SoC Designer

Version 9.6

Tools API Reference Manual

arm

101140_0906_00_en

SoC Designer

Tools API Reference Manual

Release Information

Document History

Issue	Date	Confidentiality	Change
0905-00	17 November 2017	Non-Confidential	Release with 9.5
0906-00	23 February 2018	Non-Confidential	Release with 9.6

Non-Confidential Proprietary Notice

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

LES-PRE-20349

Confidentiality Status

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by Arm and the party that Arm delivered this document to.

Unrestricted Access is an Arm internal classification.

Product Status

The information in this document is Final, that is for a developed product.

Web Address

http://www.arm.com

Contents
SoC Designer Tools API Reference Manual

Preface

This preface introduces the *SoC Designer Tools API Reference Manual*.

It contains the following:

About this book

This document describes the programming interfaces to SoC Designer tools and utilities.

Using this book

This book is organized into the following chapters:

Chapter 1 Introduction to the Tools API

This chapter describes the SoC Designer Tools API.

Chapter 2 The SoC Designer Simulator API

This chapter describes the SoC Designer Simulator API.

Chapter 3 The MxPlugin API

This chapter explains the steps required to create and install SoC Designer plugins.

Chapter 4 Creating GUI-mode Plugins

This chapter describes how to create GUI-mode plugins.

Chapter 5 MxLCD Interface Specification

This chapter describes all classes and member functions of the MxLCD API.

Chapter 6 Creating Batch-Mode Plugins

This chapter describes how to create batch-mode plugins.

Glossary

The Arm® Glossary is a list of terms used in Arm documentation, together with definitions for those terms. The Arm Glossary does not contain terms that are industry standard unless the Arm meaning differs from the generally accepted meaning.

See the *Arm® Glossary* for more information.

Typographic conventions

italic
: Introduces special terminology, denotes cross-references, and citations.

bold
: Highlights interface elements, such as menu names. Denotes signal names. Also used for terms in descriptive lists, where appropriate.

`monospace`
: Denotes text that you can enter at the keyboard, such as commands, file and program names, and source code.

`monospace`
: Denotes a permitted abbreviation for a command or option. You can enter the underlined text instead of the full command or option name.

`monospace italic`
: Denotes arguments to monospace text where the argument is to be replaced by a specific value.

`monospace bold`
: Denotes language keywords when used outside example code.

`<and>`
: Encloses replaceable terms for assembler syntax where they appear in code or code fragments. For example:

```
MRC p15, 0, <Rd>, <CRn>, <CRm>, <Opcode_2>
```

SMALL CAPITALS
: Used in body text for a few terms that have specific technical meanings, that are defined in the *Arm® Glossary*. For example, IMPLEMENTATION DEFINED, IMPLEMENTATION SPECIFIC, UNKNOWN, and UNPREDICTABLE.

Feedback

Feedback on this product

If you have any comments or suggestions about this product, contact your supplier and give:

- The product name.
- The product revision or version.
- An explanation with as much information as you can provide. Include symptoms and diagnostic procedures if appropriate.

Feedback on content

If you have comments on content then send an e-mail to *errata@arm.com*. Give:

- The title *SoC Designer Tools API Reference Manual*.
- The number 101140_0906_00_en.
- If applicable, the page number(s) to which your comments refer.
- A concise explanation of your comments.

Arm also welcomes general suggestions for additions and improvements.

Note

Arm tests the PDF only in Adobe Acrobat and Acrobat Reader, and cannot guarantee the quality of the represented document when used with any other PDF reader.

Other information

- *Arm® Developer*.
- *Arm® Information Center*.
- *Arm® Technical Support Knowledge Articles*.
- *Technical Support*.
- *Arm® Glossary*.

Chapter 1
Introduction to the Tools API

This chapter describes the SoC Designer Tools API.

It contains the following section:

1.1 Introduction

This SoC Designer Tools interface includes the following:

- The SoC Designer Simulation API (MxSimAPI).
- The SoC Designer Plugins interface.
- The SoC Designer LCD/mouse/keyboard OS-integration interface.

Note

New users must familiarize themselves with the SoC Designer tools. The *SoC Designer User Guide* (100996) and the *SoC Designer Tutorial* are recommended.

This guide contains advanced interfaces that are intended for users already familiar with SoC Designer.

SoC Designer is a SystemC simulation environment for easy modeling and fast simulation of integrated systems-on-chip with multiple cores, peripherals, and memories.

The cycle-optimized scheduler and the transaction-based component interfaces enable high simulation speed while retaining sufficient accuracy.

SoC Designer systems can be used as standalone simulation models or integrated into HW simulation or HW/SW co-simulation tools.

Third-party debuggers attach easily to one or more targets within SoC Designer for source level debugging and SW development.

Chapter 2
The SoC Designer Simulator API

This chapter describes the SoC Designer Simulator API.

Note

The functions in this chapter apply to the SoC Designer Simulator, and SoC Designer Runtime Simulator. Enums, specific types, and API names that are used in this book are specific to the namespace specified by `eslapi::`, unless stated otherwise.

It contains the following sections:

2.1 Overview

The following simulator tools are available:

SoC Designer Simulator
A standalone executable that can be run in GUI or batch mode. This simulator is distributed as part of the full SoC Designer package and can run `.mxp`, `.mxs`, or `.mxr` simulation files.

SoC Designer RTOE Simulator
A standalone executable that can be run in GUI or batch mode. This simulator is provided for running Runtime simulation files (which can be identified by their `.mxe` extension) that have been produced by the Runtime Generator feature of the SoC Designer tool.

All interfacing is done through the `MxSimAPI` class. A pointer to this class is returned from a call to `getMxSimAPI()`:

```
MxSimAPI* simulator = getMxSimAPI();
```

Use the pointer to call an API function in the simulation API:

```
simulator->simStep(10); // execute 10 cycles
```

2.2 API functions for information access

The functions in this section are used to obtain information about the simulation state.

getCycleCount

Returns the current cycle count.

```
uint64_t getCycleCount()
```

getTimeElapsed

Returns the elapsed simulation time, in clock cycles.

```
double getTimeElapsed (MxTimeUnit &unit )
```

getClockPeriod

Returns the clock period, that is, the duration of one clock cycle.

```
double getClockPeriod (MxTimeUnit &unit )
```

getSchedulerState

Returns the simulator state. The following states are possible: MX_RUNNING or MX_HALTED

```
MxSimState getSchedulerState()
```

getTopComponent

Returns a pointer to the top-level component.

```
CASIModuleIF* getTopComponent()
```

2.3 API functions for simulation control

The functions described in this section are used to control the simulation. It is not typically required to call these functions from a component, but they can be used if connecting external debuggers or other tools.

Note

These functions do not immediately execute the requested action. Use the callback interface to wait until the requested action is completed. See *2.4 API functions for managing event callbacks* on page 2-17.

This section contains the following subsections:

2.3.1 simStep

Request advancing one or more simulation steps.

```
void simStep(int num_cycles)
```

The simulation stops after the requested number of cycles or due to an external break event. An external break event can be any of the following:

- A breakpoint in SoC Designer Simulator.
- An event in an external debugger.
- If you press the **Stop** button in SoC Designer Simulator or a debugger.

The number of cycles can be passed as a parameter. If no value is passed, the default value of 1 is used.

```
simulator->simStep(10); // executes 10 cycles (or until breakpoint)
```

2.3.2 simTimeStep

Request running the simulation for the specified amount of time. The time value is converted into discrete clock cycles by dividing the specified time by the clock period of the system.

```
void simTimeStep (double time, MxTimeUnit unit = (MxTimeUnit)-1 )
```

Use the following form of the function to run the simulation with the time specified as a string.

```
bool simTimeStep (const string &time )
```

To set the time step to 10ns for example, use:

```
simulator->simStep("10 n");
```

The timeunit in the *time* string can be one of:

`f` femtoseconds

`n` nanoseconds

`u` microseconds

`m` milliseconds

`s` seconds

2.3.3 simRun

Request running the simulation until an external break event occurs.

```
void simRun()
```

An external break event can be:

- A breakpoint in SoC Designer Simulator.
- An event in an external debugger.
- If you press the **Stop** button in SoC Designer Simulator or a debugger.

2.3.4 simHalt

Request stopping the simulation.

```
void simHalt()
```

Note

It is not possible to stop the simulation in the middle of a simulation cycle. The current simulation cycle is completed before the simulator stops.

2.3.5 simReset

Request a simulation reset. The current simulation cycle is completed before the simulation is reset.

```
void simReset(level)
```

The reset `level` must be passed as a parameter and can be one of the following:

- `CASI_RESET_SOFT`
- `CASI_RESET_HARD`

The hard reset causes the components to reload the last application files that were loaded through SoC Designer Simulator. A soft reset returns the simulation counter to zero but does not reload application files.

2.3.6 simRefresh

Forces a refresh of all CADI debug windows and all attached external debuggers.

```
void simRefresh()
```

Note

A refresh is always done for every run, step, or halt call. Only use this function to refresh an external debugger without advancing the simulation time. This might occur, for example, if a value has been changed by the user in an external debugger and it is necessary to reflect the change in the internal windows and other external debuggers.

2.3.7 externalRefresh

Cause all external debuggers to refresh and also refreshes SoC Designer.

```
void externalRefresh()
```

2.3.8 setClockPeriod

Set the clock period of the currently loaded simulation.

```
void setClockPeriod (double time, MxTimeUnit unit )
```

The timeunit can be one of:

`f` femtoseconds
`n` nanoseconds
`u` microseconds
`m` milliseconds
`s` seconds.

To set the clock period to 10 nanoseconds, for example, use:

```
simulator->setClockPeriod(10, “n")
```

2.3.9 enableDebugMessages

Control debug messages from the simulation.

```
bool enableDebugMessages( bool enable )
```

This call should be made after the simulation has been initialized. Zero is returned on an attempt to invoke this function before the simulation is initialized; non-zero return value on a success.

Related concepts

2.4 API functions for managing event callbacks on page 2-17.

2.4 API functions for managing event callbacks

SoC Designer enables you to register callbacks for the following events:

- Start or stop of simulation (`SimStatusListener`).
- Reset of simulation (`ResetListener`).
- Begin of simulation cycle (`CycleStartListener`).
- End of communicate phase (`EndOfCommunicateListener`).
- End of simulation cycle (`CycleListener`).
- Simulator is about to quit (`SimQuitListener`).

The callback interfaces are implemented using C++ classes that handle the callbacks. Each callback event type provides its own pure virtual interface. The callback handlers can be registered by using one of the following functions:

- `simulator->registerStatusListener(MyStatusListener);`
- `simulator->registerResetListener(MyResetListener);`
- `simulator->registerCycleStartListener(MyCycleStartListener);`
- `simulator->registeEndOfCommunicateListener(MyEndOfCommunicateListener);`
- `simulator->registerCycleListener(MyCycleListener);`
- `simulator->registerSimQuitListener(MySimQuitListener);`

To register a callback:

1. Create an object that inherits from the appropriate listener class.
2. Implement the callback function associated with the listener class.
3. Register the callback by calling the appropriate `register`*`Type`*`Listener()` method of the MxSimAPI object with your listener object as the parameter.

To unregister a callback handler added by `register`*`Type`*`Listener()`, call the corresponding `unregister`*`Type`*`Listener()` function.

2.4.1 The MxSimStatusListener callback interface

The class declaration of `MxSimStatusListener` interface is:

```
class MxSimStatusListener
{
public:
    virtual void simulatorStatusChanged (int running ) = 0;
};
```

When inheriting from this class, the method `simulatorStatusChanged()` must be implemented. This method is called whenever the simulation is started or stopped. The status of the simulation is passed as a parameter. If `running` is 0, the simulation is halted, otherwise it is running.

2.4.2 The MxResetListener callback interface

The class declaration of the `MxResetListener` interface is:

```
class MxResetListener
{
public:
    virtual void simulationResetCB() = 0;
};
```

If a class inherits from this interface, the `simulationResetCB()` method must be implemented. This function is called every time a reset occurs. A reset can be caused by the following:

- If you press the **Reset** button in SoC Designer or an attached debugger.
- A component requests a reset through the SimAPI.

2.4.3 The MxCycleStartListener callback interface

The class declaration of the `MxCycleStartListener` interface is:

```
class MxCycleStartListener
{
public:
    virtual void cycleStartCB() = 0;
};
```

The function `cycleStartCB()` must be implemented. It is called every time a cycle is about to be executed.

2.4.4 The MxEndOfCommunicateListener callback interface

This callback is invoked after the invocation of the communicate() function for each model in the simulation for that cycle. In a multi-clock domain system, the EndOfCommunicateListener callback function is invoked once per the main simulation clock.

All models in derived clock domains that execute during that clock cycle complete execution of their communicate() functions before this callback is invoked. Also, the EndOfCommunicateListener callback is invoked before any update functions are called for the clock cycle. The following is the class declaration of the `MxEndOfCommunicateListener` class:

```
Class MxEndOfCommunicateListener{public:  virtual void endOfCommunicateCB() = 0;};
```

The function `endOfCommunicateCB()` must be implemented. It is called every time a cycle is about to be executed.

2.4.5 The MxCycleListener callback interface

The following is the class declaration of the `MxCycleListener` class:

```
class MxCycleListener
{
public:
    virtual void cycleFinishedCB() = 0;
};
```

The function `cycleFinishedCB()` must be implemented. It is called every time a cycle has been executed.

2.4.6 The MxSimQuitListener callback interface

The following is the declaration of the `MxSimQuitListener` class:

```
class MxSimQuitListener
{
public:
    virtual void simulatorWillQuit() = 0;
};
```

The function `simulatorWillQuit()` is called before the simulator quits. This can be used for cleanup purposes if it is necessary to know that SoC Designer Simulator is about to shut down.

Chapter 3
The MxPlugin API

This chapter explains the steps required to create and install SoC Designer plugins.

It contains the following sections:

3.1 Overview

Use the SoC Designer plugin feature to create the following kinds of features:

- GUI-mode plugins to add new main menu and component, or connection-context menu items.

  ——— **Note** ———

  For components that are dragged and dropped into the SoC Designer Canvas, the component and connection menus only appears after the system is saved and reopened.

  ———————————

- Batch-mode plugins to add new batch-mode script commands.

For details on how plugins are loaded, see *3.2 Loading a plugin* on page 3-21.

For details on how plugins are initialized, see *3.3 The MxPluginInit function* on page 3-23.

For details on creating plugins, see:

- *3.3 The MxPluginInit function* on page 3-23
- *3.4 The MxPluginTerminate function* on page 3-24
- *3.5 The MxPlugin class* on page 3-25
- *3.6 The MxPluginCallBack* on page 3-34
- *3.7 The MxPluginComponent callback classes* on page 3-35
- *3.8 The MxPluginDialog class* on page 3-36
- *3.9 The MxPluginDialogParameter classes* on page 3-37

The plugin initialization code specifies the menu items that must be placed in the tool. If you click a plugin menu item, the call is transferred to the callbacks functions in the plugin code. This provides a way to customize part of the tool behavior.

The following interfaces are available to SoC Designer plugin developers:

`MxPlugin` Plugin creation, insertion of menus, script commands, and basic custom dialog creation.

`CASI` SoC Designer simulation interface.

`CADI` SoC Designer debug interface.

`CAPI` SoC Designer profiling interface.

——— **Note** ———

A plugin example package is included in the `examples` folder of your SoC Designer installation.

———————————

3.2 Loading a plugin

Loading a plugin enables the SoC Designer tool to find and use your plugin libraries. You can load a plugin by pointing the SoC Designer tool to the plugin libraries in any of the following ways:

Default file location

Place the plugin libraries in a directory named `Release` (for release mode) or `Debug` (for debug mode), that are subdirectories of the default plugin directory for the SoC Designer installation. The default plugin directory for SoC Designer is the `etc/plugins` directory, located in the SoC Designer installation root directory. `MAXSIM_HOME` is the environment variable that specifies the SoC Designer installation root directory, so the default parent directory for plugin `Release` and `Debug` directories is `${MAXSIM_HOME}/etc/plugins`.

Plugins are loaded by this method when the SoC Designer tool initializes.

Caution

For SoC Designer Canvas, if all of the plugins are loaded using `MAXSIM_HOME/etc/plugins`, the CASI, CAPI, and CADI interfaces of the opened system is not available to those plugins.

If access to these interfaces is required, load the plugins using the `MAXSIM_PLUGINS` variable.

Plugin variable location

Place the plugin libraries in a directory named `Release` (for release mode) or `Debug` (for debug mode), under a parent directory you specify by the path you set in the environment variable `MAXSIM_PLUGINS`. Plugins that load by being located in a subdirectory of `${MAXSIM_PLUGINS}` load when the SoC Designer system initializes.

Note

The option **Include plugins from $MAXSIM_PLUGINS** environment variable must be checked in the **SoC Designer Preferences** > **Component Library** menu for these plugins to be recognized.

Locate by configuration file entry

Edit a configuration file to locate your plugin libraries. For information on editing configuration files, see the section on file configuration in the *SoC Designer User Guide* (100996).

The following code snippet loads release and debug libraries for a plugin named `gui_plugin_example`.

```
plugin "gui_plugin_example"
{
    path       = "./Release/gui_plugin_example.dll";
    debug_path = "./Debug/gui_plugin_example.dll";
    init_last = false; // This is the default
}
```

`init_last` is set to indicate whether this plugin must be loaded after all other plugins in the configuration file. Set `init_last` to `true` to load the plugin last.

Plugins loaded in configuration files load dynamically. In designer mode, this means that they are loaded when the component list is created or refreshed.

For SoC Designer Simulator, plugins are:

- Dynamically loaded when a system file is opened.
- Unloaded when the system closes.

Note

Plugins loaded dynamically cannot add **Toolbar** menu items.

Loading a plugin in batch mode

The following considerations apply if a plugin is loaded in batch mode:

- Plugins can be loaded in scripted or non-scripted batch mode.
- Scripts cannot access GUI functions, therefore only the non-GUI functions of plugins are available to the SoC Designer system in batch mode.

For more information on batch mode processing with the SoC Designer tools, see the *SoC Designer User Guide* (100996).

For any of the chosen methods used to load plugins into the SoC Designer tool, loading completes with the following actions:

1. Open the specified libraries.
2. Execute the `MxPluginInit()` function in each library.

For more detail on how plugins are initialized, see *3.3 The MxPluginInit function* on page 3-23.

3.3 The MxPluginInit function

Represents the entry point into the plugin dynamic library (DLL or SO).

The plugin developer must place all the plugin initialization and setup code into the `MxPluginInit()` function. The MxPluginInit contains the code for inserting all the callbacks for menu items and batch commands.

The following is an example of an empty template `MxPluginInit()` function:

```
extern "C" void MxPluginInit(void){
    //Place your plugin setup and initialization code here
    …
}
```

The `MxPluginInit()` function must be exported from the plugin dynamic library (DLL). For Win32, the function must be added to the `EXPORTS` section of the Microsoft Visual C++ DEF file.

3.4 The MxPluginTerminate function

Represents the exit point from the plugin dynamic library.

You must deallocate all the memory that was created in the `MxPluginInit()` initialization function and perform any other required cleanup operations.

The function must be exported:

```
extern "C" void
MxPluginTerminate(void)
{
    //Place your exit code here, for example
    // delete(hierarchyCallback);
    // delete(pluginP);
}
```

3.5 The MxPlugin class

The first step to implementing the `MxPluginInit()` function is the creation of an object of type `MxPlugin`. This provides the interface for all of the SoC Designer plugin functionality.

The following example shows the `MxPluginInit()` class definition.

```
class MxPlugin {
public:
    MxPlugin(const char *name);
    ~MxPlugin();
    //inserts a menu item into the main menu under “Plugins”
    bool insertItemMenu(const char *text, MxPluginCallBack *myCallback,
        const char *pixmap[] = NULL, const char *toolTips = "",
        const char *whatsThis = "");

    bool insertSeparator();

    //Inserts a context menu item for the component context menu
    bool insertContextMenuItem(const char *text,
        MxPluginComponentCallBack *myComponentCallback,
        const char *toolTips = "", const char *whatsThis = "");

    //Inserts a script batch-mode plugin command
    bool insertBatchModeCommand(const char *batchCommand,
        MxPluginCallBack *myCallback);

    bool insertCustomCallback(MxPluginCustomComponentCallBack
        *myComponentCallback, MxCustomComponentCallbackType context,
        const char *typeComponent);

     bool insertDoubleClickCallBack(MxPluginComponentDoubleClick
                    *dClickCallBack);

    bool insertComponentUpdateCallBack(MxPluginComponentUpdate *updateCallBack);

    bool insertListenerCallBack(MxPluginListener *listenerCallBack);

    //Creates a custom dialog
    MxPluginDialog *createCustomDialog(const char *title);

    //Removes a custom dialog
    void removeDialog(MxPluginDialog *);

    //Save the current system as an MXP file
    bool saveMXPFile(const char *filename);

    //Display a message on the output window
    void message(const char *text);

    //Execute a batch-mode script command
    bool executeBatchmodeCommand (const char *command, int numParams, … );

    //Get the path to the component’s library file
    const char *getDllPathComponent(CASIModule *component);

    //MxDesigner only: Returns the number of connections in the current syst.
    int getNoDesignerConnections(void);

    //MxDesigner only: Returns the “index” connection for the system for
    //which the getNoDesignerConnection() function was called last
    MxPluginDesignerConnection *getDesignerConnection(int index);

    bool loadMXPFile(const char *mxpfilepath);

    bool addLibComponent(const char *conffile, const char *releasePath,
         const char *debugPath = NULL, bool bRelease = true,
         bool bRelative = true);

    bool addLibMXP(const char *conffile, const char *mxpfilepath,
                   bool bRelative = true);

    string  getInstanceName(){return instanceName;}
    bool checkGUIMode();

    bool checkDesignerMode();

    bool exportSPIRIT(char *module_name, char *vendor, char *library,
                      char *version, char *file_path, char *lgi_path);

    void GetMxVersionInfo(int* major, int* minor, int* revision);
```

```
    void sendPluginMessage(const char *ReceiverPluginName,
        unsigned long cmdWord, unsigned long dataBuffer,
        unsigned long dataBufferSize, string& retStatus);

    bool insertCustomProbe(eslapi::CASIPortIF *portA, eslapi::CASIPortIF *portB,
        eslapi::CASIModuleIF* puProbe, eslapi::CASIPortIF *puProbeMasterPort,
        eslapi::CASIPortIF *puProbeSlavePort, string& strErrorMessage,
        bool bValueChange = false);

    bool removeCustomProbe(eslapi::CASIModuleIF* puProbe,
        string& strErrorMessage);

    bool insertCustomIcon(eslapi::CASIModuleIF* puComponentA,
        eslapi::CASIPortIF *portA,
        eslapi::CASIModuleIF* puComponentB, eslapi::CASIPortIF *portB,
        const char **pixmap,
        string& strErrorMessage);

    bool removeCustomIcon(eslapi::CASIModuleIF* puComponentA,
        eslapi::CASIPortIF *portA,
        eslapi::CASIModuleIF* puComponentB, eslapi::CASIPortIF *portB,
        const char **pixmap,
        string& strErrorMessage);"

private:
    string instanceName;     // Name of the instance given at creation time
};
```

You can access the plugin object and its functions from a pointer in your implementation. For example,

```
pluginP->GetMxVersionInfo(&major, &minor, &revision);
```

where `pluginP` points to a new plugin object that is created in `MxPluginInit()`. See *Chapter 4 Creating GUI-mode Plugins* on page 4-41 for examples of accessing plugin functions.

3.5.1 Member functions

The following sections describe the member functions for the `MxPlugin` class:

Note

To see instructive, detailed examples of the usage of the `MxPlugin` class, see the `GUIPluginExample` and `BatchModePluginExample` sample plugins under the `example` directory of your SoC Designer installation.

3.5.2 addLibComponent

This function takes the specified DLLs and installs them in the specified `maxlib.conf` component library file. This functionality is similar to the **add component to Model Libraries** item on the SoC Designer Canvas **Tools** menu.

```
bool addLibComponent(const char *conffile, const char *releasePath,
const char *debugPath = NULL, bool bRelease = true,
bool bRelative = true);
```

where:

`conffile`	Specifies the `maxlib.conf` component library file.
`releasePath, debugPath`	Specifies the paths to the release and debug versions of the DLLs.
`bRelease`	Specifies whether the library is a release or debug library.
`bRelative`	Specifies whether the entry in the conf file is a relative or absolute path.

3.5.3 addLibMXP

This function takes the specified system and installs it in the specified `maxlib.conf` component library file that is referenced from `conffile`. This functionality is similar to the **add system to Model Libraries** item on the SoC Designer **Tools** menu.

```
bool addLibMXP(const char *conffile, const char *mxpfilepath,
               bool bRelative = true);
```

where:

`conffile`	Specifies the `maxlib.conf` component library file.
`mxpfilepath`	Specifies the paths to the mxp file.
`bRelative`	Specifies whether the entry in the conf file is a relative or absolute path.

3.5.4 checkGUIMode

Returns `true` if the current environment is running in GUI mode or returns `false` for batch mode.

```
bool checkGUIMode();
```

3.5.5 checkDesignerMode

Returns `true` if the current environment is SoC Designer Canvas.

```
bool checkDesignerMode();
```

3.5.6 createCustomDialog

Enables creating a custom dialog in the GUI-based plugins (dialogs are not available in batch mode).

```
MxPluginDialog *createCustomDialog(const char *title);
```

3.5.7 executeBatchmodeCommand

Executes SoC Designer MxScript script commands from batch-mode plugins.

```
bool executeBatchmodeCommand (const char *command, int num, … );
```

where:

`command`	is the script command name (as a string).
`num`	is the number of parameters for this command.
…	are the parameters required by this script command.

Note

String type parameters must include the quotes (") explicitly in the string.

For example, the following calls the `openSystem` script command:

```
pluginP->executeBatchmodeCommand("openSystem",1,"\"SYS_ARM9-CX.mxp\"");
```

For details on the SoC Designer MxScript commands, see the *MxScript Reference Manual*.

Note

This function is only available in batch mode.

3.5.8 exportSPIRIT

The exportSPIRIT function:

- Generates an IP-XACT design XML file from an `mxp` file currently open in SoC Designer Canvas.
- Generates IP-XACT LGI-related files based on the generated SPIRIT XML file.

The declaration is:

```
bool exportSPIRIT(char *module_name, char *vendor, char *library, char *version,
                  char *file_path, char *lgi_path);
```

where:

`module_name`	is the VLNV Name entry.
`vendor`, `library`, `version`	are VLNV entries for the generated module.
`file_path`	is the path for the IP-XACT design XML file being generated.
`lgi_path`	is the path for the IP-XACT LGI XML file being generated. (If this is `NULL`, only the design XML file is generated.)

3.5.9 getDesignerConnection

This function returns the connection information for the `index` connection, where `0 <= index < getNoDesignerConnections()`. Refers to the system for which the `getNoDesignerConnections()` was called last.

```
MxPluginDesignerConnection *getDesignerConnection(int index);
```

where `MxPluginDesignerConnection` is the structure:

```
struct MxPluginDesignerConnection{
    char *connectionName;
    char *masterInstanceName;
    char *masterPortName;
    char *slaveInstanceName;
    char *slavePortName;
};
```

Related references

3.5.13 getNoDesignerConnections on page 3-29.

3.5.10 getDllPathComponent

Returns the path to the library file of the component (DLL/SO or MXP for subsystems).

```
const char *getDllPathComponent(eslapi::CASIModule *component);
```

3.5.11 getInstanceName

Returns the name of the plugin object.

```
string  getInstanceName();
```

3.5.12 getMXVersionInfo

Returns the current version of SoC Designer.

```
void GetMxVersionInfo(int* major, int* minor, int* revision);
```

A product version number of 6.1.043 would return the following values:

```
major 6
minor 1
revision 43
```

3.5.13 getNoDesignerConnections

Returns the number of connections present in the current open system.

```
int getNoDesignerConnections(void);
```

Note

This function is provided in the SoC Designer Canvas only and is meant to replace the CASI-type connection data structure. The CASI connections information is not available in the SoC Designer tool because the SoC Designer phases such as configure, init, and interconnect are not called in the SoC Designer tool and the connections are not constructed.

To get the connections information in the SoC Designer Simulator or in the batch-mode plugins, use the CASI connection information. For example, calling `getSlaves()` for each of the master ports of the component returns the slave ports that master is connected to.

3.5.14 insertDoubleClickCallback

Returns `true` if the plugin has serviced a double-click on a selected component.

A return value of `false` executes the default behavior on the component and might, for example, bring up the component properties dialog.

```
bool insertDoubleClickCallBack(MxPluginComponentDoubleClick *dClickCallBack);
```

3.5.15 insertBatchModeCommand

Enables inserting a batch-mode script command. This command is available to the users through the `callPlugin()` function in the batch-mode scripts.

```
bool insertBatchModeCommand(const char *batchCommand,
    MxPluginCallBack *myCallback);
```

Note

This function is only available in batch mode. If inserting the command is successful, the function returns `true`.

3.5.16 insertContextMenuItem

Enables inserting a context menu for the components in SoC Designer Canvas and SoC Designer Simulator.

If you right-click a component, this menu is displayed. Clicking the menu activates the provided callback. The callback function receives the component selected as an argument and returns `true` if the insertion was successful.

```
bool insertContextMenuItem(const char *text,
    MxPluginComponentCallBack *myComponentCallback,
    const char *toolTips = "", const char *whatsThis = "");
```

3.5.17 insertCustomCallback

Inserts a custom callback function and returns `true` if the insertion was successful.

```
bool insertCustomCallback(MxPluginCustomComponentCallBack *myComponentCallback,
    MxCustomComponentCallbackType context, const char *typeComponent);
```

3.5.18 insertComponentUpdateCallback

Inserts a callback for updating a component and returns `true` if the insertion was successful.

```
bool insertComponentUpdateCallBack(MxPluginComponentUpdate *updateCallBack);
```

The actions performed by the callback are specific to the type of component and the features required by the callback developer.

3.5.19 insertItemMenu

Inserts a main menu item into SoC Designer Simulator or SoC Designer Canvas and returns `true` if the insertion was successful. You can optionally insert a toolbar button. The toolbar button can be provided as an XPM data structure in the `toolbar_pixmap` parameter.

If you press the toolbar button or select the plugin menu entry, the callback function is activated.

See the actual example GUI-mode plugin sources provided in the SoC Designer Plugins package for sample usages of this and the following functions.

```
bool insertItemMenu(const char *text, MxPluginCallBack *myCallback,
    const char *pixmap[] = NULL, const char *toolTips = "",
    const char *whatsThis = "");
```

3.5.20 insertSeparator

Inserts a separator (that is, a dividing line) in the menu and returns `true` if the insertion was successful.

```
bool insertSeparator();
```

3.5.21 loadMXPFile

Opens the system whose path is specified in the parameter and returns `true` if opening was successful.

```
bool loadMXPFile(const char *mxpfilepath);
```

3.5.22 message

Logs a status message to the standard output file.

```
void message(const char *text);
```

3.5.23 removeDialog

Removes a custom dialog.

```
void removeDialog(MxPluginDialog *);
```

3.5.24 saveMXPFile

Saves the current open system as an MXP file and returns `true` if the save was successful.

If a null pointer is passes as the parameter, the current file is saved if there has been a change to the design.

```
bool saveMXPFile(const char *filename);
```

3.5.25 insertListenerCallBack

Inserts a callback that is called when another plugin sends a message (using `sendPluginMessage`).

You must implement the action to take a message received from other plugins. The function returns `true` if the insertion is successful.

```
bool insertListenerCallBack(MxPluginListener *listenerCallBack);
```

See the `MxPlugin.h` file for prototypes of the plugin functions.

3.5.26 sendPluginMessage

Sends information to other plugins. It is used if there are multiple plugins.

```
bool sendPluginMessage(const char *ReceiverPluginName, unsigned long cmdWord,
    unsigned long dataBuffer, unsigned long dataBufferSize, string& retStatus);
```

where:

`ReceiverPluginName`	Name of the plugin that receives the message.
`cmdWord`	Command that the plugin executes.
`dataBuffer`	Data buffer that holds additional information for the command.
`dataBufferSize`	Size of the data buffer array.
`retStatus`	Response that is set by the called plugin.

3.5.27 insertCustomProbe

Inserts a CASI Module at the connection between two transaction ports.

Note

As of SoC Designer version 7.1, this function is *obsolete* and is no longer supported.

In the plugin code, you must instantiate a CASI Module that contains a Master and Slave Transaction Port. The function places the CASI Module between the port connections.

```
bool insertCustomProbe(eslapi::CASIPortIF *portA, eslapi::CASIPortIF *portB,
    eslapi::CASIModuleIF* puProbe, eslapi::CASIPortIF *puProbeMasterPort,
    eslapi::CASIPortIF *puProbeSlavePort, string& strErrorMessage,
    bool bValueChange = false);
```

where:

`portA, portB`	are the pointers of the transaction ports at the end of the connection where the probe is to be placed.
`puProbe`	is the pointer to the user instantiated CASI Probe Module.
`puProbeMasterPort`	is the pointer to the Transaction Master Port of the `puProbe` module.
`puProbeSlavePort`	is the pointer to the Transaction Slave Port of the `puProbe` module.
`strErrorMessage`	is the value to be checked in case the function fails and returns a `false` value.
`bValueChange`	specifies where the probe changes the value of the transactions.

3.5.28 removeCustomProbe

Removes the Probe component inserted by the insertCustomProbe command.

Note

As of SoC Designer version 7.1, this function is *obsolete* and is no longer supported.

```
bool removeCustomProbe(eslapi::CASIModuleIF* puProbe, string& strErrorMessage);
```

where:

puProbe is the pointer to the CASI Probe module specified in the insertCustomProbe.

strErrorMessage is the value to be checked in case the function fails and returns a `false` value.

3.5.29 insertCustomIcon

Inserts an icon on a Designer Connection.

Note

As of SoC Designer version 7.1, this function is *obsolete* and is no longer supported.

```
bool insertCustomIcon(eslapi::CASIModuleIF* puComponentA,
    eslapi::CASIPortIF *portA, eslapi::CASIModuleIF* puComponentB,
    eslapi::CASIPortIF *portB, const char **pixmap,
    string& strErrorMessage);
```

where:

puComponentA, puComponentB are the pointers of the CASI module at each end of the connection.

portA, portB are the pointers of the transaction port at each end of the connection.

pixmap is the pointer to the XPM data structure. Only 16X16 bitmap images are enabled to display as icons.

strErrorMessage is the value to be checked in case the function fails and returns a `false` value.

3.5.30 removeCustomIcon

Removes the icon inserted by the insertCustomIcon command.

Note

As of SoC Designer, version 7.1, this function is *obsolete* and is no longer supported.

```
bool removeCustomIcon(eslapi::CASIModuleIF* puComponentA,
    eslapi::CASIPortIF *portA, eslapi::CASIModuleIF* puComponentB,
    eslapi::CASIPortIF *portB, const char **pixmap,
    string& strErrorMessage);
```

where:

puComponentA, puComponentB are the pointers of the CASI module at each end of the connection.

portA, portB are the pointers of the transaction port at each end of the connection.

pixmap is the pointer to the XPM data structure that was specified when the custom icon was created.

strErrorMessage is the value to be checked in case the function fails and returns a `false` value.

Note

To see instructive, detailed examples that use probe and icon functions, see the Custom Probe sample plugins found under the `example` directory of your SoC Designer installation.

Related references

Chapter 4 Creating GUI-mode Plugins on page 4-41.

3.6 The MxPluginCallBack

If the plugin developer inserts a menu item or a script command for a plugin, the developer must also provide a callback function to be called when this menu or script command is activated. This function must contain the behavior for this plugin.

The `MxPluginCallBack` is the interface for such callback functions for main menu plugin entries and for plugin script commands.

To create such a callback:

1. Derive a user class from `MxPluginCallBack`
2. Implement the `pluginCallback()` function. This is a pure virtual function.

```
class  MxPluginCallBack {
public:
    virtual bool pluginCallback(void) = 0;
};
```

This could be implemented, for example, as:

```
class myCallback: public MxPluginCallBack
{
public:
    myCallback(MxPlugin *_pluginP);
    ~myCallback();
    virtual bool pluginCallback(void);
    void printHierarchyModule(CASIModule *module, int deep);
private:
    MxPlugin *pluginP;
};

myCallback::myCallback(MxPlugin *_pluginP)
{
    pluginP = _pluginP;
}

bool myCallback::pluginCallback(void)
{
    //Get the pointer to the CADI interface
    MxSimAPI *eslapi = getMxSimAPI();
    CASIModule *top = dynamic_cast<CASIModule
        *>(eslapi->getTopComponent());
    //Print out the component hierarchy starting from the top component
    pluginP->message("Hierarchy :");
    …
}
```

3. Provide the pointer to an object of this new class to the `insertMenuItem` or `insertBatchModeCommand` function.

3.7 The MxPluginComponent callback classes

The `MxPluginComponent` classes provide the interface for the component context-menu plugin entries. The callback functions enable SoC Designer to pass the selected component as an argument to the plugin.

MxPluginComponentCallback class

```
class  MxPluginComponentCallBack {
public:
    virtual bool pluginCallback(CASIModule *) = 0;
    virtual int checkPluginCallback(eslapi::CASIModule *) { return true;} ;
};
```

MxPluginComponentDoubleClick class

```
class WEXP MxPluginComponentDoubleClick {
public:
    virtual bool pluginDoubleClickCallBack(eslapi::CASIModule *) = 0;
};
```

MxPluginComponentUpdate class

```
class WEXP MxPluginComponentUpdate {
public:
    virtual bool componentChangedCallBack(eslapi::CASIModuleIF *oldComponent,
         eslapi::CASIModuleIF *newComponent) = 0;
    virtual bool componentDeletedCallBack(eslapi::CASIModuleIF *component) = 0;
};
```

MxPluginCustomComponentCallback class

```
class WEXP MxPluginCustomComponentCallBack {
public:
    virtual bool pluginCallback(eslapi::CASIModule *, va_list vaList) = 0;
};
```

3.8 The MxPluginDialog class

The `MxPluginDialog` class enables creating basic custom dialogs in the GUI-mode plugins (for example, main menu and context menu plugin entries).

The class definition is listed in the following example.

MxPluginDialog data structure

```
class MxPluginDialog {
public:
    MxPluginDialog(const char *title);
    const char *getTitle();

    //Add parameters to the dialog
    bool addStringParameter(const char *name, const char *text,
        const char *initValue, bool editable = true);
    bool addIntegerParameter(const char *name, const char *text,
        int initValue, bool editable = true);
    bool addFloatParameter(const char *name, const char *text,
        float initValue, bool editable = true);
    bool addBooleanParameter(const char *name, const char *text,
        bool initValue, bool editable = true);

    //Execute the dialog. Returns true when the OK button was pressed
    //false otherwise
    bool executeDialog(void);

    //Get the number of parameters for the dialog
    int getNoParam();

    //Get the parameter information
    MxPluginDialogParameter *getParam(int index);
    MxPluginDialogParameter *getParam(const char *name);
    const char *getStringParameterValue(const char *name);
    bool getIntegerParameterValue(const char *name, int *value);
    bool getFloatParameterValue(const char *name, double *value);
    bool getBooleanParameterValue(const char *name, bool *value);

private:
    string title;
    vector <MxPluginDialogParameter *>parameters;
};
```

3.9 The MxPluginDialogParameter classes

The `MxPluginDialogParameter` classes enable creating entries (composed of a name and a value) and inserting them into the custom dialog.

These parameter name and value pairs are presented to you for user input. If you press the **OK** button on the dialog, the new parameter values can be retrieved and used for further processing.

`MxPluginDialogParamType` enumerates the dialog parameter types available:

```
typedef enum{
    MX_PLUGIN_DIALOG_PARAM_STRING,
    MX_PLUGIN_DIALOG_PARAM_INT,
    MX_PLUGIN_DIALOG_PARAM_FLOAT,
    MX_PLUGIN_DIALOG_PARAM_BOOL
} MxPluginDialogParamType;
```

The `MxPluginDialogParameter` class is the base class for the different parameter variants:

```
class  MxPluginDialogParameter{
public:
    MxPluginDialogParameter(const char *name, const char *text,
        bool _editable, MxPluginDialogParamType type);
    virtual MxPluginDialogParamType getType();
    virtual const char *getName();
    virtual const char *getText();
    virtual void setWidget(void *);
    virtual void *getWidget();
    virtual void setEditable(bool);
    virtual bool getEditable();
private:
    MxPluginDialogParamType type;
    string name;
    string text;
    void *widget;
    bool editable;
};
```

The `MxPluginDialogStringParameter`, `MxPluginDialogIntParameter`, `MxPluginDialogFloatParameter`, and `MxPluginDialogBoolParameter` classes represent the dialog parameters of type `string`, `int`, `float`, and `bool`. To create parameters of these types, instantiate an object of the corresponding parameter class, and provide it as an argument to the `MxPluginDialog::add`*`type`*`Parameter` function (*`type`* is `string`, `int`, `float`, or `bool`).

MxPluginDialogStringParameter

```
class  MxPluginDialogStringParameter : public MxPluginDialogParameter {
public:
    MxPluginDialogStringParameter(const char *_name, const char *text,
        bool _editable);
    string &getValue();
    void setValue(string &);
private:
    string value;
};
```

MxPluginDialogIntParameter

```
class  MxPluginDialogIntParameter : public MxPluginDialogParameter {
public:
    MxPluginDialogIntParameter(const char *_name, const char *text,
        bool _editable);
    int getValue();
    void setValue(int);
private:
    int value;
};
```

MxPluginDialogFloatParameter

```
class  MxPluginDialogFloatParameter : public MxPluginDialogParameter {
public:
    MxPluginDialogFloatParameter(const char *_name, const char *text,
        bool _editable);
```

```
    double getValue();
    void setValue(double );
private:
    double value;
};
```

MxPluginDialogBoolParameter

```
class MxPluginDialogBoolParameter : public MxPluginDialogParameter {
public:
    MxPluginDialogBoolParameter(const char *_name, const char *text,
        bool _editable);
    bool getValue();
    void setValue(bool);
private:
    bool value;
};
```

3.10 Running the example plugins

The SoC Designer plugin example folder contains the following plugin examples:

- `GuiPluginExample` shows how to create GUI-mode plugins.
- `BatchModePluginExample` shows how to create SoC Designer batch-mode plugins.

To run the example plugins provided follow these steps:

1. Install the provided SoC Designer version supporting plugins and ensure that the `MAXSIM_HOME` environment variable is pointing to the correct location.
2. If the plugin DLL/SO library is not already present in the provided package, compile the plugin sources and create the plugin DLLs/SOs.

   Note

   The release version of the DLL/SO libraries is typically already provided as part of the plugins package. If you are not debugging the source code of the component, you do not require the debug versions. This step is only shown for completeness.

3. Copy the plugin DLLs/SOs into the following directories:
   - Copy the release version DLL into `%MAXSIM_HOME%/etc/plugins/Release`.

     (for Linux, copy the SO into `$(MAXSIM_HOME)/etc/plugins/Release`).
   - Copy the debug version DLL into `%MAXSIM_HOME%/etc/plugins/Debug`.

     (for Linux, copy the SO into `$(MAXSIM_HOME)/etc/plugins/Debug`)

   Note

   The DLLs provided in the package are release version and must be placed into the `release` directory.

4. Start SoC Designer:
   - For the GUI-mode plugin example:
     1. Open SoC Designer Canvas or SoC Designer Simulator and load any `mxp` system provided as an example in the `gui_plugin_example` directory.
     2. Click the **Hierarchy** toolbar button or select **Hierarchy** from the **Plugins** menu. This launches the **Hierarchy** plugin functionality and displays the hierarchy of the current system in the Output window of SoC Designer Canvas or SoC Designer Simulator.

        Note

        To see the output window results in SoC Designer Canvas, you must explicitly open the Output window from the **Windows** menu of the SoC Designer tool.

     3. Right-click a component and select the **Plugins/View ports…** context menu. This launches the **View ports** plugin functionality that opens a dialog showing all the ports of the component selected.
     4. Right-click a component and select the **Plugins/Edit parameters…** context menu. This opens a dialog with the parameters of the component.
     5. Right-click a component and select the **Plugins/Get Dll Path** context menu. This opens a dialog and displays the path to the DLL file of the component respectively.
   - For the batch-mode example, perform one of the following:
     — Start SoC Designer Simulator using the following command:

     ```
     sdsim –script my_example.mxscr
     ```

     — Use the provided batch script called `run.bat` (Win32) or `run.csh` (Linux).

     This starts running the `my_example.mxscr` SoC Designer Simulator script that invokes the example batch-mode plugin.

The plugin:

1. Opens the `MxStub_example.mxp` system.
2. Displays the hierarchy.
3. Runs the simulation.
4. Displays some CAPI profiling results.
5. Saves the system under a different MXP file.
6. Opens or runs this other system.

Note

This functionality is described as part of the example plugins that use the provided SoC Designer interfaces. Arm recommends browsing the example plugin source code and using it as a template for your plugin development.

Chapter 4
Creating GUI-mode Plugins

This chapter describes how to create GUI-mode plugins.

Note

For more information on batch mode classes, see *Chapter 6 Creating Batch-Mode Plugins* on page 6-62.

It contains the following sections:

4.1 Creating the MxPluginInit function for GUI mode

GUI-mode plugins enable creating main menu items and component context-menu items in the SoC Designer Canvas and SoC Designer Simulator tools. The first steps required to create a GUI-mode plugin is to implement `MxPluginInit()`.

The following example shows an example `MxPluginInit()` implementation that sets up and initializes the plugin by, for this example, creating the plugin interface and inserting menu items.

Note

To export the `MxPluginInit` function from the dynamic library (DLL) on Win32, the function must be added to the `EXPORTS` section of the Microsoft Visual C++ `DEF` file.

MxPluginInit implementation

```
extern "C" void
MxPluginInit(void)
{
    // create the MxPlugin interface
    *pluginP = new MxPlugin("fabric_config");
    // log mode information
    if (pluginP->checkDesignerMode())
    {
        pluginP->message("DESIGNER MODE  ");
    }
    else
    {
        pluginP->message("EXPLORER MODE  ");
    }

    if (pluginP->checkGUIMode()) pluginP->message("GUI MODE  ");

    // create the callback object for the main menu plugin
    hierarchyCallback = new myCallback(pluginP);

    // Insert an item named "Hierarchy" in the main menu
    // When this item is selected, callback->pluginCallback() is called
    pluginP->insertItemMenu("Hierarchy",
        dynamic_cast<MxPluginCallBack*>(hierarchyCallback), Hierarchy);

    // create portsComponentCallback object for the ports-related context menu
    portsComponentCallback *portsCallback =         new
portsComponentCallback(pluginP);

    // Insert an item named "View ports…" in the context menu for a component
    // When this item is selected,
    // portsComponentCallback->pluginCallback(component) is called,
    // where component is the actual selected component
    pluginP->insertContextMenuItem("View ports…", portsCallback);

    // create paramComponentCallback object for the parameters-related
    // context menu
    paramComponentCallback *paramCallback = new paramComponentCallback(pluginP);

    // Insert an item named "Edit parameters…" in the context menu for
    // a component.  When this item is selected,
    // paramComponentCallback->pluginCallback(component) is called,
    // where component is the actual selected component
    pluginP->insertContextMenuItem("Edit parameters…", paramCallback);
    // register the context menu to print the path to the component's DLL
    dllPathCallback *dllCallback = new dllPathCallback(pluginP);
    pluginP->insertContextMenuItem("Get Dll path", dllCallback);

    // create a callback for selecting an item
    doubleClickCallBack *dClickCallBack = new doubleClickCallBack(pluginP);
    pluginP->insertDoubleClickCallBack(dClickCallBack);

    // create a callback for updating the component

    updateCallBack *uCallBack = new updateCallBack(pluginP);
    pluginP->insertComponentUpdateCallBack(uCallBack);
}
```

4.2 Plugin callback examples

List of example plugins using the sample file `gui_plugin.h`.

This section contains the following subsections:

4.2.1 MxPluginCallBack

The following example shows a plugin callback specification (from the sample file `gui_plugin.h`) for a main menu item:

MxPluginCallBack

```
// This class is used to create a general callback function accessed through
// a menu item from the SoC Designer main menu
class myCallback: public MxPluginCallBack
{
public:
    myCallback(MxPlugin *_pluginP);
    ~myCallback();
    virtual bool pluginCallback(void);
    void printHierarchyModule(CASIModule *module, int deep);

private:
    MxPlugin *pluginP;
};
```

4.2.2 MxPluginComponentCallBack for ports

The following example shows a port callback specification (from the sample file `gui_plugin.h`) for a context menu item:

MxPluginComponentCallBack for ports

```
// This class is used to create a component callback function accessed
// through a context menu item for the selected component
class portsComponentCallback: public MxPluginComponentCallBack
{
public:
    portsComponentCallback(MxPlugin *);
    ~portsComponentCallback();
    bool pluginCallback(CASIModule *selectedComponent);
private:
    MxPlugin *pluginP;
};
```

4.2.3 MxPluginComponentCallBack for parameters

The following example shows a parameter callback specification (from the sample file `gui_plugin.h`):

MxPluginComponentCallBack for parameters

```
// This class is used to create a component callback function accessed through a
// popup menu item for the selected component
class paramComponentCallback: public MxPluginComponentCallBack
{
public:
    paramComponentCallback(MxPlugin *);
    ~paramComponentCallback();
    bool pluginCallback(eslapi::CASIModule *selectedComponent);
private:
    MxPlugin *pluginP;
};
```

4.2.4 MxPluginComponentCallBack for the path of a DLL

The following example shows a callback specification that enables the path of a DLL to be determined (from the sample file `gui_plugin.h`):

MxPluginComponentCallBack for DLL

```
class dllPathCallback: public MxPluginComponentCallBack
{
public:
    dllPathCallback(MxPlugin *);
    ~dllPathCallback();
    bool pluginCallback(eslapi::CASIModule *selectedComponent);
    int checkPluginCallback(eslapi::CASIModule *);
private:
    MxPlugin *pluginP;
};
```

4.2.5 MxPluginComponentCallBack for trapping a double-click

The following example shows a callback specification that traps a double-click on a component (from the sample file `gui_plugin.h`):

MxPluginComponentCallBack for DLL

```
// This needs to be registered with the plugin using the
// insertDoubleClickCallBack. The Callback function pluginDoubleClickCallBack
// is called whenever there is a double click on the selectedComponent.
// If the value returned is true for pluginDoubleClickCallBack,
// then the default functionality for double click in SoC Designer
// is not executed.

class doubleClickCallBack:public MxPluginComponentDoubleClick
{

public:
    doubleClickCallBack(MxPlugin *);
    ~doubleClickCallBack();
    bool pluginDoubleClickCallBack(eslapi::CASIModule *selectedComponent);
private:
    MxPlugin *pluginP;
};
```

4.2.6 MxPluginComponentCallBack for for trapping a component update

The following example shows a callback specification that traps a component update (from the sample file `gui_plugin.h`):

MxPluginComponentCallBack for DLL

```
//Callback class. This needs to be registered with the plugin using the
// insertComponentUpdateCallBack. The Callback function
// pluginDoubleClickCallBack is called whenever the backend reference pointer
// changes values.
class updateCallBack:public MxPluginComponentUpdate
{
public:
    updateCallBack(MxPlugin *);
    ~updateCallBack();
    bool componentChangedCallBack(eslapi::CASIModuleIF *oldComponent,
                                  eslapi::CASIModuleIF *newComponent);
    bool componentDeletedCallBack(eslapi::CASIModuleIF *component);
private:
    MxPlugin *pluginP;
};
```

4.2.7 listenerCallBack for receiver plugins

The following example shows a callback specification that traps a component update (from the sample file `gui_plugin.h`):

listenerCallBack for receiver plugins

```
class listenerCallBack:public MxPluginListener
{
public:
    listenerCallBack(MxPlugin *);
    ~listenerCallBack();
    bool pluginListenerCallBack(const char *SenderPluginName,
            unsigned long cmdWord,         unsigned long dataBuffer,
            unsigned long dataBufferSize, string& retStatus);
private:
    MxPlugin *pluginP;
};
```

4.3 Behavior of plugin callbacks

List of examples on callback functions and other related functions.

This section contains the following subsections:

4.3.1 Callback function

The following example shows the implementation of the behavior for a `pluginCallback` function:

Callback function

```
bool myCallback::pluginCallback(void)
{
    //Get the pointer to the CADI interface
    MxSimAPI *eslapi = getMxSimAPI();
    int major = 0;
    int minor = 0;
    int revision = 0;
    pluginP->GetMxVersionInfo(&major, &minor, &revision);
    char buffer1[256];
    sprintf(buffer1, "ESL API Version : MAJOR: %d MINOR = %d REVISON = %d",
         major, minor, revision);
    pluginP->message(buffer1);

    CASIModule *top = dynamic_cast<CASIModule *>(eslapi->getTopComponent());

    //Print out the component hierarchy starting from the top component
    pluginP->message("Hierarchy :");
    printHierarchyModule(top, 0);

    //Display the connections in the SoC Designer tool
    pluginP->message("The connections in this system: ");
    int noConnections = pluginP->getNoDesignerConnections();
    for(int i = 0; i < noConnections; i++){
        MxPluginDesignerConnection *connection =
                  pluginP->getDesignerConnection(i);
        char buffer[256];
        sprintf(buffer, "\tConnection name: %s, Connection: %s::%s<-->%s::%s",
             connection->connectionName, connection->masterInstanceName,
             connection->masterPortName, connection->slaveInstanceName,
             connection->slavePortName);
        pluginP->message(buffer);
    }
    return true;

}
```

4.3.2 Printing the hierarchy

The following example shows the implementation of a helper function for use by the callback:

Printing the hierarchy

```
//This function prints the component hierarchy starting from a top-level module
void myCallback::printHierarchyModule(CASIModule *module, int deep)
{
    char buff[100];
    char tabs[100] = "";
        if(module == NULL){
        return;
    }

    for(int i = 0; i < deep; i++){
        strcat(tabs, "\t");
    }

    //Print out the module instance name
    sprintf(buff,"%s %s",tabs, module->getInstanceName().c_str());
    pluginP->message(buff);

    //Call the same function recursively for sub-modules of this component
    const vector< CASIModuleIF* >& uList = module->getSubcomponentList();
    vector<CASIModuleIF*>::const_iterator uIter;
    for ( uIter = uList.begin(); uIter != uList.end(); uIter++ ) {
        printHierarchyModule(dynamic_cast<CASIModule *>((*uIter)), deep + 1);
    }
}
```

4.3.3 Creating parameter callback

The following example shows a parameter callback implementation (from the sample file `gui_plugin.h`):

Creating parameter callback

```
// the actual callback function which is executed from the popup menu
// for a particular component
bool
paramComponentCallback::pluginCallback(eslapi::CASIModule *_selectedComponent)
{
    vector <string> paramNames;
    vector <eslapi::CASIParameterType> paramTypes;
    string newValue, oldValue;
    CASIModule *selectedComponent = (CASIModule *)_selectedComponent;

    if(_selectedComponent == NULL) return false;

    //Create a custom dialog
    MxPluginDialog *dialog = pluginP->createCustomDialog("Edit parameters");

    //Add a string type parameter entry to this dialog, called componentName
    dialog->addStringParameter("componentName", "Selected component:",
         selectedComponent->getInstanceName().c_str(), false);

    //Add parameter entries into the dialog for every CADI parameter
    // of this component
    string key, value;
    eslapi::CASIParameterProperties prop;
    const CASIParameterMapIF *paramList = selectedComponent->getParameterList();
    bool flag = paramList->getFirst (key, value, prop );
    while ( flag ) {
        string value = selectedComponent->getParameter(key);
        switch(prop.type){
        case eslapi::CASI_PARAM_STRING:
        case eslapi::CASI_PARAM_UNDEFINED:
        case eslapi::CASI_PARAM_VALUE:
            dialog->addStringParameter(key.c_str(), key.c_str(),
                    value.c_str(), !prop.is_readonly && prop.is_runtime);
            break;
        case eslapi::CASI_PARAM_BOOL:
            {
                bool val = 1;
                if(value == "false"){
                    val = 0;
                }
                dialog->addBooleanParameter(key.c_str(), key.c_str(), val,
                     !prop.is_readonly && prop.is_runtime);
            }
            break;
        }
        paramNames.push_back(key);
        paramTypes.push_back(prop.type);
        flag = paramList->getNext( key, value, prop, true );
    }

    //Execute the dialog
    if(dialog->executeDialog()){
        //If the OK button was pressed on the dialog

        //Make the parameter modifications
        const CASIParameterMapIF *paramList =
            selectedComponent->getParameterList();
        int noParam = paramNames.size();
        for(int i = 0; i < noParam; i++){
            string value = selectedComponent->getParameter(paramNames[i]);

            switch(paramTypes[i]){
            case eslapi::CASI_PARAM_STRING:
            case eslapi::CASI_PARAM_UNDEFINED:
            case eslapi::CASI_PARAM_VALUE:
                {
                    MxPluginDialogParameter * param =
                            dialog->getParam(paramNames[i].c_str());
                    if(param->getType() == MX_PLUGIN_DIALOG_PARAM_STRING){
                        const char *valueParam =
                        (dynamic_cast<MxPluginDialogStringParameter*>(param))
                                ->getValue().c_str();
                        newValue = valueParam;
                        string paramName = paramNames[i];
                        oldValue = selectedComponent->getParameter(paramName);
                        if(oldValue != newValue){
                        //if the value has been changed, set it in the component
                        selectedComponent->setParameter(paramName, newValue);
                        }
                    }
                }
                break;
```

```
            case eslapi::CASI_PARAM_BOOL:
                {
                    bool boolValue;
                    string paramName = paramNames[i];

                    MxPluginDialogParameter * param =
                         dialog->getParam(paramNames[i].c_str());
                    if(param->getType() == MX_PLUGIN_DIALOG_PARAM_BOOL){
                        boolValue =
                        (dynamic_cast<MxPluginDialogBoolParameter*>(param))
                            ->getValue();
                        char buff[10];
                        if(boolValue){
                            strcpy(buff, "true");
                        }
                        else{
                            strcpy(buff, "false");
                        }
                        newValue = buff;
                        oldValue = selectedComponent->getParameter(paramName);
                        if(oldValue != newValue){
                        //if the value has been changed, set it in the component
                          selectedComponent->setParameter(paramName, newValue);
                        }
                    }
                }
                break;
            }
        }
    }
    paramNames.clear();
    paramTypes.clear();

    return true;
}
```

4.3.4 Displaying callback messages

You can add status messages to some of the callback functions to show the different stages of the plugin execution as shown in the following example.

Displaying callback messages

```
dllPathCallback::pluginCallback(eslapi::CASIModule *_selectedComponent)
{
    string newValue, oldValue;
    CASIModule *selectedComponent = (CASIModule *)_selectedComponent;

    if(_selectedComponent == NULL) return false;

    const char *dllPath = pluginP->getDllPathComponent(selectedComponent);
    pluginP->message(dllPath);

    return true;
}

int dllPathCallback::checkPluginCallback(eslapi::CASIModule
                *_selectedComponent)
{
    pluginP->message("CHECK PLUGIN CALLBACK CALLED returned false");
    return 0;
    //return 1 if the Context Menu Item has to be displayed.
}

bool doubleClickCallBack::pluginDoubleClickCallBack(eslapi::CASIModule
                *selectedComponent)
{
    char buf[500];
    sprintf (buf,"pluginDoubleClickCallBack called for component %s",
            selectedComponent->getInstanceName().c_str());
    pluginP->message(buf);
    return true;
}

bool  updateCallBack::componentChangedCallBack(eslapi::CASIModuleIF
     *oldComponent, eslapi::CASIModuleIF *newComponent)
{
    char buf[500];
    sprintf (buf,"componentChangedCallBack called for component %s",
            newComponent->getInstanceID().c_str());
    pluginP->message(buf);
    return true;
}

//Note this is called just before the component is deleted.
bool updateCallBack::componentDeletedCallBack(eslapi::CASIModuleIF *component)
{
    char buf[500];
    sprintf (buf,"componentDeletedCallBack called for component %s",
            component->getInstanceID().c_str());
    pluginP->message(buf);
    return true;
}
```

4.3.5 Constructors and destructors

The following example shows the constructors and destructors that are required for the classes.

Constructors and destructors

```
myCallback::myCallback(MxPlugin *_pluginP)
{
    pluginP = _pluginP;
}
myCallback::~myCallback()
{}

dllPathCallback::dllPathCallback(MxPlugin *_pluginP)
{
    pluginP = _pluginP;
}
dllPathCallback::~dllPathCallback()
{}

doubleClickCallBack::doubleClickCallBack(MxPlugin *_pluginP)
{
    pluginP = _pluginP;
}
doubleClickCallBack::~doubleClickCallBack()
{}

updateCallBack::updateCallBack(MxPlugin *_pluginP)
{
    pluginP = _pluginP;
}
updateCallBack::~updateCallBack()
{}

paramComponentCallback::paramComponentCallback(MxPlugin *_pluginP)
{
    pluginP = _pluginP;
}
paramComponentCallback::~paramComponentCallback()
{}

portsComponentCallback::portsComponentCallback(MxPlugin *_pluginP)
{
    pluginP = _pluginP;
}
portsComponentCallback::~portsComponentCallback()
{}
```

4.4 Place the plugin dynamic library into etc/plugins directory

The plugin dynamic library must be placed into the *MAXSIM_HOME*`/etc/plugins/Release` or *MAXSIM_HOME*`/etc/plugins/Debug` directory.

Note

If the environment variables have been set, replace *MAXSIM_HOME* with:

- `%MAXSIM_HOME%` Windows systems
- `$(MAXSIM_HOME)` for Linux

On startup, SoC Designer by default searches all the plugin libraries in this directory and activates the corresponding plugins.

Note

If you are running the release SoC Designer version, the release version of the plugin DLL is used. Similarly for the debug SoC Designer version, the debug plugin is required.

4.5 Run the plugin behavior

After the plugin has been placed into `etc/plugins`, SoC Designer shows the inserted menu items in the **Plugins** main menu entry and the component context menu items in the component context menu.

To test the plugins behavior, click the menu items. This activates the plugin callback functions.

You can attach a host level debugger to debug such plugins in the same manner as for debugging SoC Designer components. For example, right-click on any SoC Designer component and select **Launch host level debugger**. You can then set breakpoints in the plugin source code.

4.6 Checklist for plugin Troubleshooting

If a plugin does not load, ensure that the following plugin preparations have been performed correctly:

- An `MxPluginInit()` function is defined (see *6.1 Create the MxPluginInit function* on page 6-63).
- Required callback specifications are specified (see *4.2 Plugin callback examples* on page 4-43).
- If you load plugins with the `$MAXSIM_PLUGINS` environmental variable, the check box in the **SoC Designer Preferences** > **Component Library** menu must set **Include plugins from $MAXSIM_PLUGINS** (see *3.2 Loading a plugin* on page 3-21).

Chapter 5
MxLCD Interface Specification

This chapter describes all classes and member functions of the MxLCD API.

The MxLCD interfaces are a set of C++ interfaces that cover control interfaces for LCD display and keyboard and mouse captures.

It contains the following sections:

5.1 MxLCD interface classes

This class provides the main interface to MxLCD. A SoC Designer component can instantiate one or more MxLCD interfaces and use the appropriate access functions to control the LCD window. A unique name must be assigned to each MxLCD instance. This name is displayed on the LCD window title bar.

A Keyboard/Mouse Interface (KMI) object can be associated with each LCD. The KMI uses the LCD name to identify the LCD that it is bound to. If an LCD has focus, keyboard and mouse input is sent to the associated KMI object.

The definition of the MxLCD class is listed in the following example.

MxLCD interface class

```
class MxLCD : public MxLCDDisplayInterface
{
public:
    MxLCD();
    // use this constructor if the name is already known by the component
    // name is the identifier for the interface class
    MxLCD( string name);

    // convenience functions for DISPLAY I/F, typically called from init
    // open/close the display window
    void open();
    void close();

    // schedule a display update (recommended)
    void update( bool paint_all = false );
    // set all display parameters at once (e.g. default initialization)
    MxLCDStatus setGlobalParameters(int x, int y, int pixelmau,
              int clrtablemau, double zoom, unsigned char*
              memorybuffer, uint32_t addressspace, unsigned char*
               colortablebuffer, AddressMode addrmode = RGB24 );
    // set display width
    MxLCDStatus setWidth( int x ) ;
    // set display height
    MxLCDStatus setHeight( int y );
    // set address mode
    MxLCDStatus setAddressMode( AddressMode addrmode );
    // set minimum addressable unit for display memory
    MxLCDStatus setPixelMAU( int pxmau );
    // set minimum addressable unit for color table memory
    MxLCDStatus setColorTableMAU( int clrmau );
    // set zoom factor
    MxLCDStatus setZoom( double zoom );
    // set the memory buffer from which display data is read.
    void setMemoryBuffer( unsigned char* buf, uint32_t addressspace );
    // set the pointer to color table memory
    void setColorTableBuffer( unsigned char* buf );

    // MOUSE I/F
    // get mouse x position
    int getMouseX();
    // get mouse y position
    int getMouseY();

    // register mouse callback interface class
    // the mouse callback functions have to be thread-safe
    void registerMouseCallback( MouseCallbackInterface* mouseCallback );

    // KEYBOARD I/F
    // register keyboard callback interface class
    void registerKeyboardCallback( KeyboardCallbackInterface*
    keyboardCallback, MxLCDKeyCallbackType type );

    // GENERAL DISPLAY I/F
    // display name
    void setName( string name );
   // get the list w/ all available LCD Displays
    static const vector<MxLCD*>* getMxLCDList();
    // get display id
    const int getID() const { return id; }
    // get display name
    const string& getName() const { return name; }
    // release mouse from display
    void releaseMouse();
    // set the used scancode set, 1 and 2 is supported by MxLCD,
    // component has to reject scancode request of type 3
    void setScanCodeSet( uint8_t currentScanSet );
    // The MxLCD window is updated the same frequency like e.g. the
    // cycle counter. To get slower update frequency, increase division
    // factor (may result in faster simulation)
    void changeUpdateDivider( uint32_t div );
};
```

5.1.1 MxLCDStatus

`MxLCDStatus` is an enumeration with the following values:

- `MXLCD_STATUS_OK`
- `MXLCD_STATUS_ERROR`
- `MXLCD_STATUS_MEMORY_TOOSMALL`

5.1.2 Keyboard Callback Interface

A SoC Designer component implementing a keyboard interface must derive off one of the following callback interfaces:

- `KeyboardCallbackInterfaceScanCode`
- `KeyboardCallbackInterfaceASCII`
- `KeyboardCallbackInterfaceUnicode`

When a keyboard button is pressed while the LCD window has focus, the corresponding callback function in the keyboard interface component is called by MxLCD.

A keyboard interface component must register itself with MxLCD during initialization by:

1. Search all available MxLCD instances with `getMxLCDList()`
2. Find the correct instance by comparing the name. This can be done with `MxLCD::getName()`.
3. Register itself by using the function `registerKeyboardCallback()`. See the following examples.

Keyboard callback for scan code

```
// using PS/2 scancode
class KeyboardCallbackInterfaceScanCode : public KeyboardCallbackInterface
{
public:
    virtual void sendMakeCode (uint8_t makecode ) = 0;
    virtual void sendBreakCode (uint8_t breakcode ) = 0;
};
```

Keyboard callback for ASCII

```
// using ASCII code
class KeyboardCallbackInterfaceASCII : public KeyboardCallbackInterface
{
public:
    virtual void keyPressed (int ascii, MxLCDKeyState keystate ) = 0;
    virtual void keyReleased (int ascii, MxLCDKeyState keystate  ) = 0;
};
```

Keyboard callback for Unicode

```
// using UNICODE
class KeyboardCallbackInterfaceUniCode : public KeyboardCallbackInterface
{
public:
    virtual void keyPressed (string unicode, MxLCDKeyState keystate ) = 0;
    virtual void keyReleased (string unicode, MxLCDKeyState keystate  ) = 0;
};
```

Registering the keyboard callback

`MxLCDKeyState` is an enumeration with the following values:

- `NoButton`
- `LeftButton`
- `RightButton`
- `MidButton`
- `MouseButtonMask`
- `ShiftButton`
- `ControlButton`
- `AltButton`
- `MetaButton`
- `KeyButtonMask`
- `Keypad`

Depending on the `MxLCDKeyCallbackType` used during the callback registration, MxLCD sends PS/2 compatible bytes, ASCII, or Unicode. The callback functions have to be thread safe as they are called from outside the simulation thread.

Registering the keyboard callback

```
// callback registration with MxLCD
registerKeyboardCallback(KeyboardCallbackInterface* kbdCB,
          MxLCDKeyCallbackType type);
```

`MxLCDKeyCallbackType` is an enumeration with the following values:

- `MXLCD_SCANCODE`
- `MXLCD_UNICODE`
- `MXLCD_ASCII`

5.1.3 Mouse Callback Interface Class

A SoC Designer component implementing a mouse interface must be derived from this class. When a mouse button is pressed while the LCD window has focus, the `sendMovementByte()` callback function is called by MxLCD.

MxLCD sends PS/2 compatible movement bytes to the mouse controller. The callback function must be thread safe as it is called from outside of the simulation thread.

A mouse interface component registration with MxLCD is done in the same way as described in *5.1.2 Keyboard Callback Interface* on page 5-58.

Mouse callback

```
class MouseCallbackInterface
{
public:
    virtual void sendMovementByte (uint8_t data ) = 0;
};
```

Registering the mouse callback

```
// callback registration with MxLCD
registerMouseCallback(MouseCallbackInterface* mouseCB);
```

5.2 MxLCD display implementation example

Example code in this section shows how to add a custom MxLCD display from a SoC Designer component.

This section contains the following subsections:

5.2.1 MxLCD includes

The SoC Designer component header file must include `MxLCD.h` header file. This file is located under:

- `$(MAXSIM_HOME)/include` for Linux.
- `%MAXSIM_HOME%\include` for Windows.

5.2.2 MxLCD implementation in SoC Designer component

List of code examples describing the different steps of MxLCD implementation.

Instantiation of MxLCD in init()

The component must instantiate MxLCD in its `init()` function. Opening the LCD display window can be done in `init()` in addition or can optionally be deferred to a later stage.

```
lcdInterface = new MxLCD(“MyDisplay”);
// use values obtained from setParameter() to configure the LCD display
// (alternatively, simply use hard-coded values)
MxLCDStatus status = lcdInterface->setGlobalParameters (… );
// prepare display buffer
updateDisplayBuffer();
// open the display
lcdInterface->open();
terminate()
```

Closing the LCD window

The component must close the LCD window (if not closed already) and de-allocate any memory associated with the MxLCD instance.

```
lcdInterface->close();
if  (lcdInterface )
    delete lcdInterface;
    setParameter()
```

Setting CD parameters and zoom factor

Arm recommends that you add a set of MxLCD related parameters to the component for LCD display configuration.

The following examples show such parameter settings.

Setting LCD parameters

```
if (name == "width" )
    {
        // configure the width of LCD display
        status = CASIConvertStringToValue (value, &sizex  );
        // “sizex” to be used in setGlobalParameters in init(), or
        if (initCompleted)
        {
            // changing the width dynamically during run-time
            // this involves updating the display buffer size
            updateDisplayBuffer();
            MxLCDStatus lcdStat = lcdInterface->setWidth(sizex);
        }
}
```

Setting zoom factor

```
if (name == "zoom factor" )
    {
        // set zoom factor
        status = CASIConvertStringToValue (value, &zoom );
        // “zoom” value can be used in setGlobalParameters in init(), or
        if (initCompleted)
        {
            // changing the zoom factor dynamically during run-time
            MxLCDStatus lcdStat = lcdInterface->setZoom(zoom);
        }
}
updateDisplayBuffer()
```

LCD buffering

It is important to maintain a display buffer representing the correct size of data to be displayed on the LCD window. MxLCD reads the amount of data from this buffer according to the display size configurations.

`updateDisplayBuffer()` is an example function that dynamically adjusts the display buffer to the current size configurations. See the following example.

```
// delete old buffer
lcdInterface->setMemoryBuffer (0, 0 );
if  (memBuf )
delete memBuf;

// prepare new buffer according to new settings
int width_mult = 0;
switch  (addressMode )
{
   case RGB32:
        width_mult = sizex * 4 * pixelMAU;
        break;
   case RGB24:
        width_mult = sizex * 3 * pixelMAU;
        break;
   case BGR24:
        width_mult = sizex * 3 * pixelMAU;
        break;
   case GRAYSCALE8:
        width_mult = sizex * pixelMAU;
        break;
    case COLORTABLE8:
        width_mult = sizex * pixelMAU;
        break;
}
addressSpace = width_mult * sizey;
// allocate/initialize buffer
memBuf = new unsigned char[addressSpace];
memset(memBuf, 0, addressSpace);
// pass the new buffer to MxLCD
lcdInterface->setMemoryBuffer (memBuf, addressSpace );
```

Chapter 6
Creating Batch-Mode Plugins

This chapter describes how to create batch-mode plugins.

It contains the following sections:

6.1 Create the MxPluginInit function

Batch-mode plugins enable creating batch-mode script commands to be used from the script-based batch-mode SoC Designer execution. The following steps are required to create a batch-mode plugin:

The following example shows an `MxPluginInit()` implementation. This function sets up and initializes the plugin for procedures that include creating the plugin interface and inserting script commands.

Note

To export the `MxPluginInit()` function from the dynamic library (DLL) on Win32, the function must be added to the EXPORTS section of the Microsoft Visual C++ `DEF` file.

MxPluginInit implementation

```
// This function is called from the ESL API system at the beginning, here is the
place to
// create the objects and set the callbacks
extern "C" void
MxPluginInit(void)
{
    //Create the plugin interface
    MxPlugin *pluginP = new MxPlugin("batchmode_plugin_example");

    //create the callback
    myBatchModeCallback *batchModeCallback =
        new myBatchModeCallback(pluginP);

    //insert the batch command
    pluginP->insertBatchModeCommand("get_hierarchy", batchModeCallback);
}
```

6.2 Create the plugin callbacks

The following example shows a plugin callback structure for a main menu item and a context menu item:

Callback structure

```
class myBatchModeCallback: public MxPluginCallBack
{
public:
    myBatchModeCallback(MxPlugin *_pluginP);
    ~myBatchModeCallback();
    virtual bool pluginCallback(void);
    …
private:
    MxPlugin *pluginP;
};
```

6.3 Add the plugin behavior to the plugin callbacks

The following example shows the behavior for a callback function:

```
//The actual plugin callback function, called when the user enters the plugin
//command in a batch-mode script
bool
myBatchModeCallback::pluginCallback(void)
{
    //Get the pointer to the CADI interface
    MxSimAPI *eslapi = getMxSimAPI();

    //Open the system:
    pluginP->executeBatchmodeCommand("setAppFile",2,"\"core0\"",
        "\"dlx0.elf\"" );
    pluginP->executeBatchmodeCommand("setAppFile",2,"\"core1\"",
        "\"dlx1.elf\"" );
    pluginP->executeBatchmodeCommand("openSystem",1,"\"SYS_ARM9-CX.mxp\"");

    //Get the top-level component
    CASIModule *top = dynamic_cast<CASIModule *>(eslapi->
        getTopComponent());

    //Start printing out the hierarchy
    pluginP->message( "Hierarchy:\n" );
    printHierarchyModule(top, 0);

    //Run the simulation a bit
    pluginP->executeBatchmodeCommand("step",1,"10000");

    //Save the system to an MXP file:
    bool res = pluginP->saveMXPFile("SYS_ARM9-CX-saved.mxp");
    if(res == false){
        pluginP->message("There was an error saving the MXP file.");
    }

    //Close the simulation:
    pluginP->executeBatchmodeCommand("closeSystem",0);

    //Open another system:
    pluginP->executeBatchmodeCommand("openSystem",1,"\"SYS_ARM9-CX-saved.mxp\"");

    //Run the simulation a bit
    pluginP->executeBatchmodeCommand("step",1,"10000");

    //Close the simulation:
    pluginP->executeBatchmodeCommand("closeSystem",0);
    return true;
}

//This function prints out the hierarchy starting from a top-level
//component
void myBatchModeCallback::printHierarchyModule(CASIModule *module, int deep)
{
    char buff[100];
    char tabs[100] = "";

    if(module == NULL){
        return;
    }
    for(int i = 0; i < deep; i++){
        strcat(tabs, "\t");
    }

    sprintf(buff,"%s Module: %s",tabs, module->getInstanceName().c_str());
    pluginP->message(buff);
    pluginP->message("\n");

    //Print out all the ports for this module
    sprintf(buff,"%s Ports:",tabs);
       pluginP->message(buff);
    const CASIPortMapIF *ports = module->getPortList();
    string strPortName;
    CASIPortIF*        puInterface;
    while ( ports->getNext( strPortName, &puInterface ) ) {
        sprintf(buff, "%s %s", tabs, strPortName.c_str());
        pluginP->message(buff);
        sc_port<eslapi::CASITransactionIF, 1> *master_port =
            dynamic_cast<sc_port<eslapi::CASITransactionIF, 1>
              *>(puInterface);
        if(master_port != NULL){
            sprintf(buff, "%s    Slaves:", tabs);
```

```
            pluginP->message(buff);

            vector<eslapi::CASITransactionIF*> slaves =
                master_port->getSlaves();
            for( int i = 0; i < slaves.size(); ++i ) {
                eslapi::CASITransactionIF *slave = slaves[i];
                string portSlaveName = slave->getPortInstanceName();
                CASIModuleIF *comp = slave->getCASIOwner();
                if(comp != NULL){
                    string compSlaveName =
                        comp->getInstanceName();
                    if((!compSlaveName.empty()) &&
                            (!portSlaveName.empty())){
                                sprintf(buff, "%s     %s %s", tabs,
                                compSlaveName.c_str(),
                            portSlaveName.c_str());
                    }
                }
                pluginP->message(buff);
            }
            if(slaves.size() == 0){
                sprintf(buff, "%s No slaves", tabs);
                pluginP->message(buff);
            }
        }
    }
    pluginP->message("\n");

    sprintf(buff,"%s Parameters:",tabs);
    pluginP->message(buff);

    //Print out all the parameters of this module
    string key, value;
    eslapi::CASIParameterProperties prop;
      const CASIParameterMapIF *p = module->getParameterList();
    while (  p->getNext( key, value, prop ) ) {
        sprintf(buff,"%s %s --> %s",tabs, key.c_str(), value.c_str());
        pluginP->message(buff);

        //Change the parameters as desired
        …
        //Set the new parameter values in the system
        if(!prop.is_readonly && prop.is_runtime){
            if(value == "0"){
                module->setParameter(key, string("1"));
            }
            else if(value == "1"){
                module->setParameter(key, string("0"));
            }
        }
    }

    //Call recursively the sub-components of this component
    const vector< CASIModuleIF* >& uList = module->getSubcomponentList();
    vector<CASIModuleIF*>::const_iterator uIter;
    for ( uIter = uList.begin(); uIter != uList.end(); uIter++ ) {
        printHierarchyModule(dynamic_cast<CASIModule *>((*uIter)),
            deep + 1);
    }
}
```

6.4 Place the plugin dynamic library into etc/plugins directory

The plugin dynamic library must be placed into the *MAXSIM_HOME*/etc/plugins/Release or *MAXSIM_HOME*/etc/plugins/Debug directory.

If the environment variables have been set, replace *MAXSIM_HOME* with:

- %MAXSIM_HOME% for Windows system.
- $(MAXSIM_HOME) for Linux.

On startup, SoC Designer by default searches all the plugin libraries in this directory and activates the corresponding plugins.

——— **Note** ———

When running the release version of SoC Designer, the release version of the plugin DLL is used. Similarly, for the debug version of SoC Designer, the debug plugin is required.

———————————

6.5 Run the plugin

Once the plugin has been placed into `etc/plugins`, you can activate the plugin by calling the plugin script command from a SoC Designer script.

The following example shows a SoC Designer script (`my_example.mxscr`) calling such a plugin:

SoC Designer script

```
// Call the plugin command
callPlugin("get_hierarchy");
```

To run this script, give the following command at the DOS prompt (for Win32) or in the Linux console (for Linux):

```
sdsim --script my_example.mxscr
```

See the *MxScript v3.3 for Cycle Models Reference Manual* (101109) for details on the scripting commands.