*pset*`,RECORD_IP_BITS,`*bit_mask*`);`

This command specifies that you want to record instruction parcel data. The *bit_mask* parameter specifies which bits of the current instruction parcel should be recorded. This command forces the `RECORD_MODE` parameter to `RECORD_256BY512`.

For more information about the *pset* parameter, refer to “pset Parameter Description” on page 26.

`LMESetParams(`*pset*`,RECORD_MODE,RECORD_256BY32|`
`RECORD_512BY16|RECORD_1024BY8);`

This command sets the recording mode for the parameter set specified by *pset*.

For more information about the *pset* parameter, refer to “pset Parameter Description” on page 26.

`LMESetParams(`*pset*`,RECORD_ONEWORD,TRUE|FALSE);`

This command sets/clears the one-word-per-trigger recording mode flag for the parameter set specified by *pset*.

For more information about the *pset* parameter, refer to “pset Parameter Description” on page 26.

`LMESetParams(`*pset*`,RECORD_PREG_BITS,`*bit_mask*`);`

This command specifies that you want to record P register data. The *bit_mask* parameter specifies which bits of the P register should be recorded. This command forces the `RECORD_MODE` parameter to `RECORD_256BY512`.

For more information about the *pset* parameter, refer to “pset Parameter Description” on page 26.

`LMESetParams(`*pset*`,SCOPE_CHIP,`*chip*`[,`*test_point*`]);`

This command directs test-point data to the maintenance connector and then to external scope points connected to the bulkhead. The *pset* parameter specifies the parameter set. The *chip* parameter indicates the name of the chip on which the test point is located (for example, CI03). The *test_point* parameter specifies the test point (0200_8 through 0377_8).

If you omit the *test_point* parameter, this command uses the currently selected test point for the chip.

For more information about the *pset* parameter, refer to “pset Parameter Description” on page 26.

`LMESetParams(`*pset*`,SCOPE_CHIP_CLEAR);`

This command clears all set scope points for the parameter set specified by *pset*.

For more information about the *pset* parameter, refer to “pset Parameter Description” on page 26.

`LMESetParams(`*pset*`,TESTPOINT,`*chip*`,`*test_point*`);`

This command sets the test-point value specified by *test_point* for a single chip. The *chip* parameter indicates the chip on which the test point is located. The *pset* parameter specifies the parameter set in which you want to set the test point.

For more information about the *pset* parameter, refer to “pset Parameter Description” on page 26.

`LMESetParams(`*pset*`,TRIGGER_CHIP,`*chip*`,`*state*`[,`*test_point*`]);`

This command sets a trigger condition for the parameter set specified by *pset*. The *chip* parameter indicates the chip on which the test point is located (for example, CI03). The *state* parameter indicates whether the signal is a 1 or a 0. The *test_point* parameter specifies the test point (0200_8 through 0377_8).

If you omit the *test_point* parameter, this command uses the currently selected test point for the chip.

For more information about the *pset* parameter, refer to “pset Parameter Description” on page 26.

`LMESetParams(`*pset*`,TRIGGER_CHIP_ARM,`*chip*`,`*state*`,[,`*test_point*`]);`

This command sets the trigger arm condition for the parameter set specified by *pset*. The *chip* parameter indicates the chip on which the test point is located (for example, CI03). The *state* parameter indicates whether the signal is a 1 or a 0. The *test_point* parameter specifies the test point (0200_8 through 0377_8).

If you omit the *test_point* parameter, this command uses the currently selected test point for the chip.

For more information about the *pset* parameter, refer to "pset Parameter Description" on page 26.

`LMESetParams(`*pset*`,TRIGGER_CHIP_CLEAR);`

This command clears all set triggers in the parameter set specified by *pset*.

For more information about the *pset* parameter, refer to "pset Parameter Description" on page 26.

`LMESetParams(`*pset*`,TRIGGER_CHIP_DISARM,`*chip*`,`*state*`[,`*test_point*`]);`

This command sets the trigger disarm condition for the parameter set specified by *pset*. The *chip* parameter indicates the chip on which the test point is located (for example, CI03). The *state* parameter indicates whether the signal is a 1 or a 0. The *test_point* parameter specifies the test point (0200_8 through 0377_8).

If you omit the *test_point* parameter, this command uses the currently selected test point for the chip.

For more information about the *pset* parameter, refer to "pset Parameter Description" on page 26.

`LMESetParams(`*pset*`,TRIGGER_DELAY,`*delay*`);`

This command sets a trigger delay condition for the parameter set specified by *pset*. The *delay* parameter indicates the length of the delay in clock periods.

For more information about the *pset* parameter, refer to "pset Parameter Description" on page 26.

LMETPDump( )

```
LMETPDump(module);
```

This command dumps all test points for the module specified by *module*.

LMEWait( )

```
LMEWait(limit);
```

This command delays command buffer execution for *limit* seconds. After the delay, this command returns a zero (false) value if any logic monitors are still active. This command returns a nonzero (true) value if no logic monitors are active or if the user of your command buffer program clicks on  in the CBP base window.

LMEWrite( )

```
LMEWrite(buffer,address,length,cpu);
```

This command writes a block of data that is stored in *buffer* to mainframe memory using the CPU specified by *cpu*. The *buffer* parameter must be an array of byte, short, unsigned integer (uint), or long data and must contain at least as much data as you specify in the *length* parameter.

The block begins at the word address specified by *address*. The *length* parameter specifies the length of the block in 64-bit words.

Reader Comment Form

Title: CBP Runtime Module (CRAY T90™ Series) **Number: HDM-071-A**

Your feedback on this publication will help us provide better documentation in the future. Please take a moment to answer the few questions below.

For what purpose did you primarily use this document?

_____Troubleshooting _____Tutorial or introduction
_____Reference information _____Classroom use
_____Other - please explain _____

Using a scale from 1 (poor) to 10 (excellent), please rate this document on the following criteria and explain your ratings:

_____Accuracy _____
_____Organization _____
_____Readability _____
_____Physical qualities (binding, printing, page layout) _____
_____Amount of diagrams and photos _____
_____Quality of diagrams and photos _____

Completeness (Check one and explain your answer)

_____Too much information _____Too little information _____Correct amount
_____
_____
_____

You may write additional comments in the space below. Mail your comments to the address below, fax them to us at 715-726-4353, or E-mail them to us at *spt@cray.com.* When possible, please give specific page and paragraph references. We will respond to your comments in writing within 48 hours.

_____
_____
_____
_____
_____

NAME _____
JOB TITLE _____
E-MAIL ADDRESS _____
SITE/LOCATION _____
TELEPHONE _____
DATE _____

[or attach your business card]

CRAY RESEARCH, INC.

Attn: Service Publications and Training
890 Industrial Boulevard
Chippewa Falls, WI 54729