BMC FootPrints Asset Core - Customization

Version 11.5

BUSINESS RUNS ON I.T.
I.T. RUNS ON BMC™

www.bmc.com

Legal Notices

BMC Software, Inc.

2101 CityWest Blvd, Houston TX 77042-2827, USA

713 918 8800

Customer Support: 800 537 1813 (United States and Canada) or contact your local support center

Contents

Launching the Asset Core Console via the Command Line

The Asset Core console may also be launched via the command line. In this case it can be opened directly for a specific device and at a specific functionality, such as remote control for device X.

To open the console on the **Direct Access** node of a device of your network:

1. Open a terminal window.
2. Enter the following command line to open the console on the dashboard:

```
NumaraFootPrintsAssetCore.jar -u [myusername] -p [mypassword] -s
[master:1610] -n [mydevice.mycompany.com]
```

If the device does not exist or you do not have sufficient rights to access it, an error message will appear on the screen and the console will be opened on the dashboard.

If `-u, -p` and `-s` are not supplied, the main login window will be displayed asking for the login and password before opening the console on the requested device/functionality.

A device may exist under several nodes, that is under the **Device Topology** as well as in device groups under the main **Device Groups** node. When using the command line options, the console will be opened on the device node which is first returned by the search mechanism. This will always be a node under the **Device Groups** node. The console will open on the device under the **Device Topology** if the device is not a member of any group. If the search command line option is defined, the console will open on the device shown directly under the **Search** node.

The Asset Core console appears on the screen and opens the hierarchy in the left panel on the **Device Groups** > ***Your Device Group***

Launching the Asset Core Console via Java Web Start

The Java Web Start console may also be launched with command line options. It can be opened directly for a specific device and at a specific functionality, such as remote control for device X.

How to create the Agent Interface Page

Asset Core previews the launch of the Console via JWS with different command line options, however this is not included in the software, you need to create the respective interface yourself. Following you will find an example how to do so:

1. Create a new subdirectory in the *[BMC Installation Directory]* `/master/ui directory`, for example, *jws*.
2. Copy the code example provided at the end of this chapter into an .hchl file, for example, *demo.hchl*.
3. Save the file in the new directory.

The new agent interface page via which the Java Web Start Console can be opened is now ready to be used.

How to launch the JWS Agent Interface

Once the agent interface page is created and stored at the proper location it can be launched via a browser.

1. Open a browser and enter the following address:

   ```
   http://IPAddress:PortNumber/jws/demo.hchl
   ```

   The agent interface page for launching the Console via JWS is displayed in the browser.
2. Enter the required information into the fields.
3. Select the functionality on which the console is to open by clicking the respective button, for example, **Direct Access**.

   If the device does not exist or you do not have sufficient rights to access it, an error message will appear on the screen and the console will be opened on the dashboard.

   A device may exist under several nodes, that is under the **Device Topology** as well as in device groups under the main **Device Groups** node. When using the command line options, the console will be opened on the device node which is first returned by the search mechanism. This will always be a node under the **Device Groups** node. The console will open on the device under the **Device Topology** if the device is not a member of any group. If the search command line option is defined, the console will open on the device shown directly under the **Search** node.

   The Asset Core console appears on the screen and opens the hierarchy in the left panel on the ***Your Device*** > **Direct Access** node either under the **Device Groups** > ***Your Device Group*** or the **Device Topology** node in its hierachical position.

Available Options

The following base options are available and may be used for opening the console via Java Web Start:

Cmd	Description
-u user	The login name of the user trying to log on. This option requires the -p option to follow providing the corresponding password. If this is not the case the login window will open requesting the password. The console will then open according to the parameters provided by the command line.
-p password	The corresponding password. This option must be used together with the -u command.
-s server:port	The master server name and port number to which the console it to connect.
-ssl sslmode	The secure mode with which the connection between the console and the master is to be established.
-n device name	The name of the device on which the console is to open. You can either list this option or the -i option to identify the device.
-i device ID	The identification of the device (the ID it is assigned in the devices table of the database) on which the console is to open. You can either list this option or the -n option to identify the device.
-search	Opens the device node under the search node.
-limited	Opens a mini console.

You can use the following options to open the console on a specific context:

Cmd	Description
-ConfigSummary/-inv	Opens the device node on the inventories subnode.
-op	Opens the device node on the assigned operational rule's subnode.
-rc	Opens the device node on the remote control subnode.
-rcd	Opens a remote control connection with the specified device.
-da	Opens the device node on the Direct Access subnode.
-FileSystems	Opens the device on the File System subnode of the Direct Access.
-Registry	Opens the device on the Registry subnode of the Direct Access.
-Services	Opens the device on the Services subnode of the Direct Access.
-Events	Opens the device on the Windows Events subnode of the Direct Access.
-Processes	Opens the device on the Process Management subnode of the Direct Access.
-Ping	Opens the console on the specified device and sends a ping to it. If the options -limited is used only a popup window will be displayed with the result of the ping operation.

Cmd	Description
-Reboot	Opens the console on the specified device and reboots it. If the options -limited is used only a popup window will be displayed with the result of the reboot operation.
-Shutdown	Opens the console and shuts down the device. If the options -limited is used only a popup window will be displayed with the result of the shutdown operation.
-Wakeup	Opens the console on the specified device and tries to wake it up. If the options -limited is used only a popup window will be displayed with the result of the wakeup operation.
-FileTransfer	Opens the device node with the File Transfer window already opened.

Code Example for a JWS Agent Interface page

The following code example launches a web page in which you need to enter the required logon information, such as login name, password, master and port, etc. The line below will provide a row of button via which the specific functionalities are accessed for the required device.

```
<INCLUDE htmlfile="../common/scripts/defs.hchl" onceonly>

<SCRIPT>
    // Get keywords translations

    TranslationInfo TranslationList[]
    string szKeywords[]
    string szLine
    int i, iSize

    szKeywords <<= "_TITLE_CONSOLE_"
    szKeywords <<= "_NOTE_CONSOLE_"
    szKeywords <<= "_CONSOLE_DESC_"
    szKeywords <<= "_JRE_LINKNOTE_"
    szKeywords <<= "_JRE_LINKOTHEROS_"
    szKeywords <<= "_CONSOLE_ONECLICKINSTALL_"

    TranslationList = Translation (szKeywords)
    iSize = ArrayGetSize (TranslationList)

    szLine = "<SCRIPT language='Javascript'>" + ENDLINE
    szLine += "a = new Array (" + iSize + ");" + ENDLINE

    for (i = 0; i < iSize; i += 1)
        szLine += "a[" + i + "]=new Array (2);" + ENDLINE
        szLine += "a[" + i + "][0]=\"" + TranslationList[i+1].szKeyword + "\";" + ENDLINE
        szLine += "a[" + i + "][1]=\"" + TranslationList[i+1].szTranslation + "\";" + ENDLINE
    endfor

    szLine += "</"+"SCRIPT>" + ENDLINE

    Print (szLine)
</SCRIPT>

<SCRIPT language='Javascript'>
    function Popup (link)
    {
        window.open (link, "NumaraFootPrintsAssetCore", "height=400, width=650, toolbar=no,
 menubar=no,
     scrollbars=no, resizable=no, location=no, directories=no, status=no");
    }
</SCRIPT>
<FORM name='console'>
<INPUT type='hidden' name=URL>
```

```
<TABLE>

   <TR>
       <TD class='COLUMN_TEXT1'><SCRIPT language='Javascript'>document.write ('Login');
  </SCRIPT>:  </TD>
       <TD class='COLUMN_TEXT1'><INPUT type='text' name='login' size='20'>
       </TD>
    </TR>
   <TR>
       <TD class='COLUMN_TEXT1'><SCRIPT language='Javascript'>document.write ('Password');
  </SCRIPT>:  </TD>
       <TD class='COLUMN_TEXT1'><INPUT type='text' name='password' size='20'>
       </TD>
    </TR>
    <TR>
       <TD class='COLUMN_TEXT1'><SCRIPT language='Javascript'>document.write ('Server:Port');
  </SCRIPT>:  </TD>
       <TD class='COLUMN_TEXT1'><INPUT type='text' name='serverport' size='20'>
       </TD>
    </TR>
     <TR>
       <TD class='COLUMN_TEXT1'><SCRIPT language='Javascript'>document.write ('SSL');
  </SCRIPT>:  </TD>
       <TD class='COLUMN_TEXT1'><INPUT type='text' name='ssl' size='20' value='0'>
       </TD>
    </TR>
   <TR>
       <TD class='COLUMN_TEXT1'><SCRIPT language='Javascript'>document.write ('DeviceName');
  </SCRIPT>:  </TD>
       <TD class='COLUMN_TEXT1'><INPUT type='text' name='devicename' size='20'>
       </TD>
   </TR>
   <TR>
       <TD class='COLUMN_TEXT1'><SCRIPT language='Javascript'>document.write ('MiniConsole');
  </SCRIPT>:  </TD>
       <TD><INPUT type='checkbox' name='checkmini' size='20'></TD>
   </TR>
   <TR>
       <TD class='COLUMN_TEXT1'><SCRIPT language='Javascript'>document.write ('Choose one
  context:');</SCRIPT>:  </TD>
    </TR>
</TABLE>
</FORM>
<SCRIPT LANGUAGE='Javascript'>
   function RefreshURL ()
   {
       if (console.checkmini.checked)
           console.URL.value="jwsconsole.hchl?u=" + console.login.value +"&p=" +
   console.password.value + "&n=" + console.devicename.value + "&s=" +
   console.serverport.value + "&ssl=" + console.ssl.value +"&Limited"
       else
           console.URL.value="jwsconsole.hchl?u=" + console.login.value +"&p=" +
   console.password.value + "&n=" + console.devicename.value + "&s=" +
   console.serverport.value + "&ssl=" + console.ssl.value
   }
</SCRIPT>

    <INPUT type='button' Value='Operational Rules' OnClick='RefreshURL ();Popup
  (console.URL.value +"&context=op");'>
    <INPUT type='button' Value='Inventory' OnClick='RefreshURL ();Popup (console.URL.value +
  "&context=inv");'>
    <INPUT type='button' Value='Ping' OnClick='RefreshURL ();Popup (console.URL.value +
  "&context=Ping&Limited");'>
    <INPUT type='button' Value='Remote Control' OnClick='RefreshURL ();Popup (console.URL.value +
  "&context=rc");'>
    <INPUT type='button' Value='Registry Limited' OnClick='RefreshURL ();Popup (console.URL.value
 +
  "&context=Registry&Limited");'>
    <INPUT type='button' Value='Processes' OnClick='RefreshURL ();Popup (console.URL.value +
  "&context=Processes");'>
```

```
  <INPUT type='button' Value='Events' OnClick='RefreshURL ();Popup (console.URL.value +
"&context=Events");'>
  <INPUT type='button' Value='Services' OnClick='RefreshURL ();Popup (console.URL.value +
"&context=Services");'>
  <INPUT type='button' Value='Direct Access' OnClick='RefreshURL ();Popup (console.URL.value +
"&context=da");'>
  <INPUT type='button' Value='File Transfer' OnClick='RefreshURL ();Popup (console.URL.value +
"&context=FileTransfer");'>
```

Launching Operational Rules/Software Deployments through XML

The entire process of creating, distributing and executing operational rules/software deployments can be accomplished within the Console. Alternatively you can include a XML file which contains information about:

- assigned operational rules/packages
- assigned devices
- administrator with which assignment is created
- schedule

With this method you can modify operational rules/software deployments by editing the XML file without needing to have access rights to the Console.

1. Create an XML file with the following format:
   - for operational rules:

```
<?xml version="1.0" encoding="UTF-8"?>
<RULEASSOCIATIONS>
    <!-- This section must contain the list of operational rules to assign -->
    <RULES>
        <RULE id="1001"/>             // The Rule ID/Name must be the exact value that
        <RULE name="Test Rule"/>         the rule is known in the Console.
    </RULES>
    <DEVICES>
        <!-- This section must contain the list of devices to which the rules are to
    be assigned -->
        <DEVICE id="1000"/>             // The Device ID/name must be the exact value
        <DEVICE name="Device X"/>          that the device is known in the Console.
    </DEVICES>
    <OPTIONS>
        <!-- When will the rule be activated (number of hours, minutes, etc and 0 for
    immediate) -->
        <ADMINISTRATOR name="admin"/>
        <SCHEDULE hour="16" minute="0" day="9" month="10" year="2011"/>
    </OPTIONS>
</RULEASSOCIATIONS>
```

   - for software deployments:

```
<?xml version="1.0" encoding="UTF-8"?>
<PACKAGEASSOCIATIONS>
    <!-- This section must contain the list of packages to assign -->
    <PACKAGES>
        <PACKAGE id="1001"/>          // The Package ID must be the value that the
 package
    </PACKAGES>       is known in the Console.
    <DEVICES>
        <!-- This section must contain the list of devices to which the packages are
    assigned to-->
        <DEVICE id="1000"/>           // The Device ID must be the value that the
 device is
        <DEVICE id="1002"/>               known in the Console.
    </DEVICES>
    <OPTIONS>
        <!-- This section contains the optional parameters, such as the
 administratorunder
```

```
    which the packages are assigned and the schedule. -->

        <ADMINISTRATOR name="admin"/>
        <SCHEDULE hour="16" minute="0" day="9" month="10" year="2011"/>
    </OPTIONS>
</PACKAGEASSOCIATIONS>
```

2. Add the following step to a new or existing operational rule:
   - For operational rules: **Master Steps** > **Operational Rule Assignment via XML File**
   - For software deployments: **Master Steps** > **Package Assignment via XML File**
3. In the **Properties** dialog enter the complete path to the storage location of the XML file, as well the administrator name in the respective text boxes. This is a default administrator that will only be used if no administrator is defined in the XML file.

You created an Operational Rule which launches Operational Rules/Software Deployments through an XML file. If you want to change schedule or modify included Operational Rules/Packages/Adminstrator you can do so directly in the XML file without having to launch the Console.

Launching Patch Deployments/Assigning Monitoring Policies through XML

The entire process of creating, distributing and executing operational rules/software deployments can be accomplished within the Console. Alternatively you can include a XML file which contains information about:

- assigned patch groups
- assigned devices
- administrator with which assignment is created
- schedule

With this method you can modify patch deployments by editing the XML file without needing to have access rights to the Console.

1. Create an XML file with the following format:

```
<?xml version="1.0" encoding="UTF-8"?>
<OBJECTASSOCIATIONS>
    <!-- This section must contain the list of patch groups to assign -->
    <OBJECTS>
        <!-- type can be OperationalRule, Package, PatchGroup or ApplicationList -->
        <!-- the object can be referenced with its database ID with attribute "id" -->
        <!-- or through its name with attribute "name" -->

        <OBJECT type="PatchGroup" name="MS Office 2007 Patches"/>
    </OBJECTS>
    <DEVICES>
        <!-- This section contains the list of devices to which the objects are to be
    assigned -->
        <!-- Devices can be referenced with database ID (attribute "id") -->
        <!-- or name (attribute "name") -->

        <DEVICE id="1000"/>
        <DEVICE name="Device X"/>
    </DEVICES>
    <DEVICEGROUPS>
        <!-- This section contains the list of device groups to which the objects are to
 be
    assigned -->
        <!-- Groups can be referenced with database ID (attribute "id") -->
        <!-- or name (attribute "name") -->

        <DEVICEGROUP id="1000"/>
        <DEVICEGROUP name="My Group"/>
    </DEVICEGROUPS>
    <OPTIONS>
        <!-- Which administrator profile will be used for the assignment -->
        <ADMINISTRATOR name="admin"/>
        <!-- When will the object assignment be sent to devices (number of hours, minutes,
    etc and 0 for immediate) -->
        <!-- This only applies to OperationalRule, Package, PatchGroup object types -->

        <SCHEDULE hour="16" minute="0" day="9" month="10" year="2011"/>
    </OPTIONS>
</OBJECTASSOCIATIONS>
```

You can mix object types in this file, that is, you can list patch groups with packages in this file, if they are all to be assigned to the same target devices and/or groups.

> You can list device and/or device groups in this file, depending on the targets of the specified objects. If, for example, the objects are only to be assigned to device groups, the <DEVICES> section is not needed.

2. Add the following step to a new or existing operational rule: **Master Steps**->**Assignment Management via XML File**.
3. In the **Properties** dialog enter the complete path to the storage location of the XML file as well the administrator name in the respective text boxes. This administrator is a default administrator that will only be used if no administrator is defined in the XML file.
4. Enter the directories into which the xml files are to be copied in case of success or error.

> If an error occurred during the assignment process you can find explanations in the *OperationalRules.log* file.

5. If you want the assignments directly activated, that is, to become operational right away, check the **Activate Created Assignment** box. If this box is left unchecked the assignments will remain paused.
6. If the newly defined assignment is to overrule any possibly already existing assignment, that is, if the object is to be reassigned with the new information, check the **Reassign if Assignment Already Exists** box. If you do not check this box and such an object assignment already exists, the original assignment will still be valid.

You created an object assignment which launches a patch deployment through an XML file. If you want to change the schedule or modify the included patch groups/application policies/adminstrator, you can do so directly in the XML file without having to launch the Console.

Unassigning Patch Deployments

This step also allows you to unassign patch deployment assignments. To execute such an operation the same information is required as for the assignment process. The only difference is in the contents of the XML file: the opening and closing tag of the file use the expression <OBJECTUNASSIGN> instead of <OBJECTASSOCIATIONS>.

Example:

```
  <![CDATA[
<?xml version="1.0" encoding="UTF-8"?>
<OBJECTUNASSIGN>
 <OBJECTS>
  <OBJECT type="PatchGroup" name="PG Test Rule"/>
 </OBJECTS>
 <DEVICEGROUPS>
  <DEVICEGROUP id="1000"/>
  <DEVICEGROUP name="My Group"/>
 </DEVICEGROUPS>
</OBJECTUNASSIGN>
```

Assigning Application Management Policies through XML

The entire process of creating, distributing and assigning Application Management Policies can be accomplished within the Console. Alternatively you can include an XML file which contains information about:

- assigned application management policies
- assigned devices
- administrator with which assignment is created
- schedule

With this method you can modify Application Management Policies by editing the XML file without needing to have access rights to the Console.

1. Create an XML file with the following format:

```
<?xml version="1.0" encoding="UTF-8"?>
<OBJECTASSOCIATIONS><!-- This section must contain the list of application rules
 to assign -->
 <OBJECTS>
  <!-- type can be OperationalRule, Package, PatchGroup or ApplicationList -->
  <!-- the object can be referenced with its database ID with attribute "id" -->
  <!-- or through its name with attribute "name" -->
  <OBJECT type="ApplicationList" id="1001"/>
 </OBJECTS>
 <DEVICES>
  <!-- This section contains the list of devices to which the objects are to be
 assigned -->
  <!-- Devices can be referenced with database ID (attribute "id") -->
  <!-- or name (attribute "name") -->

  <DEVICE id="1000"/>
  <DEVICE name="Device X"/>
 </DEVICES>
 <DEVICEGROUPS>
  <!-- This section contains the list of device groups to which the objects are
 to be assigned -->
  <!-- Groups can be referenced with database ID (attribute "id") -->
  <!-- or name (attribute "name") -->

  <DEVICEGROUP id="1000"/>
  <DEVICEGROUP name="My Group"/>
  </DEVICEGROUPS>
 <OPTIONS>
  <!-- Which administrator profile will be used for the assignment -->

  <ADMINISTRATOR name="admin"/>

  <!-- When will the object assignment be sent to devices (number of hours,
 minutes, etc and 0 for immediate) -->
  <!-- This only applies to OperationalRule, Package, PatchGroup object types -->

  <SCHEDULE hour="16" minute="0" day="9" month="10" year="2011"/>

  <!-- Add if type is OperationalRule and only advertizemnt is needed -->

  <ADVERTIZE/>
 </OPTIONS>
```

```
</OBJECTASSOCIATIONS>
```

You can mix object types in this file (for example,include application lists, devices and/or groups.

2. Add the following step to a new or existing operational rule: **Master Steps** > **Assignment Management via XML File** .
3. In the **Properties** dialog enter the complete path to the storage location of the XML file as well the administrator name in the respective text boxes. This administrator is a default administrator that will only be used if no administrator is defined in the XML file.
4. Enter the directories into which the xml files are to be copied in case of success or error.

   If an error occurred during the assignment process you can find a explanation on what happened in the *OperationalRules.log* file

   You can list device and/or device groups in this file, depending on the targets of the specified objects. If, for example, the objects are only to be assigned to device groups, the <DEVICES> section is not needed.

5. If you want the assignments directly activated, that is, to become operative right away, check the **Activate Created Assignment** box. If this box is left unchecked the assignments will remain paused.
6. If the newly defined assignment is to overrule any possibly already existing assignment, that is, if the object is to be reassigned with the new information, check the **Reassign if Assignment Already Exists** box. If you do not check this box and such an object assignment already exists, the original assignment will still be valid.

You created an object assignment which launches the management of application lists through an XML file. If you want to change the schedule or modify the included patch groups/application policies/adminstrator you can do so directly in the XML file without having to launch the Console.

Unassigning Application Lists

This step also allows you to unassign application list assignments. To execute such an operation the same information is required as for the assignment process. The only difference lies in the contents of the XML file: the opening and closing tag of the file use the expression <OBJECTUNASSIGN> instead of <OBJECTASSOCIATIONS>.

Example:

```
  <![CDATA[
<?xml version="1.0" encoding="UTF-8"?>
<OBJECTUNASSIGN>
 <OBJECTS>
  <OBJECT type="ApplicationList" name="Prohibiting Pinball"/>
 </OBJECTS>
 <DEVICEGROUPS>
  <DEVICEGROUP id="1000"/>
  <DEVICEGROUP name="My Group"/>
 </DEVICEGROUPS>
</OBJECTUNASSIGN>
```

Integrating with the Asset Core Database

Third party applications can gather data directly from the Asset Core database.

The Entity Relationship Diagrams can be found on the BMC *support site.*

These include a description of each table and lists of columns grouped in subject areas.

It is not recommended to write data to the Asset Core tables, but only perform read-only access.

Adding Custom Operational Rule Steps

Asset Core provides a large number of predefined steps for **Operational Rules**. However, you can create customized operational rule steps.

Introduction to Operational Rule Steps

Asset Core software includes a large number of predefined operational rule steps which are located in the `data/Vision64Database/opsteps` directory. In this directory Asset Core expects to find pairs of files: an `.xml` file describing an individual operational step and a `.chl` file which is the script to execute for the step.

Upon finding such a pair, the .xml file is parsed to see if a new step can be imported into the database. If the .xml file is wrong, both files remain where they are and nothing happens. If it is correct, the script it points to is compiled using the local **Operational Rules** module. If the compile passes, the step is added to the database and the two files are placed in a specified directory structure under `opsteps`. This allows a history to be kept of all versions of this specific step. If the compile fails, the files are moved to a directory called `opsteps.invalid`. The reason for the failure is described in the `chilli.log` file. For more information, refer to the debug chapter of the **Chilli Reference**.

A step always needs two files:

- a `<StepName>.xml` file describing an individual operational step and
- a `<StepName>.chl` file which is the script to execute for the step

Custom operational rule steps may be added to Asset Core by simply creating the necessary scripts and adding them to the respective directory. The operational rules steps can easily be localized for different languages if necessary. The following paragraphs explain the contents of both the .xml and .chl file, and how to create your own steps. In the following chapter you will find some examples of custom operational rule steps for specific needs and situations.

Importing Newly Created Steps

Steps may be added to Asset Core at any time. Once their scripts are created and copied in the proper location on the master, they may be directly imported in the Console. To import steps:

1. Put the files (.xml and .chl) for the new steps in the directory data/Vision64Database/opsteps.
2. Then select the**Tools**->**Import New Steps** menu item.
3. Specify in the **Schedule Import of New Steps** window if the import is to be launched immediately or if it is to be fixed for a specific date and time by entering the values in the respective fields.
4. Click **OK** to confirm the schedule and close the window.

Any new steps that are located in the source directory will be imported into the Asset Core database at the specified time.

XML File of a Step

The .xml file is the "link" between the actual action which will be executed by the Chilli script and the operational rule step action's definition in the Console. It contains a general description the step, calls the Chilli script, and provides the parameters to the Console that need to be defined and then passed on to the script for execution. These are the parameters displayed in the **Properties** window of the **Select a Step** dialog in which you enter the values.

The following general rules apply for the xml tags:

- None of the xml tags is allowed parameters.
- All tags must have an opening and an end tag.

Which elements can I use?

For the XML file you can use the following elements. Their relation is indicated in the table by indendation, for example, PARAM is a child of PARAMS and a parent of LABEL.

Note

Be aware that the xml file must always start with the processing instruction: `<?xml version="1.0"?>`

Name	Description	Required
STEPTYPE	The root element. It has no parameters. The only allowed tags within this element are CLASS, NAME, SCRIPT, NOTE and PARAMS.	Yes
CLASS	The name of the step class to which this step belongs. A step class is a container for a number of steps concerning a specific topic, such as Agent Configuration, Directory and File Handling or User Message Box. You can enter either a hardcoded text such as Tools or a keyword, such as _DB_STEPCLASS_TOOLS_.	Yes
NAME	The name of the step. You can enter either a hardcoded text such as Add Line or a keyword, such as _DB_STEPNAME_ADDLINE_.	Yes
SCRIPT	The name of the Chilli script which corresponds to the .xml file, in this example `addline.chl`.	Yes
NOTE	Additional information suchas a short description of the step. You can enter either a hardcoded text such as Add a line to a file or a keyword, such as _DB_STEPNOTE_ADDLINE_.	No
PARAMS	Container for the parameters. If the step doesn't have any parameters, leave this element empty.	Yes
PARAM	Container for a single parameter.	No
NAME	The name of the Chilli variable as listed in the CHL script.	If child of PARAM
LABEL	The label of the above defined Chilli variable. This label is displayed in the **Properties** dialog when adding the step. You can enter either a hardcoded text or a keyword.	If child of PARAM

Name	Description	Required
TYPE	The type of the parameter. Possible values are: • `String`: parameter will be represented as a text field where you can enter a sequence of characters • `Integer`: parameter will be represented as a text field where you can only enter integers • `Text`: parameter will be represented as a multiline text field • `Boolean`: parameter will be represented as a check box • `Enum`: parameter will be represented as a drop-down list. In this case ENUMGROUP, ENUMTYPE and ENUMVALUE must also be specified. • `ObjectType`: parameter will be represented as a text field in which objects of a specific type may be added. In this case the OBJECTTYPE, OBJECTTYPEMEMBER, OBJECTTYPEGROUP and OBJECTTYPENBOFITEMS must also be specified.	If child of PARAM
DEFAULT	The default value if one is proposed for the parameter. If no default is proposed, leave this element empty.	If child of PARAM
ENUMGROUP	The name of the enumeration under which it will be stored in the database (for example, *FirewallProfiles*).	If TYPE has the value ENUM
ENUMTYPE	The type of the items of the drop-down list. Possible values are: • `String`: items are a sequence of characters • `Integer`: items are integers • `Boolean`: two items Yes and No	If TYPE has the value ENUM
ENUMVALUE	The values of the items of the drop-down list. All values need to be separated by a comma, for example, *ZIP, PKG*. You can enter either a hardcoded text or a keyword.	If TYPE has the value ENUM
OBJECTTYPE	The type of the object which may be selected (that is, *_DB_OBJECTTYPE_DEVICE_* if devices and device groups may be selected).	If TYPE has the value ObjectType
OBJECTTYPE-MEMBER	Defines if the **Select Objects** dialog displays the individual objects of the specified object type (for example, devices or operational rules). Possible values are: • `1`: objects are displayed • `0`: objects are not displayed If both OBJECTTYPEMEMBER and OBJECTTYPEGROUP are set to `0` this option will automatically be considered as set to `1`.	If TYPE has the value ObjectType

Name	Description	Required
OBJECTTYPE-GROUP	Defines if the **Select Objects** dialog also displays the groups or folders of the selected object type (for example, device groups or operational rule folders). Possible values are: • 1: objects are displayed • 0: objects are not displayed	If TYPE has the value ObjectType
OBJECTTYPE-NBOFITEMS	Defines the number of objects that can be added to the list. If set to 0, an unlimited number of objects can be added.	If TYPE has the value ObjectType
OPTIONAL	Defines that the parameter is optional. In this case a check box is added before the actual parameter field which must be checked to activate the actual parameter	No
DEFAULT-PRESENCE	Defines if the OPTIONAL box before the parameter field is checked by default, that is, if the parameter is activated by default.	If OPTIONAL is present

Examples

The following paragraphs provide you with a number of examples for the above explained tags. The examples are shown in two versions, once with keywords to be localized and once with hardcoded text if no localization is required. For information on how to localize your steps see chapter Localization of this manual.

Example 1

The following is an excerpt of a script collecting values from an ini file and placing these in the Custom Inventory, this parameter defines the path to the configuration file from which the value is to be recovered, using keywords:

```
<PARAM>
 <NAME>IniFilePath</NAME>
 <LABEL>_DB_STEPPARAM_INIFILEPATH_</LABEL>
 <TYPE>String</TYPE>
 <DEFAULT></DEFAULT>
</PARAM>
```

whereby `IniFilePath` is internal Chilli variable name for the `_DB_STEPPARAM_INIFILEPATH_` step label, the variable is of type string and does not have any preentered default value.

Following you will find the same example using hard coded text instead of keywords:

```
<PARAM>
 <NAME>IniFilePath</NAME>
 <LABEL>File Path</LABEL>
 <TYPE>String</TYPE>
 <DEFAULT></DEFAULT>
</PARAM>
```

Example 2

The following parameter tag defines a drop-down list field from which a choice may be made:

```
<PARAM>
 <NAME>Protocol</NAME>
 <LABEL>_DB_STEPPARAM_PROTOCOL_</LABEL>
 <TYPE>Enum</TYPE>
 <ENUMGROUP>FirewallProtocol</ENUMGROUP>
 <ENUMTYPE>String</ENUMTYPE>
 <ENUMVALUE>TCP, UDP</ENUMVALUE>
 <DEFAULT>TCP</DEFAULT>
</PARAM>
```

whereby `Protocol` is internal Chilli variable name for the `_DB_STEPPARAM_PROTOCOL_` step label, the ENUMGROUP defines that the parameter is to be found in the database in the *FirewallProtocol* column, the enumeration is of type `String`, its values are `TCP` and `UDP` and the preentered default value is the `TCP` protocol.Following you will find the same example using hard coded text instead of keywords:

```
<PARAM>
 <NAME>Protocol</NAME>
 <LABEL>Protocol</LABEL>
 <TYPE>Enum</TYPE>
 <ENUMGROUP>FirewallProtocol</ENUMGROUP>
 <ENUMTYPE>String</ENUMTYPE>
 <ENUMVALUE>TCP, UDP</ENUMVALUE>
 <DEFAULT>TCP</DEFAULT>
</PARAM>
```

Example 3

The following set of tags defines a list box in which a number of devices and device groups may be selected:

```
<PARAM>
 <NAME>Objects</NAME>
 <LABEL>_DB_STEPPARAM_OBJECTOFTYPEDEVICEORDEVICEGROUP_</LABEL>
 <TYPE>ObjectType</TYPE>
 <OBJECTTYPE>_DB_OBJECTTYPE_DEVICE_</OBJECTTYPE>
 <OBJECTTYPEMEMBER>1</OBJECTTYPEMEMBER>
 <OBJECTTYPEGROUP>1</OBJECTTYPEGROUP>
 <OBJECTTYPENBOFITEMS>0</OBJECTTYPENBOFITEMS>
 <DEFAULT/>
</PARAM>
```

whereby `Objects` is internal Chilli variable name for the `_DB_OBJECTTYPE_DEVICE_` step label, the list has no default object entered, individual objects, that is, *devices*, as well as group objects, that is, *device groups*, may be selected and the list is unlimited.

Following you will find the same example using hard coded text instead of keywords:

```
<PARAM>
 <NAME>Objects</NAME>
 <LABEL>Devices and/or Device Groups</LABEL>
 <TYPE>ObjectType</TYPE>
 <OBJECTTYPE>_DB_OBJECTTYPE_DEVICE_</OBJECTTYPE>
 <OBJECTTYPEMEMBER>1</OBJECTTYPEMEMBER>
 <OBJECTTYPEGROUP>1</OBJECTTYPEGROUP>
 <OBJECTTYPENBOFITEMS>0</OBJECTTYPENBOFITEMS>
 <DEFAULT/>
</PARAM>
```

Example of a complete XML File

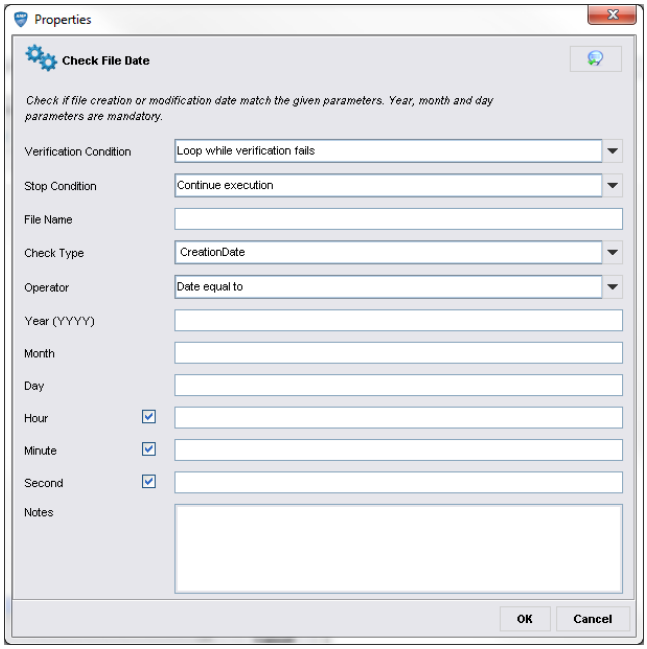
The following example shows the code of an XML file as well as what the step looks like in the Console.

This is the code of the `checkfiledate.xml` file of the predefined step **Check File Date**:

```
<?xml version="1.0"?>
<STEPTYPE>
    <CLASS>_DB_STEPCLASS_MONITORING_</CLASS>
    <NAME>_DB_STEPNAME_CHECKFILEDATE_</NAME>
    <SCRIPT>checkfiledate.chl</SCRIPT>
    <NOTE>_DB_STEPNOTE_CHECKFILEDATE_</NOTE>
    <PARAMS>
        <PARAM>
            <NAME>FileName</NAME>
            <LABEL>_DB_STEPPARAM_FILENAME_</LABEL>
            <TYPE>String</TYPE>
            <DEFAULT></DEFAULT>
        </PARAM>
        <PARAM>
            <NAME>CheckType</NAME>
            <LABEL>_DB_STEPPARAM_CHECKTYPE_</LABEL>
            <TYPE>Enum</TYPE>
            <ENUMGROUP>CheckType</ENUMGROUP>
            <ENUMTYPE>String</ENUMTYPE>
            <ENUMVALUE>ModificationDate, CreationDate</ENUMVALUE>
            <DEFAULT>CreationDate</DEFAULT>
        </PARAM>
        <PARAM>
            <NAME>CheckDateRange</NAME>
            <LABEL>_DB_STEPPARAM_CHECKDATERANGE_</LABEL>
            <TYPE>Enum</TYPE>
            <ENUMGROUP>CheckDateRange</ENUMGROUP>
            <ENUMTYPE>String</ENUMTYPE>
            <ENUMVALUE>_DB_STEPPARAM_DATELESSTHAN_,_DB_STEPPARAM_DATEGREATERTHAN_,
   _DB_STEPPARAM_DATEEQUALORGREATERTHAN_,_DB_STEPPARAM_DATEEQUALORLESSTHAN_,_
   DB_STEPPARAM_DATEEQUAL_,_DB_STEPPARAM_DATENOTEQUAL_</ENUMVALUE>
            <DEFAULT>_DB_STEPPARAM_DATEEQUAL_</DEFAULT>
        </PARAM>
        <PARAM>
            <NAME>Year</NAME>
            <LABEL>_DB_STEPPARAM_YEAR_</LABEL>
            <TYPE>Integer</TYPE>
            <DEFAULT></DEFAULT>
        </PARAM>
        <PARAM>
            <NAME>Month</NAME>
            <LABEL>_DB_STEPPARAM_MONTH_</LABEL>
            <TYPE>Integer</TYPE>
            <DEFAULT></DEFAULT>
        </PARAM>
        <PARAM>
            <NAME>Day</NAME>
            <LABEL>_DB_STEPPARAM_DAY_</LABEL>
            <TYPE>Integer</TYPE>
            <DEFAULT></DEFAULT>
        </PARAM>
        <PARAM>
            <NAME>Hour</NAME>
            <LABEL>_DB_STEPPARAM_HOUR_</LABEL>
            <TYPE>Integer</TYPE>
            <OPTIONAL>true</OPTIONAL>
            <DEFAULTPRESENCE>false</DEFAULTPRESENCE>
            <DEFAULT></DEFAULT>
        </PARAM>
        <PARAM>
```

```
            <NAME>Minute</NAME>
            <LABEL>_DB_STEPPARAM_MINUTE_</LABEL>
            <TYPE>Integer</TYPE>
            <OPTIONAL>true</OPTIONAL>
            <DEFAULTPRESENCE>false</DEFAULTPRESENCE>
            <DEFAULT></DEFAULT>
        </PARAM>
        <PARAM>
            <NAME>Second</NAME>
            <LABEL>_DB_STEPPARAM_SECOND_</LABEL>
            <TYPE>Integer</TYPE>
            <OPTIONAL>true</OPTIONAL>
            <DEFAULTPRESENCE>false</DEFAULTPRESENCE>
            <DEFAULT></DEFAULT>
        </PARAM>
    </PARAMS>
</STEPTYPE>
```

This is what the step looks like in the Console:



Notice that all keywords in the code have been replaced by their english translations. **Verification Condition**, **Stop Condition** and **Notes**. Parameters with different values for the TYPE elements are displayed in the following way:

- `String`: a free text box
- `Boolean`: a check box
- `Integer`: a text box in which only numbers can be entered
- `Enum`: a dropdown list with several options defined via the ENUMVALUE tag.
- `Optional`: a check box before the actual data field, the DEFAULTPRESENCE parameter defines if it is checked by default.

CHL File of a Step

What is a CHL file?

A CHL file is script that executes the desired action(s). It is written in Chilli, the BMC Software proprietary programming language.

What is the structure of a CHL file?

A CHL file has the following structure:

- Description
- Global variables:
- Local variables:
- Custom defined procedures
- Main procedure

Description

Contains information about the script, version, creation date, programmer, etc.

```
########################################################################
# CustomPackagerModuleSetup
# Modify the Custom packager module parameters
#
########################################################################
```

Global variables

Similar to other programming languages Chilli may use global variables. You may call any number and type of external variables, however be aware that the following two are mandatory for all step scripts:

- StepParamsContainer: contains the values of all parameters defined by the step
- RuleParamsContainer: contains general information about the **Operational Rule** the step is added to. It may be used by the steps to communicate with each other if an **Operational Rule** contains more than one step.

```
#### The Rule container can be used by steps to communicate with each other.

extern ContainerHandle StepParamsContainer, RuleParamsContainer
```

Local variables

The value of the NAME elements of parameters defined in the XML file need to be defined as local variables in the CHL file.

Following you find the example of the predefined step **Custom Package Module Setup** which demonstrates how the parameters in the XML and CHL files are linked:

```
<PARAMS>
     <PARAM>
         <NAME>PackageExtension</NAME>
         <LABEL>_DB_STEPPARAM_PACKAGEEXTENSION_</LABEL>
         <TYPE>String</TYPE>
         <DEFAULT>.zip</DEFAULT>
     </PARAM>
     <PARAM>
         <NAME>MaxRetry</NAME>
         <LABEL>_DB_STEPPARAM_MAXRETRY_</LABEL>
         <TYPE>Integer</TYPE>
         <DEFAULT>5</DEFAULT>
     </PARAM>
```

```
        <PARAM>
            <NAME>RetryInterval</NAME>
            <LABEL>_DB_STEPPARAM_RETRYINTERVAL_</LABEL>
            <TYPE>Integer</TYPE>
            <DEFAULT>300</DEFAULT>
        </PARAM>
    </PARAMS>
```

`custompackagermodulesetup.xml`

```
#### Our parameters to be found in StepParamsContainer:
const PACKAGERCUSTOM_PARAM_PACKAGEEXTENSION   "PackageExtension"
const PACKAGERCUSTOM_PARAM_MAXRETRY           "MaxRetry"
const PACKAGERCUSTOM_PARAM_RETRYINTERVAL      "RetryInterval"
const ERROR_CODE                              "ErrorCode"

const PARAM_NAME                              "Name"
const PARAM_PERSISTENT                        "Persistent"
const PARAM_STATE                             "State"
const PARAM_STATE_INITIALISE                  1
const ACTIONDB_ERROR_DUPLICATE                11
```

`custompackagermodulesetup.chl`

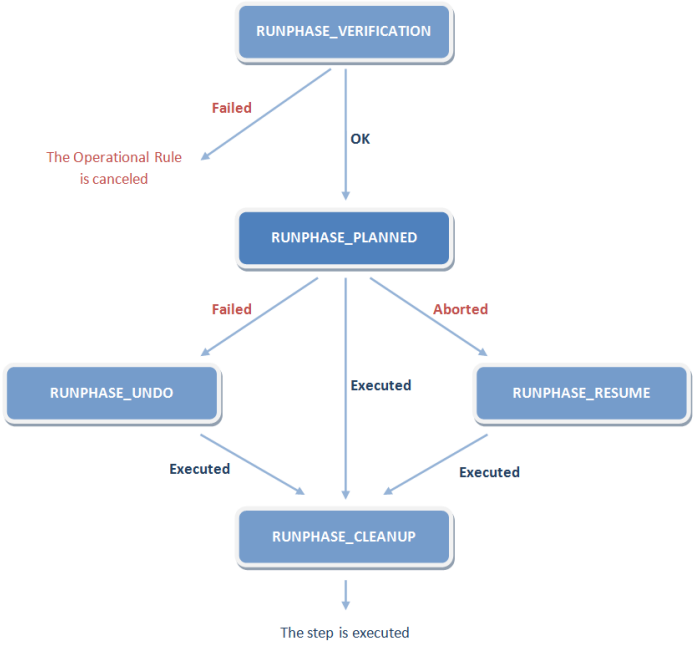
Notice how the three NAME elements PackageExtension, MaxRetry and RetryInterval are defined in the CHL file. Additionally you find the five local variables ERROR_CODE, PARAM_NAME, PARAM_PERSISTENT, PARAM_STATE, PARAM_STATE_INITIALISE and ACTIONDB_ERROR_DUPLICATE which are used later on in the Main procedure.

Custom defined procedures

code that is defined once in the script and may be executed as often as needed by other parts of the script or program

Main procedure

The main procedure follows the same basic rules as in other programming languages. However, due to the functioning of the operational rules module the Main procedure must be divided into runphases. There are 5 different runphases, of which the RUNPHASE_PLANNED is mandatory. The operational rules module will not execute step scripts which do not at least contain this runphase. Also all individual operations are required to return either 0 for successful execution or non-zero for failed.The process of executing an operational rule containing three steps for example is as follows:



RUNPHASE_VERIFICATION

The verification phase verifies if all prerequisites for the execution of the steps are given, that is, it verifies for the correct operating system, if the environment variable or the registry key exists, etc. The process verifies the steps in the order in which they are defined. If the verification of all steps returns OK, the script passes on to RUNPHASE_PLANNED. If the verification fails, the **Operational Rule** is canceled.

RUNPHASE_PLANNED

This phase is the actual execution phase of the script. It executes one step after the other in the defined order. This phase can have one of the following results:

- If all steps are successfully executed, the script passes to RUNPHASE_CLEANUP
- If execution failed, the script passes to RUNPHASE_UNDO
- If execution is aborted due to a "non-execution" problem such as the Asset Core Agent crashing or the machine shutting down, the script passes to RUNPHASE_RESUME at the next Asset Core Agent startup

RUNPHASE_UNDO

This phase is used to undo all changes executed via the steps if the execution of the **Operational Rule** fails. After the undo the device will be in exactly the same state and situation as before the execution of the **Operational Rule**.

RUNPHASE_RESUME

If execution is aborted due to a "non-execution" problem such as the Asset Core Agent crashing or the machine shutting down, at the next Asset Core Agent this phase will check the **Operational Rule** and verify which of its steps it finished executing. It will then proceed to re-execute the step at which the execution was interrupted, and run all remaining steps.

RUNPHASE_CLEANUP

This last phase cleans up the device, for example, deleting the MSI file used for a software installation.

How does Chilli work?

Chilli is a procedural programming language, combining the features of BASIC and C as well as some C++ concepts into a flexible computer language. This powerful script language is a self-contained stand-alone language with its own compiler. For in-depth information about Chilli consult the *Chilli Reference*.

Adding a Customized Menu to Devices

When right-clicking an object in the left or right window pane a context menu opens. The available items in the menu depend on the object and the position in the Console. You can expand options for devices by adding a **Customized Menu**.

What is a Customized Menu?

A **Customized Menu** is an additional menu for devices in the Console with which you can quickly execute customized actions.

Are there different types of Customized Menus?

You can create one **Customized Menu** which comprises three different item types:

- **Executable**: launching an executable file
- **HTTP**: opening a web page in your browser
- **Command Line**: executing a command in the command line

How many Customized Menus can I create?

You can create one **Customized Menu** with an arbitrary number of menu items.

Creating a Customized Menu

To create a **Customized Menu** proceed as follows:

1. Go to **Global Settings** > **System Variables** and make sure that the **Device Menu** tab is selected.
2. Click the **Create Device Menu** icon in the icon bar.

   The **Properties** dialog displays.
3. From the **Menu Type** drop-down list select the item type and fill in the two text boxes.

   The following three examples illustrate possible applications:

Example	Name	Value	Menu Type
Example 1	`Open *your web page*`	`http://www.*yourwebpage*.com`	**HTTP**
Example 2	`Open Editor`	`C:\Windows\notepad.exe`	**Executable**
Example 3	`Open Registry Editor`	`regedit.exe`	**Command Line**

4. Click **OK**.

   The dialog closes and the new **Customized Menu** item is listed in the right window pane.

You created a **Customized Menu** with one item. You can add further items by repeating the steps.

Launching a Customized Menu

After creating a new **Customized Menu** you can launch its items. To launch the items:

1. In the left window pane, right-click a device.
2. In the context menu select the **Customized Menus** submenu and click the item you created.

   The defined operation is executed.

You launched a **Customized Menu** item. You can access your **Customized Menu** from any device in the left window pane.

Customizing the Agent Web Interface

The following chapters describe the individual elements of the Agent Web interface. . This section begins with information about the HCHL pages in general and specific information on HCHL for the agent pages in particular:

- *Elements of the* Asset Core *Agent Interface Pages*
- *New and Extended Metrix HTML Tags and Parameters*
- *Chilli in the Metrix Web Console*
- *Tags and Parameters*

The pages for the agent interface are designed exclusively in standard HTML 4.0, the BMC Software extended tags and parameters, and Chilli, a proprietary BMC Software programming language. See the BMC Software Chilli Reference Guide for detailed information. The following instructions assume you are already familiar with standard HTML 4.0. This section explains in detail tags and parameters specifically designed by BMC Software for the optimally run the agent interface.

Elements of the Asset Core Agent Interface Pages

The basis for the web page (HCHL) files of the Asset Core Agent Interface is standard HTML 4.0. However, for optimal execution of the Asset Core Agent Interface, the functionalities for several standard tags were extended (for example, new parameters, tags were created). Another important element in the handling of the HCHL files are headers. Some adjustments were made to the headers for the easy manipulation of the interface. The following paragraphs and chapters describe in detail the use and concepts for the following basic elements of the Asset Core Agent Interface pages:

- **BMC Software Tags**

  The basis of all Asset Core Agent Interface pages is standard HTML 4.0. However, as the Agent Interface has some very specific needs, existing HTML tags were given extended functionalities, such as the capability to handle Chilli expressions., New tags and parameters were created where necessary
- **Chilli in the Asset Core Agent Interface**

  Chilli is used in the Agent Interface to execute repetitive and automated tasks through scripts. These scripts can be called through the new the command line. In addition to the script being called, further parameters can be specified within the tag's arguments. These Chilli functions are explained in more detail in later chapters. References to the *Chilli Reference* guide will be made where further explain a topic or item.

New and Extended Metrix HTML Tags and Parameters

The basis for the HTML files of the FootPrints Asset Core is standard HTML 4.0. However, for Agent Web Interface to run optimally, several standard tags were extended in their functionalities (for example, by adding parameters and new tags). This reference assumes that you are already familiar with standard HTML 4.0.

HTML Tags

BMC Software extended standard tags:	New BMC Software tags:
A (Anchor)	Deftag
Form	IF/ELSE

BMC Software extended standard tags:	New BMC Software tags:
Script	Include
	Loop
	Setvar

The difference between these two groups of tags is that the new BMC Software tags are completely newly created tags, as well as the parameters they use. The Extended Tags are standard HTML 4.0 tags with additional parameters to extend their functionality, which will be explained in detail later in this reference.

In principle, the new and extended tags follow the general rules of standard HTML tags, which have three parts:

- a start tag
- the content
- an end tag

Standard Tag Syntax

```
<TAG parameter1=value1, parameter2=value2, ....>Content</TAG>
```

A tag is special text ("markup") that is delimited by angle brackets ("<" and ">"). An end tag includes a forward slash ("/") after the left angle bracket ("<"). For example, the anchor tag "A" has a start tag, "<A>", and an end tag, "</A>". The start and end tags surround the content of the anchor tag:

```
<A>http://www.metrixsystems.com</A>
```

Some tags may not have an end tag, and this will be explicitly mentioned in the respective section. Tag names are always case-insensitive, so <setvar>, <Setvar>, and <SETVAR> are all the same.

Chilli and HTML Tags

One of the principal functionalities of all BMC Software extended or created tags is the possibility to directly handle Chilli expressions. (Chilli is a programming language created by BMC Software specifically for the purpose of network and systems management. For more information on Chilli, refer to the Chilli Reference guide.)

Metrix Tag Syntax

```
<TAG parm1=(Chilli Expression) parm2=....>Text</TAG>
```

Chilli expressions can be used as values of HTML tag parameters. They are enclosed in parentheses to distinguish them from "normal" parameter values. When the parser finds an opening parenthesis at the beginning of the parameter value, it will evaluate everything in the parentheses as a Chilli expression.

Parameters

To further extend the functionality of standard HTML 4.0 tags, as well as those of the tags newly created by BMC Software, the following parameters were added to the above-mentioned tags:

Parameter Name	Used by Tags
condition	IF/ELSE, INLCUDE, LOOP
htmlfile	A, FORM, INCLUDE
language	SCRIPT

Parameter Name	Used by Tags
name	DEFTAG, SETVAR
onceonly	INLCUDE
parseoutput	DEFTAG, SCRIPT
proc	DEFTAG
value	SETVAR
vars	A, SCRIPT

A tag's parameters define various properties for the tag. For example, the IMG element takes an SRC parameter to provide the location of the image and an ALT parameter to give alternate text for users who disabled image autoloading:

```
<IMG SRC="numarasoftwarelogo.gif" ALT="BMC Software Logo">
```

A parameter is included in the start tag only - never the end tag - and takes the form parameter name="parameter-value". The 'name=value'pairs are separated from each other by a space (and no comma). The parameter value is enclosed by single or double quotes if it consists of more than one word. For example, when listing variables: vars='iid, myvar, secondvar'. If the value is a one-wordexpression, it does not need to be put in quotes. The parser does not distinguish between single and double quotes, so either of them can be used. If quotes are used when not needed, the parser will ignore them. Parameter names are case-insensitive, but parameter values are case-sensitive.

Chilli in the Asset Core Agent Interface

Chilli is a procedural programming language, combining the features of BASIC and C, as well as some C++ concepts into a flexible computer language. This powerful script language is a stand-alone language with its own compiler.

Chilli is used in the Asset Core Agent Web Interface to execute repetitive and automated tasks through scripts integratedinto its HCHL pages. These scripts can be called through the new BMC Software extended tags or via the command line. Additional parameters can be specified with the script to be called within the tag's arguments. How these Chilli functions are called is explained in more detail in the chapters on the respective tags later in this reference.

Below are some examples of a tag including Chilli.

Example 1

```
   <SETVAR name=CountModels value=(ArrayGetSize (NaviLoopModelList.ExpModelDescList))>
```

In this example a Chilli function, ArrayGetSize with its argument, is used as the value for the value parameter of the BMC Software specific SETVAR tag.

Example 2

```
<SCRIPT language=chilli>
 if (defined ('_wpcolor'))
  if(_wpcolor != "")
   UserSettingSetValue (REMOTE_USER, "Prop/WpColor", _wpcolor)
  endif
 endif
</SCRIPT>
```

In this example a Chilli script is executed through the SCRIPT tag, where the language parameter is defined as the Chilli scripting language.

Chilli Functions in the Asset Core Agent Interface

The Chilli language has been extended for the Asset Core Agent Interface with some additional function modules. However, almost all existing modules either are or can be used with it.

Functions of the following general Chilli modules are used by the Asset Core Agent Interface:

Functions

- File
- SNMP
- String
- Miscellaneous
- HTML File
- CSV File
- Gif Image Manipulation
- Variable Manipulation
- DBM Database

As this reference concentrates on the Asset Core Agent Interface specific elements, it will not explain any Chilli functions. For more information on these refer to the BMC Software Chilli Reference guide. This guide includes detailed information about the Chilli language in general, as well as the overall possibilities and functionalities of all Chilli functions.

Tags and Parameters

The following chapters describe in detail all standard HTML 4.0 tags which were extended by BMC Software to allow for additional functionality, as well as all HTML tags specifically created by BMC Software for the purpose of a smooth and uncomplicated execution of the Asset Core Agent Interface. It is a complete reference to all changed and new tags and parameters of theAsset Core Agent Interface. There are also a number of examples showing the typical use and manner of application of these tags and parameters.

- *A (Anchor Tag)*
- *DEFTAG*
- *FORM*
- *INCLUDE*
- *IF*
- *LOOP*
- *SCRIPT*
- *SETVAR*

This guide only explains the special Asset Core Agent Interface tags and the Asset Core Agent Interface extended tags. They are all based on standard HTML version 4.0. If you are not familiar with HTML yet, it is recommended you refer to a standard HTML guide first.

A (Anchor Tag)

The A tag denotes an *anchor*- a hypertext link or the destination of a link. This section will only deal with the additional functionalities of the anchor tag created by BMC Software. For more information on the anchor tag's standard functions refer to a general HTML 4.0 reference. Contrary to the general anchor tag of the standard HTML versions, the BMC Software extended anchor tag does not use the *href* parameter to define links in the source file. It uses a tag called `htmlfile` with additional parameters and variables to pass the necessary information on to the Chilli programming language. Chilli will then translate this information into the standard *href* parameter.

Start Tag	required
End Tag	required

Syntax	`<A htmlfile='file name' [vars='variable names']></A>`

Mandatory Parameters

htmlfile	The htmlfile parameter defines the file to be displayed next on the screen.

Optional Additional Parameters

vars	The anchor tag can also mention variables (vars) to be passed through to the scripts included in the HTML file.

htmlfile

The BMC Software extended anchor tag expects the htmlfile parameter to define the file to be displayed on the screen. The file path can either be relative to the current directory or absolute.

Example

You can use any Chilli supported operations, specified within parentheses, so that the `htmlfile` parameter is calculated. For instance, if `_hchldir` equals to `"/myhchl"`, then the following anchor tag in an HCHL page:

```
<A htmlfile=(_hchldir + "/editquery/editquery_utils.hchl")>
```

will be translated by the parser into the following:

```
<A href="/myhchl/editquery/editquery_utils.hchl">
```

vars

The anchor tag can also specify a list of variables (vars) to be passed through to the next script.

DEFTAG

The DEFTAG tag provides a means for agent interface developers to define their own tags that call user-defined Chilli procedures. (The majority of commands in the agent interface are executed through Chilli scripts.)

Start Tag	required
End Tag	forbidden
Syntax	`<DEFTAG name='tag name' 'proc='procedure name' [parseoutput]>`

Mandatory Parameters

name	The name of the tag to be defined.
proc	The name of the Chilli procedure which will handle the tag when it is encountered in an HCHL file (=calling a Chilli procedure).

Optional Additional Parameters

parseoutput	Defines if the output of the execution of the chilli script is to be parsed again. If the parseoutput parameter is not mentioned in the tag, the Chilli generated output will not be reparsed before being passed to the browser.

name

The BMC Software DEFTAG expects the name parameter to define the name of the new tag. The naming conventions for new tag names are the same as the naming rules in the Chilli language, that is:

- The name may have a maximum length of 32 characters.
- Thename may be any combination of alphanumeric characters (that is, letters, digits, and underscores.
- The name may be in lowercase or uppercase or a mixture of both.
- Thename may start with an underscore (_) or a letter, but not with a digit.
- The name must not contain any spaces.
- The name must not be a reserved statement, a function name, or a keyword.
- The name must not be an already declared variable or constant.

Example

The following example shows an excerpt that creates a tag called SUBMIT that draws the submit buttons (Apply, Reset, Cancel, etc.) on the screen.

```
<SCRIPT>
   proc DrawSubmitTag (HtmlTagList HtmlFile, int index)
     string szFormName, szTarget, szButtons
     int iButtons
     GetTagParamValueStr (HtmlFile, index, "formname", "", szFormName)
     GetTagParamValueStr (HtmlFile, index, "target", "", szTarget)
     GetTagParamValueInt (HtmlFile, index, "buttons", SUBMIT_ALL, iButtons)
     DrawSubmit (szFormName, szTarget, iButtons)
   endproc
   </SCRIPT>

   <DEFTAG name=SUBMIT proc=DrawSubmitTag>
```

proc

The BMC Software DEFTAG expects the

`proc` parameter to declare the name of the procedure which defines the new HCHL tag. The naming rules for the procedure are the same as those for the new tag name. The defined procedure has a fixed signature and must include the following arguments:

Syntax

```
proc NewProcedure (HtmlTagList HtmlFile, Int Index)
```

HchlFile	contains the list of all tags called by the procedure. For more information on this structure, see the Chilli Reference Manual in Section II under the heading HTML Functions.
Index	the number of tags called by the procedure.

Example

The following script excerpt of an HCHL file defines two new tags PRINT and MAILTO in its procedures and declares them then through the DEFTAG.

```
<SCRIPT>
   ....
   proc HandlePrint (HtmlTagList HtmlFile, int index)
        print (StrEvalAsString (HtmlFile.Tags[index].TagParams[1].TagParamValue))
   endproc

   proc HandleMailto (HtmlTagList HtmlFile, int index)
        string mailto
        mailto = StrEvalAsString (HtmlFile.Tags[index].TagParams[1].TagParamValue)
        print ("&lt;A href='mailto:" + mailto + "'>" + mailto + "&lt;/A")
   endproc
   ....
 </SCRIPT>

   <DEFTAG name=PRINT proc=HandlePrint>
   <DEFTAG name=MAILTO proc=HandleMailto>
```

parseoutput

If the parseoutput parameter is supplied with the DEFTAG tag, when the script is run, the output is parsed again. If the `parseoutput` parameter is not mentioned, then the output, in whichever format it was returned by the script, will display on the screen.

Example

The following script is an excerpt of an HCHL file that defines a procedure to display panels on the screen.

```
<SCRIPT>
   proc DrawPanelTag (HtmlTagList HtmlFile, int index)
     string title, width, height, name
     int buttons, NoTitle
     bool fNoTitle
     GetTagParamValueStr (HtmlFile, index, "title", "", title)
     GetTagParamValueInt (HtmlFile, index, "buttons", 0, buttons)
     GetTagParamValueStr (HtmlFile, index, "name", "", name)
     GetTagParamValueStr (HtmlFile, index, "height", "400", height)
     GetTagParamValueStr (HtmlFile, index, "width", "250", width)
     GetTagParamValueInt (HtmlFile, index, "notitle", 0, NoTitle)
     ....
     if (NoTitle == 1)
          fNoTitle = TRUE
     else
          fNoTitle = FALSE
     endif
     DrawPanel (title, buttons, name, height, width, ...., fNoTitle)
  endproc
  </SCRIPT>

  <DEFTAG name=PANEL proc=DrawPanelTag parseoutput>
```

FORM

The FORM tag defines an interactive form and is used in the agent web interface in its standard format with additional BMC Software specific functionality. For more information on the FORM tag in general. refer to a standard HTML 4.0 reference.

Forms are generally created for data input. When the user submits the form through an input or button element the form values are submitted to the URL given in the form's required action attribute.

The FORM tag was extended by BMC Software through the htmlfile parameter to allow the agent interface to execute specific actions. It is not necessary to specifically include the action and method parameters, as they are directly included into the parser: action always is 'mwcparse' and method is 'post'.

Start Tag	required
End Tag	required
Syntax	`<FORM htmlfile='filename.html'></FORM>`

Mandatory Parameters

htmlfile	The htmlfile parameter defines the file to be loaded next. This file contains the necessary Chilli scripts to process the entered data.

Optional Additional Parameters

There are no optional parameters for this function.

To enter and process data via the FORM tag in the agent interface two files are necessary:

1. An HTML file containing the FORM with its fields to be filled in and the *htmlfile* parameter.
2. A file containing Chilli scripts to process the data entered into the FORM of the HTML file called through the *htmlfile* parameter in the first form sheet HTML file.

htmlfile

The BMC Software extended FORM tag expects the htmlfile parameter to define the file which will compile and execute the data entered into the form sheet. The file path can either be relative to the current directory or absolute.

Example 1

myfile.html:

```
....
   <FORM htmlfile=DataProcess.html>
   Enter Data:
     <INPUT name=Data type=text>
     ......
     <INPUT name=submit type=submit>
   </FORM>
```

When the form is submitted, it calls the file DataProcess.html, which processes all entered data through Chilli scripts.

DataProcess.hchl

```
   1     <SCRIPT>
```

```
2          extern string Data
3     </SCRIPT>

4     <SCRIPT>
5          if (defined (`Data'))
6               print ("Your data is " + Data.s)
7          else print ("No data specified")
8          endif
9     </SCRIPT>
```

Line 1-3:

Before any of the entered data can be processed in this file, all data variables need to be declared through the external keyword in a structured script. If the variables are not specifically declared at the beginning of the file, the following script will not compile because it cannot find the variables.

In this example, the "Data" variable has the suffix ".s" to indicate that the variable is a string variable. The variables may optionally be specified in the form of var.suffix, if the value of the variable might be ambiguous. The parser will automatically add the respective suffix if it is not mentioned.

Possible suffixes are:

```
i = integer
c = character
f = float
d = double
b = boolean
s = string
```

Line 4-9

After the variables are declared, a script is needed to compile all entered data. In the above example, this script could consist of only lines 4, 6 and 9. However, if the "Data" field is left empty, this might cause the script to not compile or compile with an error. Therefore, it is recommended to always add an if/endif (lines 5-8) statement to avoid any compiling problems.

IF

The agent interface's HCHL pages may now also contain conditional sections through the integration of the IF tag. The IF tag consists in effect of four different tags -- IF, ELSEIF, ELSE and ENDIF -- used exactly the same way as the Chilli IF language statement. The IF tags can be nested to any level. Formore information on the IF tag, see the Chilli Reference Manual.

Start Tag	required
End Tag	required
Syntax	`<IF condition=(Chilli expression)><ELSEIF condition=(Chilli expression)> <ELSE> <ENDIF>`

Mandatory Parameters

condition	a value which is evaluated as an expression and determines if the statement following the condition is executed or not.

Optional Additional Parameters

There are no optional parameters for this tag.

condition

The condition parameter is a value which is always evaluated as an expression:

Example

```
<IF condition=(BROWSER_TYPE="Mozilla/3.6.13 [en] (X11; U; Linux 2.2.5-15 i568), Mozilla Firefox")>
    <P>Your browser is Mozilla Firefox version 3.6.13.
<ELSEIF condition=(BROWSER_TYPE="Mozilla/5.0 (compatible; MSIE 5; Windows 2000), Microsoft
 Internet Explorer")>
    <P>Your browser is Microsoft Internet Explorer version 8
<ELSE>
    <P>Your browser is not certified for use with the Asset Core Agent Interface v 10.1.
<ENDIF>
```

INCLUDE

The INCLUDE tag is a very useful tag to keep the HTML files unclustered and to guarantee the same appearance of all pages.

General information about the basic structure of all HTML files, such as background color and background image in the BODY tag, as well as the structure of titles and footers can be defined in special small files.They can then be included into all pages via the INCLUDE tag instead of being repeated in every single file. Many of the actions to be executed through Chilli scripts are very similar in quite a number of pages. Therefore, they also can be put into separate files, which will then be included in many other files through the INCLUDE tag.

This considerably reduces the work required to set up the pages as well as the file sizes. It furthermore facilitates the task of making changes to the basic appearance of the HTML files should the need occur. The INCLUDE tag will be replaced in its entirety by the contents of the included file.

Start Tag	required
End Tag	forbidden
Syntax	`<INCLUDE htmlfile='file name' [condition='condition value'] [onceonly]>`

Mandatory Parameters

htmlfile	The htmlfile parameter defines the name and path of the file with which to replace the INCLUDE tag. The file path is relative to the path of the current file into which the new file is to be included.

Optional Additional Parameters

condition	A value which is evaluated as an expression and determines if the file include is to be executed or not.
onceonly	The onceonly parameter is used to avoid the multiple loading of the same data, which causes an error and stops the execution of a script.

htmlfile

The file parameter defines the name and path of the file with which to replace the INCLUDE tag. The file path is relative to the path of the current file into which the new file is to be included. If, for example, the current file is located in a directory `webconsole/commonviews/user`, and the file containing all body information, located under directory `webconsole/lib` is to be included, the file information would read like this: `file='../../lib/Body.html'`.

Example

The included file in the following example is a file called `Body.html` which gets the background image file. If this is set to NONE it gets the wall paper color.

Source File

```
<INCLUDE htmlfile=../../ScriptLib/Body.html onceonly>
```

Output File

```
<BODY TEXT='#000000' LINK='#0000ff' VLINK='#800080' BACKGROUND='/Console/Icons/backgrounds/
blue.gif'>
```

onceonly

The onceonly parameter is used to avoid the multiple loading of the same data, which can cause an error and stop the execution of a script. This is used to make sure a file is only loaded once, as files can be called several times through the INCLUDE tag of included files, for example.

Example

The index file calls the following files containing the components required to create the base page:

```
<INCLUDE htmlfile="../common/scripts/defs.hchl" onceonly>
<INCLUDE htmlfile="../common/scripts/header.hchl" onceonly>
<INCLUDE htmlfile="../common/scripts/footer.hchl" onceonly>
```

Due to the `onceonly` parameter, the scripts contained in these files are only executed once, as required for the creation of the pages.

condition

The condition parameter is a value which is always evaluated as an expression, whether it is included in parentheses () or not. If the value is true, the included file is executed. If the value is false, nothing happens and the included file will be ignored. The default value is "true" (if the condition parameter is not specifically mentioned).

LOOP

The LOOP tag defines a section in an HCHL file which repeats an operation. This tag is very useful for listings of values where it is not known how many instances of the attribute exist. The loop tag is only used to initiate the loop action for a table; it cannot be seen in the output file's source code.

Start Tag	required
End Tag	required
Syntax	`<LOOP [condition='condition value']></LOOP>`

Mandatory Parameters

There are no mandatory parameters for this tag.

Optional Additional Parameters

condition	A value which is evaluated as an expression and determines if the loop is executed or not.

condition

The condition parameter is a value which is always evaluated as an expression, whether it is enclosed in parentheses () or not. If the value is true (or a non-zero value), the loop is executed. If the value is false (or zero), nothing happens and the loop will be ignored (that is, the parser jumps to the </LOOP> tag and continues with the next element). The default value is `true' (if the condition parameter is not specifically mentioned).

SCRIPT

Scripts offer authors a means to extend HTML documents in highly active and interactive ways. For example:

- Scripts may be evaluated as a document loads to modify the contents of the document dynamically.
- Scripts may accompany a form to process input as it is entered. Designers may dynamically fill out parts of a form based on the values of other fields. They may also ensure that input data conforms to predetermined ranges of values, that fields are mutually consistent, etc.
- Scripts may be triggered by events that affect the document, such as loading, unloading, element focus, mouse movement, etc.
- Scripts may be linked to form controls (for example, buttons) to produce graphical user interface elements.

In the agent web interface, the SCRIPT tag is mainly used to call Chilli scripts which do most of the internal work within the agent interface. However, the SCRIPT tag can also call other scripts, such as Unix shell scripts or others.

Scripts can be either included into the HTML file if they are only applicable in a specific situation, or they can be stored in separate files. Since most scripts are applicable in more that one situation, those are stored in separate HTML files under the ScriptLib directory. These scripts are called via the BMC Software INCLUDE tag. Depending on the context of the script, its execution can then result in other scripts being executed, in a direct HTML output or an output which is then parsed again before being passed on to the browser.

When encountering a SCRIPT tag in an HTML file, the parser checks the [ScriptInterpreters] section of the webconsole.ini file for an entry matching the supplied language name. If it does not find a non-blank value, it passes the tag and its contents to the browser without changes. If it finds an entry in the .ini file with a non-blank value, it assumes that the value is the path of the interpreter to be used for handling the script. In this case it copies the text between the <SCRIPT>and </SCRIPT> tags into a temporary file and executes it using the supplied path. The path may be relative (for example,

`sh.exe`) or absolute (`/bin/sh`) as long as it can be found by the system.

The default value of the language parameter is "Chilli". If the language specified is "Chilli" or the lang parameter is absent, the script tag will treat the value as a Chilli script. If the parameter value is not Chilli or empty, the parser will check the .ini file for the value specified in the lang parameter.

Start Tag	required
End Tag	required
Syntax	`<SCRIPT [language='script language'] [vars='variable name(s)'] [parseoutput]></SCRIPT>`

Mandatory Parameters

There are no mandatory parameters for this tag.

Optional Additional Parameters

language	The language parameter defines the program with which the script is to be executed.
vars	The variable parameter vars defines which variables are being used in the script.
parseoutput	If the parseoutput parameter is supplied with the SCRIPT tag, the output after the script has been run is parsed again by the parser.

parseoutput

If the `parseoutput` parameter is supplied with the SCRIPT tag, the output after the script has been run is parsed again by the parser. If the `parseoutput` parameter is not mentioned, the output will be directly displayed on screen.

Example

The following code is from the HCHL file creating the layout of the Application Kiosk main page in the browser:

Source File

```
<SCRIPT parseoutput>
 Print ("<INCLUDE htmlfile='"+ HTTP_DOCUMENT_ROOT + "common/scripts/tabs.hchl'>" + ENDLINE)
</SCRIPT>
```

vars

The variable parameter vars defines which variables are used in the script. The variables need to be defined before the SCRIPT tag through the SETVAR parameter. If the script language is Chilli, no vars parameter is needed, as the Chilli scripts are run inside the HTML pages, where all variables are already mentioned. All other scripts run outside the HTML pages and need the variables specified in order to run.

language

The `language` parameter defines the program with which the script is to be executed. The default value is `chilli.` If the language parameter is missing, the parser automatically interprets the language parameter as being Chilli.

Example 1

The following script is written in JavaScript and after execution shows the language choices for the the agent interface.

```
<!-- Start location -->

<IMG src="/common/images/bulletsubsection.gif" width="13" height="11">
<SPAN class="LOCATION">
    <SCRIPT language='Javascript'>document.write (FindTranslation ("_MENU_KIOSK_"));&lt;/SCRIPT>
</SPAN>

<IMG src="../../../common/images/bulletsubsection.gif" width="13" height="11">
<SPAN class="LOCATION">
    <SCRIPT language='Javascript'>document.write (FindTranslation ("_MENU_LIST_"));&lt;/SCRIPT>
</SPAN>
```

```
<!-- End location -->
```

Example 2

An HTML file contains the following SCRIPT tag:

```
<SCRIPT language= bourneshell>
```

When the parser encounters this tag it will check the [ScriptInterpreters] section of the webconsole.ini file for an entry called bourneshell. This section contains the following entry:

```
[ScriptInterpreters]
BourneShell=/bin/sh
```

This entry tells the parser that the executable with which to execute the script is a Unix shell script called "sh", which is located in the local bin directory. Note that the language name and the entry in the .ini file are NOT case sensitive, so 'bourneshell' and 'BourneShell' are equivalent.

The parser now places the text between the <SCRIPT> and </SCRIPT> tags into a newly created temporary file and executes this file with the unix shell "sh".

SETVAR

The SETVAR tag defines variables for use in the HCHL file. The variables created through SETVAR have global scope. Variables are named storage locations capable of containing a certain type of data, such as a numerical value or string of text used in the program that can be modified during program execution. These variables can be used directly by a tag or a script that follows the variable definition. They can also be used by files that are called through the INCLUDE tag. Contrary to the other BMC Software tags, the SETVAR tag does not produce any direct output.

Start Tag	required
End Tag	forbidden
Syntax	`<SETVAR name='variable name' value='value of variable>`

Mandatory Parameters

name	The name parameter defines the name of the variable.
value	The value parameter is an expression.

Optional Additional Parameters

There are no optional parameters for this tag.

name

The name parameter defines the name of a variable to be used later in a script within the HTML page. The naming conventions for new variable names are the same as the naming rules in the Chilli language, that is:

- The name may have a maximum length of 32 characters.
- The name may be any combination of alphanumeric characters (that is, letters and digits, and underscores.
- The name may be in lowercase or uppercase or a mixture of both.

- The name may start with an underscore (_) or a letter, but not with a digit.
- The name must not contain any spaces.
- The name must not be a reserved statement, a function name, or a keyword.
- The name must not be an already declared variable or constant.

Example

```
<SETVAR name="_hchldir" value="/myhchl">
<SETVAR name="_id" value="345">
...
<A htmlfile=(_hchldir+"/mypage.hchl?_id="+_id)>
```

value

The value parameter is an expression representing the value of the defined variable. The value can be a simple integer or string or it can be a rather complicated function expression surrounded by parentheses.

Customizing Asset Core Reports

The Asset Core Console comes with two different types of reports: style-based and template-based reports. Both types may be customized, however, while template-based reports are generated via a quite complicated XML file it is very easy to customize style-based reports according to your requirements.

Style-based reports are based on a layout type that defines the number of subreports the report contains and how these subreports are ordered on the displayed or printed page. The appearance of all subreports is based on a css style sheet.

You may customize the following elements of this report type:

- the report logo
- the css style sheet to modify the overall appearance.

Customizing the Report Logo

Asset Core allows you to store more than one logo, so you can use different ones for different reports if necessary.

1. Go to the *[BMC Installation Directory]*`/data/Vision64Database/reports/common/images/logos/` directory.

   This directory contains all logos that can be used in reports. The default logo, `BMC.png`, comes with Asset Core.

2. Either modify the existing logo or copy the new logo to this location.

   The logo file must be in `.png` format and have a size of 272 x91 pixels. If it is larger it will be cut down to the right size.

The new or modified logo is now available via the Console and will appear in the dropdown list of the **Logo** field in the **Properties** window when creating a new or modifying and existing report.

Customizing the CSS Style Sheet

Asset Core comes with two css style sheets:

- Numara.css

  This style sheet is the default css, it has a fixed width of 1024 pixels.

- Compatible.css

  This style sheet has the same values as the `Numara.css` sheet apart from the width which is not fixed. It is used for the existing reports after upgrading from a pre 10.1 version.

1. Go to the<InstallDir>/data/Vision64Database/reports/common/css directory.

   This directory contains all css files that may be used for the reports.

2. Either modify the existing css file or copy the new css file to this location.

The new or modified style sheet is now available via the Console and will appear in the dropdown list of the **Style Sheet** field in the **Properties** window when creating a new or modifying and existing report.

Localizing Asset Core to an Unsupported Language

Asset Core is available in the following languages:

- British English
- American English
- French
- German
- Japanese
- Spanish

Localization comprises translations of all elements of the Console GUI, balloon tips, and Instant Expert.

In addition to the six available languages, you can localize Asset Core in an unsupported language. This process consists of the following steps:

Localizing the Console

You can localize the following elements of the Console:

- elements of the GUI like the names of buttons, nodes, tabs, etc.
- balloon tips
- Instant Expert

The translations for these elements are saved in different files.

To localize the Console

Adding Languages to the Database

All available languages for localization are in a specific database table called *Enumeration*. When you want to add a new language to the Console, you must add a new entry to the table. To add a new entry:

1. Copy the following command and replace the four sample values in the second line with your information:

```
INSERT INTO Enumerations (EnumID, EnumGroup, EnumName, EnumValue)
  VALUES (<userinput>11905, 'AvailableLanguages', '_DB_LANGUAGE_NEWLANGUAGE_',
 'NewLanguage'</userinput>);
```

The four expressions represent:

Expression	Value	Information
EnumID	`11905`	Unique identifier for any option in the database. You may use any value between `11950` to `11999`. If you add more than one language option make sure to use different EnumIDs.
EnumGroup	`AvailableLanguages`	Type of the enumeration. In this case the value must always be `AvailableLanguages`.

Expression	Value	Information
EnumName	`_DB_LANGUAGE_NEWLANGUAGE`	Keyword for the new language, for example, _DB_LANGUAGE_CHINESE.
EnumValue	`NewLanguage`	Name of the language (for example, `Chinese`). Do not use special characters such as ç orñ.

2. In the *Enumeration* table of your database execute the modified command.

The new language is added to your database. In thenext step, you add a .locale files for your new language which contain the translations.

Creating Localization Files

Localization files are text files with the extension *.locale*. To add a new language you need to create three new *.locale* files. To create the new .locale files:

1. On the device on which the Console is installed go to `<BMC Installation Directory>/ui/console/jws`.
2. Open the `NumaraFootPrintsAssetCore.jar` file with a file archiver such as WinZip or WinRAR.
3. Extract its `locales` folder to the `jws` folder.
4. In the `NumaraFootPrintsAssetCore.jar` file delete the `locales` folder and close it.
5. Open the `locales` folder and add the following line to each `<Language>.locale` file:

   `_DB_LANGUAGE_NEWLANGUAGE_=LanguageValue`

   To add Chinese as a new language add: `_DB_LANGUAGE_CHINESE_=Chinese` to `English.locale` or `_DB_LANGUAGE_CHINESE_=Chinois` to `Francais.locale`.

   `Language Value` is the value that will be displayed among the language options in the **Preferences** dialog, from which you choose the language of the Console.

6. Duplicate a *<Language>.locale* file and rename it so that it matches `EnumValue` of your new language that you added to the database.

   To create a Chinese *.locale* file rename it to `Chinese.locale`.

7. Repeat the last step for *<Language>Params.locale* and *<Language>_GuidedHelp.locale*.

   If you don't create three new *.locale* files, the english.*locale* file will be used in place of the missing file.

8. Translate all translation values in the files (the expression to the right of the equal sign (=).
9. To verify your translations, launch a Console and select your newly added language from the **Language** drop-down list in the **Preferences** dialog.

   The Console is displayed in your new language.

You created the three required *.locale* files and localized the Console to a new language.

Localizing the Agent Interface, Reports, and E-mails

In addition to the Console you can also localize the following elements of Asset Core that are accessible outside of the Console:

- Agent Interface in a browser
- reports created in the Console and displayed in a browser or with other software programs
- E-mails sent by the Console

To localize the Agent Interface, reports, and E-mails:

Adding Language to Agent Interface Files

To add a new language to the Agent Interface you need to define the new language in a file, as well as an image of a flag that represents it. To add a language:

1. Go to `[BMC Installation Directory]/ui/common/images` and check if there is a matching `LANG_YourLanguageAbbreviation.gif` file for your language.
   - If there is no such file, create a new `.gif` file of the flag representing your language with dimensions of 16 x 11 pixels.
2. Open the `[BMC Installation Directory]/ui/common/scripts/menu_items.js` file in a text editor.
3. Replace the three variables in the following line with your information and add it after line 29:

```
[
<img style=\"position:absolute;top:5px;left:7px;\" src=\"/common/images/
LANG_<varname>YourLanguageAbbr</varname>.gif\" width=16 height=11>
<div style=\"position:absolute;top:4px;width:46px;left:27px;
\" align=left><varname>YourLanguage</varname></div>',szURL +
 '_language=<varname>YourLanguage</varname>']
```

   To add Chinese as a new language, replace *YourLanguageAbbr* with `CN` and *YourLanguage* with `Chinese`.

4. Save the file and close it.

You included the new language with its matching flag. In the Agent Interface the new language can be selected, but the translations are still missing. In the next step you add the translations to the database.

Adding a Language to a SQLite Database

To add a language to a SQL database you need:

- your new as well as the existing <*Language*>*.locale* files as described in XXX.
- Asset Core Installation file

Contrary to the Console the localization data of the Agent Interface, reports and E-mails are contained in a small SQLite database to which all new languages must be added.

To add a new language to the SQLite database:

1. Copy the files `Locale2SQLite.bat`, `Locale2Sqlite.jar` and `sqlite.exe` to any folder on your computer.
2. Open `Locale2SQLite.bat` in a text editor and modify the command as follows: `java -jar Locale2Sqlite.jar "[BMC Installation Directory]/ui/console/jws/locales"`
3. Save `Locale2SQLite.bat` and double-click it.

☞ The command line displays and the SQL file is generated.

4. Wait until the command line closes.

   ☞ In the folder a `translation.sqlite` file is created.

5. Copy the `translation.sqlite` file to *`[BMC Installation Directory]`*`/ui/common/dict` and *`[BMC Installation Directory]`*`/data/core` which will overwrite the existing files.
6. To verify your translations open the Agent Interface and select your newly added language from the drop-down list on the top left.

   ☞ The Agent Interface is displayed in your new language.

How to Translate .locale Files

What is a .locale file?

A *.locale* file is a text file with the extension *.locale*. It contains all keywords used by Asset Core andits respective translations. There is a separate.*locale* file for each language.

How is a .locale file structured?

A *.locale* file has the following structure:

```
KEYWORD1=Keyword1Value KEYWORD2=Keyword2Value KEYWORD3=Keyword3Value ...
```

Each pair of keywords and values is on a separate line. The keyword and its value are connected by an equal sign (=). A *.locale* file must be in UTF-8 format.

What should I translate in a .locale file?

In a *.locale* file only translate the value of a keyword, which is the expression on the right side of a a equal sign (=).

Never modify any part of the keyword, which is on the left side of the equal sign. If you do, the link between keyword and translation breaks and Asset Core keywords appear instead of translations.

What is the syntax of a keyword?

A keyword has the following characteristics:

- The general syntax of a keyword is the following: *`_SECTION_ACTION_OBJECT_`*
  - For the most important *SECTION*s see the table.
  - *ACTION* is generally composed of a verb only or a verb plus its child
  - *OBJECT* is the element on which the action is executed, for example,

    `_ASSIGNGROUP_QUERY_` (Assign a group to a query)
- Each keyword starts and ends with an underscore, with the exception of values directly coming from the Asset Core Agent database, which may not be forced into this scheme. These keyword values appear as they are (. that is, as simple text, such as *DebugLogMax)*
- The 'name' part of an element name is generally dropped for the keyword (for example, _COLNAME_DEVICE_ for the table column "Device Name".
- The following abbreviations are used:
  - Administrator = ADMIN
  - Operational rule(s) = OPRULE(S) or OR(S)
  - Parameter(s) = PARAM(S)

- Attribute(s) = ATTR(S)
- Database Server(s) = DBSERVER(S)
- Hardware Inventory = HWINVENTORY
- Software Inventory = SWINVENTORY
- Transfer Window Folder = TWFOLDER

The most important *SECTION*s are:

Name	Description
ACTION	Name of an action, menu item, window, button, etc.
AGTMOD	Any type of values or messages being generated from the Asset Core Agent
CLASS	Name of an action, menu item, window, button, etc.
COLNAME	Name of a table column
CONSOLE	Any type of values or messages being generated from the Asset Core agent
CONST	Constants in drop-down lists in dialogs
DB	Values from the database
ERROR / ERRORCODE	Text returned by errors
HEADER	Title right window pane if different from the node name
HOME	All elements which may appear on the home page
LABEL	Any type of item appearing in the main window of the Agent Interface, such as field names, etc.
MENU	Name of menus, those of the Console as well as those of the Agent Interface
MESSAGE	Any content appearing in simple message boxes on the screen
MISC	Any item appearing in the right window panes which are not fields or table elements, or do not fit in any of the other sections.
MSI	MSI package specific items in the right window panes
NODENAME	Name of the tree node in the left window pane
NOTE	Explanations of the contents of a browser page
POPUP	All items contained in a popup window
PREF	All elements of the **Preferences** dialog

Name	Description
SCHEDULE	All schedule specific items
SEARCH	All elements of the search tab
SNPXXX	Snapshot specific items in the right window panes
STATUSBAR	Info appearing in the status bar of the main window
SUBTITLE	This is the prefix for all subtitles of the Agent Interface
SWINV	Elements with this prefix pertain to software inventory
TABNAME	Name of the tabs in the right window pane
TITLE	Titles of the Agent Interface
TOOLTIP	All tooltip expressions -- for each action there is a tooltip equivalent
WINTITLE	Title of a dialog

Is there anything else I need to pay attention to?

Make sure that:

- you do not modify any keywords
- you do not modify the tructure of the file with a pair of keywords and values per line
- every space ( ) in the value is preceded by a backslash (\), for example, `Create\ Package\ Folder...`
- you save the file in UTF-8 format
- you do not modify the file name

Can I add new keywords to the .locale file?

You can add new keywords to the.*locale* file, but they will have no functionality in Asset Core.

Do I have to translate all keyword values?

If you want all elements of Asset Core to appear in your new language, you have to translate all keyword values of the *.locale* file. You can also just translate keyword values that are important to you and leave the rest in the original language of the *.locale* file.