Platform Express Component Integrator's Guide

Software Version 1.1

Release 1.1

Table of Contents

Table of Contents (cont.)

List of Figures

List of Figures (cont.)

About This Manual

This manual is for engineers who create components for use with Platform Express. The Platform Express application enables rapid creation and initial debugging of system designs based on standard processor platforms. To use this manual effectively, you should have knowlege of embedded system design concepts, including the use of Hardware Description Languages, embedded processor programming, and simulation. Additionally, you need to have a basic understanding of the Extensible Markup Language (XML). For some aspects of Platform Express component development, you need to know Java programming.

Manual Organization

This manual contains the following chapters:

Chapter 1, "Components," describes what a component consists of and provides guidelines for integrating a component into Platform Express.

Chapter 2, "User-Input Parameters and Configurators," which discusses how to handle user-configurable parameters for components.

Chapter 3, "Generators" discusses generators, which are Java classes that create HDL code, software, simulation environment scripts, simulation stimulus or anything else that contributes to building and verifying a design in Platform Express.

Chapter 4, "Decoder Templates" discusses bus decoder template files, which specify the hardware logic and connections that need to be created to allow a peripheral to function properly on a particular bus.

Related Publications

See the online User's Guide that is accessable from the Platform Express main window for additional information on using components within a design. For an introduction to XML, see the following web pages:

- http://www.xml.com/pub/a/98/10/guide0.html
- http://www.w3schools.com/xml/default.asp

Chapter 1
Components

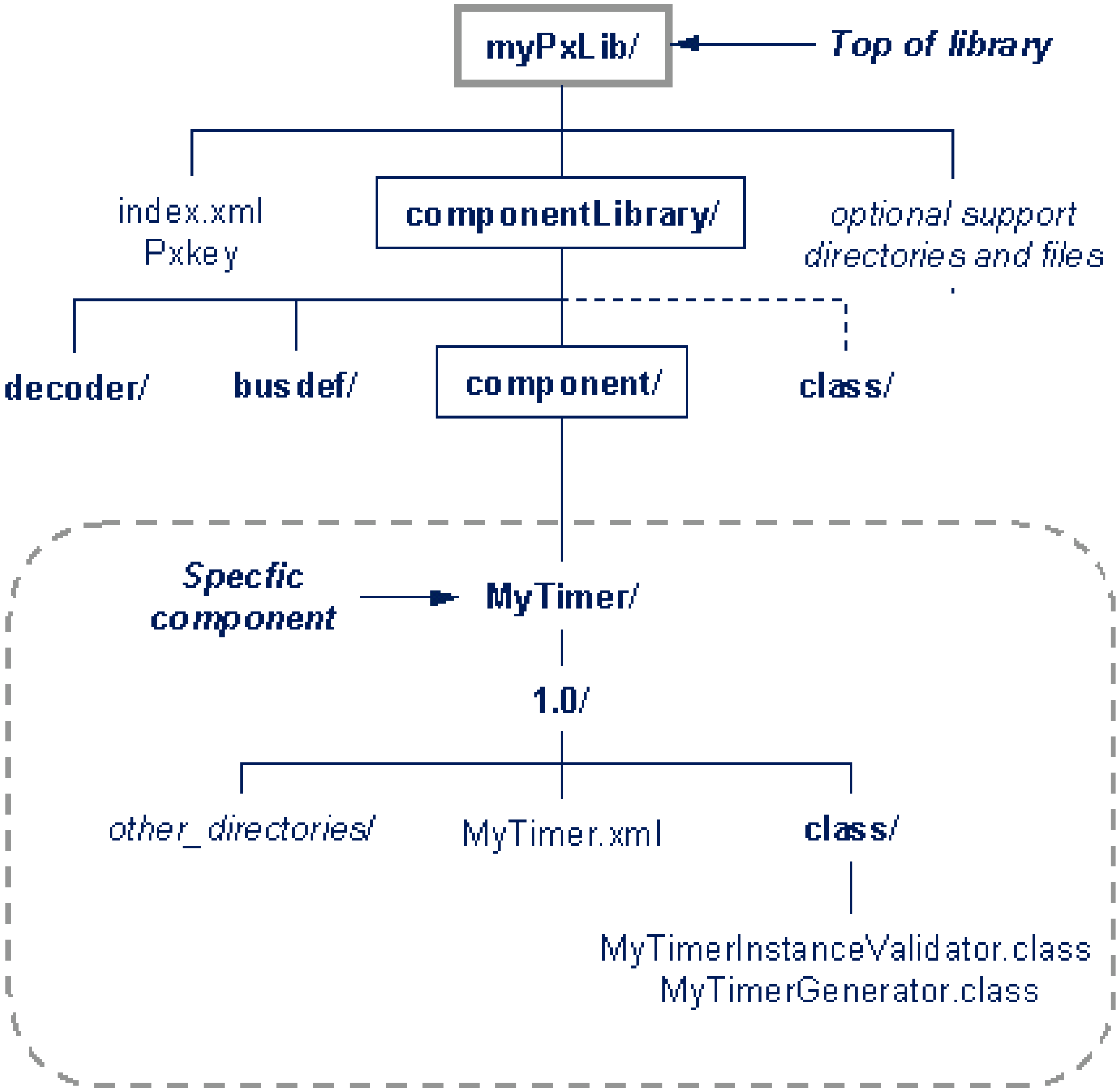
A Platform Express component is a set of files containing all the information that Platform Express needs to instantiate the component into a design and verify that it works correctly within the design. This chapter describes what a component consists of and provides guidelines for integrating a component into Platform Express.

Defining a Component

A component resides in a single uniquely named directory within a Platform Express component library. Figure 1-1 shows the structure of an example component that resides in a component library created for Platform Express. A component library is simply a defined directory structure. The root directory for a library should have a name that indicates what kind of components the library contains. Beneath the root directory, a directory named *componentLibrary* contains all the components, along with an *index.xml* file, which is an index of the components that the library contains, the Pxkey file, which contains licensing information, and optional directories and supporting files as necessary. Platform Express accesses component libraries either through the PXPATH environment variable or through a default search method; this is discussed in more detail in "Packaging Components," later in this chapter.

The *componentLibrary* directory may contain any of the following subdirectories: *component*, *busdef*, *class*, *generator*, or *decoder*. The *component* directory may contain any number of components, each with a standard directory structure. The root directory of the component in this example is *MyTimer*. Each component has to have a name that is unique across all component libraries. Within the component directory are one or more directories containing specific versions of the component; in this example the only version present is *1.0*.

Figure 1-1. Typical Component Directory Structure



The *1.0* directory and its contents actually define the component. For the MyTimer example, the files and directories are follows:

- The XML file *MyTimer.xml* specifies the component structure in detail. All components must have such a component-definition file, and this is the only item that is required to be in the *version* directory. This file names the component, defines the component's I/O pins (names, active logic level, and so on), and much more.

- The files *MyTimerGenerator.class* and *MyTimerInstanceValidator.class*, are Java object files for a generator and an instance validator, respectively. Notice the *class* directory can be present under the *componentLibrary* directory as well as under the component itself. Generators are discussed in Chapter 3. Validators are discussed in Chapter 2, under "Validators."

- Other files and subdirectories may be present, but these depend entirely on the how the component is specified in the XML file. Components do not necessarily need such directories, and they need not be named in any particular way.

The component structure is covered in more detail in subsequent sections.

General Procedure for Creating a Component

Here is a suggested process for creating a Platform Express component:

1. Gain an understanding of XML and XML Schema syntax. See "Understanding XML and XML Schemas," which follows this section.

2. Gain a general understanding of the Platform Express component structure. You should study the syntax and structure of the XML-based component definition file and be aware of the kinds of supporting files and programs that may be required for a component. This is described in detail in "Creating the Component Definition File," later in this chapter.

3. Find a component in an existing component library that is similar to the one you are designing and use it as a template.

4. Create a working component directory structure. You could copy one from an existing component.

5. Create a component definition (*component*.xml) file. Depending on how similar the component is to an existing one, the pxedit application supplied with Platform Express may be useful. See "Using the pxedit Application" later in this chapter for more information.

6. Study the Generator, Configurator, and Validator documentation and examples to determine which of these, if any, your component will need. If

required, create new ones and specify them in the *component*.xml file. See Chapter 2, “User-Input Parameters and Configurators,” and Chapter 3, “Generators,” for additional information.

7. Package the component and place it in a component library. Along with the *component*.xml file, you may need to package additional supporting files:

   - Place any software files (target processor code, initialization code, command files, and so on) required for the component in appropriate directories within the component structure.

   - Place any HDL simulation models and associated files required for the hardware portion of the component in appropriate directories.

   See “Packaging Components” later in this chapter for more information.

8. Run the mkIndex utility on the component library where you installed the component. This creates an index of all the components in the library and validates the component’s *.xml* file against the schema. Repeat this step until the *.xml* file is valid. See “Running the mkIndex Utility” for additional information.

9. Run the Platform Express license key generator for the component library, as described in “Licensing a Library.” This generates a key that Platform Express requires in order to use the library. A new key must be generated whenever anything within the componentLibrary subdirectory hierarchy is modified.

10. Test the component and integrate it into Platform Express:

    - Make sure the component library in which your component resides is specified in your PXPATH environment variable or is located in the pxLibraries directory.

    - Invoke Platform Express, note any parsing errors or other problems that Platform Express detects, and fix those problems.

- Add the component to a design and build the design. Examine the generated files and make any corrections needed to the generators or component structure.

- Run a verification session and check the component for correct behavior within the design.

Understanding XML and XML Schemas

Creating a component definition file requires some understanding of XML and XML schemas. Good starting references for XML and XML schemas can be found at the following locations:

- http://www.w3schools.com/xml/default.asp.

- http://www.xml.com/pub/a/2000/11/29/schemas/part1.html

The schema for component files can be found in the Platform Express installation directory (PXHOME) under the *schema* subdirectory. Documentation for these schema, in the form of HTML files that depict the structure of the schema, can be found in *$PXHOME/doc/schema*. Invoke your HTML browser on *index.html* to gain access to this documentation.

Understanding XPath and Platform Express Extensions

XPath is a language used to navigate through XML structures. In Platform Express it is useful to generator and configurator writers for locating specific component elements. It is also useful in the component and bus definition files for setting up dependencies on configurable values.

An XPath tutorial can be found at http://www.w3schools.com/xpath. The standard language definition can be found at http://www.w3.org/TR/xpath.

Platform Express has extended XPath to address some unique requirements. The following is the complete list of XPath functions added by Platform Express.

containsToken

boolean **containsToken**(*string, string*)

The containsToken function returns true if the first argument string contains the second argument string as a token, and otherwise returns false. To be interpreted as a token, the second string must be found within the first string, and be separated by white space from any other characters in the first string that are not white space characters.

Example: `containsToken('default spine driver', 'pin')` evaluates to false, whereas the standard XPath function *contains* would have evaluated true with the same arguments.

Purpose: Some attributes used in Platform Express are of type NMTOKENS which is a list of tokens separated by white space. This function allows XPath selection based on whether the attribute contains a specific token.

decode

number **decode**(*string*?)

The decode function decodes the string argument to a number and returns the number, or returns the NaN number if the string cannot be decoded. If the argument is omitted, it defaults to the context node converted to a string. If the string argument is a decimal formatted number, it is returned unchanged. If it is a hexadecimal representation starting with “0x” or “#”, it is converted to a decimal number and returned. If it is in engineering notation ending in a ‘k’, ‘m’, ‘g’, or ‘t’ suffix, case insensitive, the numeric part is multiplied by the appropriate power of two magnitude.

Examples: `decode('0x4000')` evaluates to 16384. `decode('4G')` evaluates to 4294967296.

Purpose: Platform Express allows numbers to be expressed in hexadecimal format and engineering format. When setting up dependencies on configurable values, it is sometimes necessary to perform some arithmetic in the dependency XPath expression. However, XPath only supports arithmetic on numbers and it only recognizes decimal strings as numbers. This function allows the alternate formats to be converted to numbers recognizable by XPath.

pow

number **pow**(*number*, *number*)

The pow function returns a number which is the first argument raised to the power of the second argument.

Example: `pow(2, 10)` evaluates to 1024.

Purpose: It is common for a Platform Express component to have a configurable number of address bits. When this happens, the size of the address range it occupies on a memory map varies exponentially with the number of address bits. This function gives XPath the mathematical capabilities needed to describe this relationship in a dependency expression.

log

number **log**(*number*, *number*)

The log function returns a number that is the log of the second argument in the base of the first argument.

Example: `log(2, 1024)` evaluates to 10.

Purpose: This is the inverse of the pow function. It is intended to express the reverse of the dependency described for the pow function. In this case the range of an address block might be configurable and the number of address bits might be expressed as a dependency of the address range using the log function.

Creating the Component Definition File

The component definition file is an XML file that describes the properties of a component. Each component has a unique component definition file named *component*.xml, where *component* represents the name of the component. An example is *timers.xml*. The first step in creating a component is to specify the component's properties in the *component*.xml file using a predefined set of XML elements. These elements are defined in the schema file *PxComponents.xsd*, which can be found in the *schema* directory under PXHOME.

A convenient starting point for creating a new *component*.xml file is to find an existing component that is similar and copy that file. The excerpt below is the beginning of a component definition file for a UART component.

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
Copyright Mentor Graphics Corporation 2001
All Rights Reserved
-->
<component xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:noNamespaceSchemaLocation="schema/1.0/pxComponents.xsd">
    <name>uart</name>
    <version>2.37</version>
    <busInterfaces>
        <busInterface interfaceId="ambaAPB">
            <busType>ambaAPB</busType>
            <slave>
                <memoryMap>
                    <addressBlock name="ambaAPB">
```

The file refers to the schema in *schema/1.0/pxComponents.xsd*, which is found under PXHOME. The first element of significance is the <name> element, which is *uart*. Just below that, the <version> element specifies the version number of the component, which is *2.37*. The <busInterfaces> element specifies all the buses that the component connects to. This component connects to the *ambaAPB* bus, as specified by the <busInterface> element. The uart component is a bus slave, as specified by the presence of the <slave> element. Additional subelements under <busInterface> specify the memory map and various other properties.

Top-Level Elements

A component is defined in terms of the following top-level XML elements, each of which can have numerous subelements:

- name

  The *name* element is the name of the component and must match the name of the component directory in which it is installed.

- version

  The *version* element is the version number assigned to the component and must match the name of the version directory in which it is installed

- busInterfaces

  Lists bus interfaces supported by the component. See “Bus Interfaces” for additional information

- componentInstances

  Reserved for future use in defining hierarchical components. Currently this element is only used in design files to list all the component instances in the design.

- busInstances

  Reserved for future use in defining hierarchical components. Currently this element is only used in design files to list all the bus instances in the design and describe their connectivity.

- addressSpaces

  For bus masters, lists all the address spaces defined for the component.

- presentation

  Contains information that affects the display of the component in various Platform Express views.

- hwModel

  Describes the hardware model including its signal list, its verification environment, and references to the files used in the verification environment.

- generators

  Lists all generators that the component requires. See Chapter 3, "Generators," for additional information.

- configurators

  Lists all the non-default configurators the component requires. See Chapter 2, "User-Input Parameters and Configurators for additional information.

- ui

  The *ui* element is discussed in Chapter 2, "User-Input Parameters and Configurators"

- fileSets

  Specifies files associated with the component.

- persistentInstanceData

  This is a container for any data that is specific to an instance of the design object. The contents are not interpreted or validated by Platform Express. This element is saved with the design and restored when the design is loaded. It is intended to be used by configurators and generators to store and retrieve instance-specific data.

Bus Interfaces

Each component has one or more bus interfaces that determine how it can be connected to other components. Platform Express considers all intercomponent connections to be through busses, including connections such as interrupt lines and clock input. The following excerpt (from the UART example) defines two bus interfaces: an ambaAPB bus and an Interrupt signal. The UART is an ambaAPB bus slave, as specified by the <slave> element. The <memoryMap> defines the memory space occupied by the UART: 6 bits wide by 16 bytes long, mapped to the low-order bits of the bus (bitOffset = 0). The base address for the UART is a user-input value, as indicated by the resolve="user" attribute (which is discussed in more detail in Chapter 2, "User-Input Parameters and Configurators"). In both the ambaAPB and Interrupt interfaces, the <signalName> element maps external bus signals to UART signals.

```
    <busInterface interfaceId="ambaAPB">
        <busType>ambaAPB</busType>
        <slave>
            <memoryMap>
                <addressBlock name="ambaAPB">
                    <baseAddress format="long" id="baseAddress"
                         prompt="Base Address:" resolve="user"
                              configGroups="requiredConfig"/>
                    <bitOffset>0</bitOffset>
                    <range>16</range>
                    <width id="width">6</width>
                </addressBlock>
            </memoryMap>
            <connection>required</connection>
            <signalMap>
                <signalName busSignal="PCLK">PCLK</signalName>
                <signalName busSignal="PRESETN">PRESETN</signalName>
                <signalName busSignal="PADDR">PADDR</signalName>
                <signalName busSignal="PWDATA">PWDATA</signalName>
                <signalName busSignal="PRDATA">PRDATA</signalName>
                <signalName busSignal="PWRITE">PWRITE</signalName>
                <signalName busSignal="PSELx">PSEL</signalName>
                <signalName busSignal="PENABLE">PENABLE</signalName>
            </signalMap>
        </slave>
    </busInterface>

    <busInterface interfaceId="Interrupt">
        <busType>singlePinInterrupt</busType>
        <slave>
            <connection>required</connection>
            <signalMap>
                <signalName busSignal="interruptAH">IRQ</signalName>
            </signalMap>
        </slave>
    </busInterface>

</busInterfaces>
```

Numeric Values

Numeric values in component files are interpreted as decimal, hexadecimal or octal numbers depending on their prefix.

Prefix:	**Format:**
none	decimal
0x	hexadecimal
#	hexadecimal
0	octal

A numeric value may also contain a magnitude specifier suffix.

Suffix:	**Multiplier:**
k or **K**	1,024
m or **M**	1,048,576
g or **G**	1,073,741,824
t or **T**	1,099,511,627,776

The following XML contains examples of numeric values:

```
<baseAddress>0x400000</baseAddress>
<addressRange>128k</addressRange>
```

Variables in Platform Express XML Documents

Platform Express is platform independent and generates files for many different programs such as operating system shells, Tcl intpreters, make, and so on. Because the variable syntax changes from one environment to another, variables in Platform Express XML documents must be expressed in a canonical form, and will then be converted to the appropriate syntax.

Variables in Platform Express documents take the following canonical form:

```
${variableName}
```

Reserved Variables

There are two Platform Express reserved variables:

```
${PXVAR_COMPONENT_LOCATION}
${PXVAR_COMPONENT_LIBRARY}
```

The first reserved variable can be used to refer to the top-level directory of the component. It will be resolved at Platform Express runtime. This is most commonly used to specify file pathnames, as in the following example:

```
${PXVAR_COMPONENT_LOCATION}/software/include/defs.h
```

For most file specifications inside a Platform Express component document, this variable is not needed. File specifications should be relative pathnames. Platform Express resolves a pathname at runtime, first looking for the file relative to the component directory, then looking relative to the component library that contains the component. Exceptions are the configurable elements <flags> that are specified for files and defaultBuilders. A typical flags argument may be something like this:

```
<flags>-c -g -DCPU=arm7tdmi
     -I${PXVAR_COMPONENT_LOCATION}/software/include</flags>
```

In this case, the **-I** switch in the flags specifies an include directory that the compiler will look in for included files. The variable for the component location is needed because the content of <flags> is just taken as a string.

The second variable, ${PXVAR_COMPONENT_LIBRARY}, will be resolved to the component library directory that contains the component. This would typically be used to specify common files in a component library that are used for more than one component.

Any other variable in the component *.xml* file will simply be converted to the appropriate variable syntax for the environment in which it is being generated.

Using the pxedit Application

For components intended for the ModelSim and Seamless simulation environment, the pxedit application supplied with Platform Express can significantly reduce the amount of hand editing required in creating component definition files. The application allows you to automatically generate bus interface mappings from compiled HDL models, as well as to edit various other aspects of the component definition files.

Invoking the Editor

1. Set the following environment variables:

   MODELTECH must point to the ModelSim installation directory.

   PXHOME must point to the Platform Express installation directory.

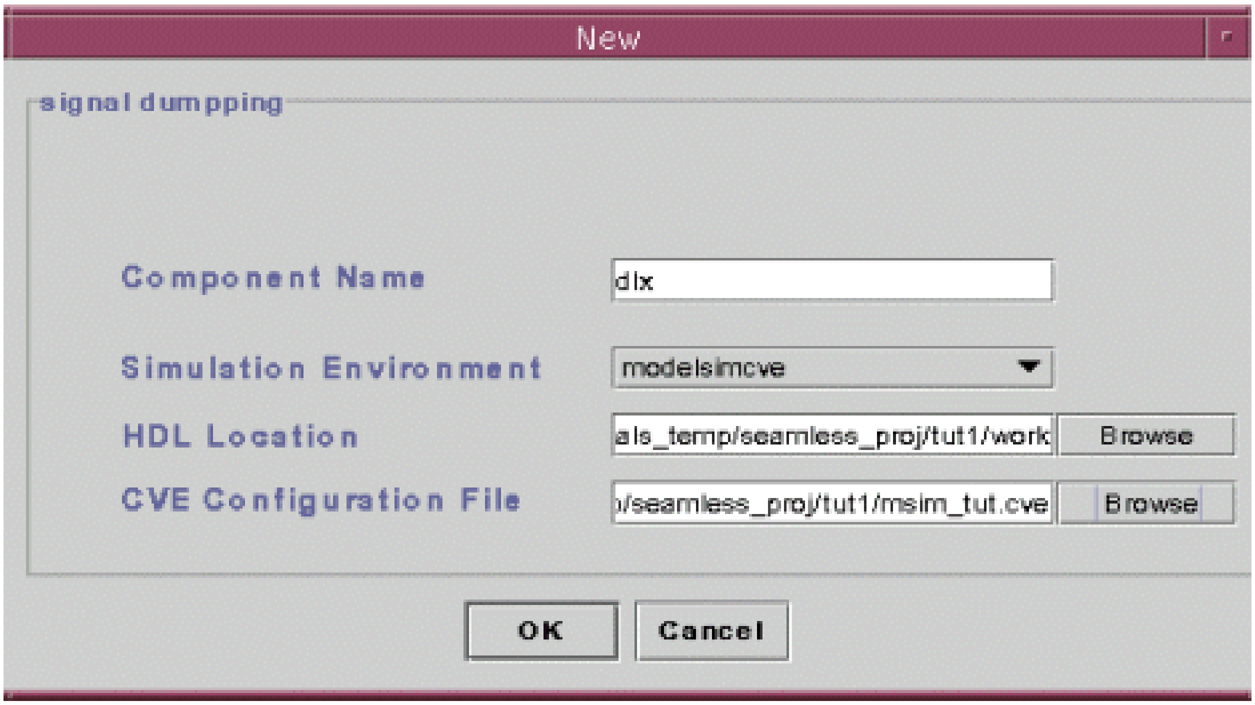
2. Invoke the pxedit application:

   ```
   $PXHOME/tools/bin/pxedit
   ```

   The pxedit main window appears.

Creating a New Component Definition File

1. In the pxedit main window, select **File** > **New**. The New dialog box appears, as shown below.



2. Enter or select the following information in the dialog box:

   - Top-level model name of the component.
   - Simulation environment. For example, for the ModelSim/Seamless CVE environment, you would select modelsimcve. The simulation environments are as follows:

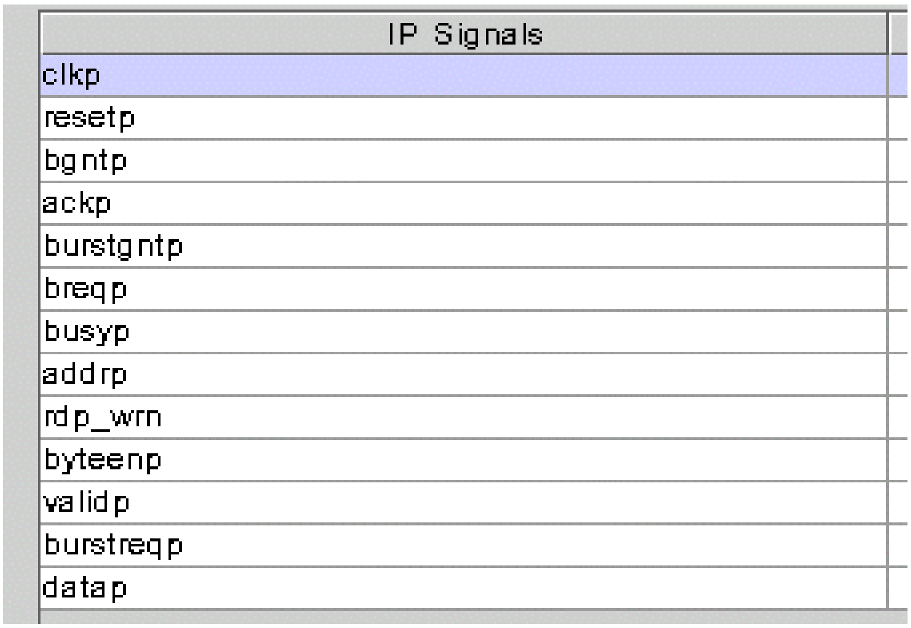
     modelsim: The component can be instantiated in a design with ModelSim alone.

     modelsimcve: The component requires Seamless libraries and can be instantiated only when used with Seamless CVE.

     Location of the compiled HDL library. This is typically the "work" directory.

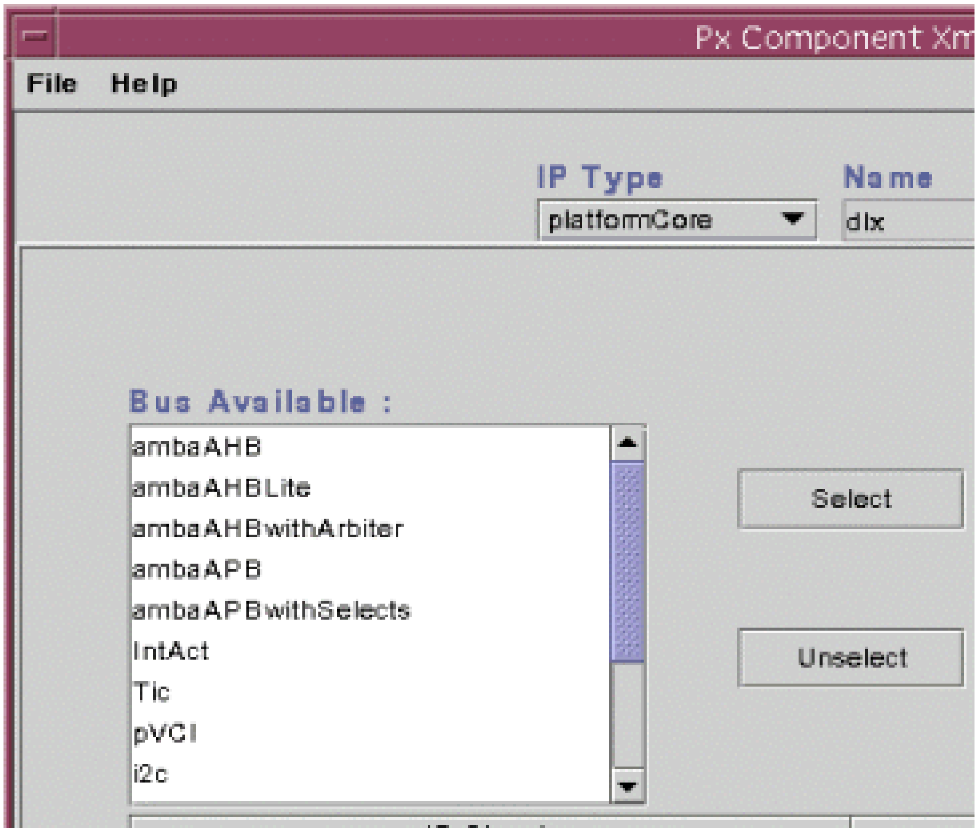
- For a design in the Modelsim/Seamless CVE environment, enter the Seamless CVE configuration filename.

3. Click OK. At this point, pxedit extracts the top-level signal names from the model. When it is finished, the pxedit application displays the signal list, as shown below. The pxedit main window presents a tabbed input area in which you supply information about the component.

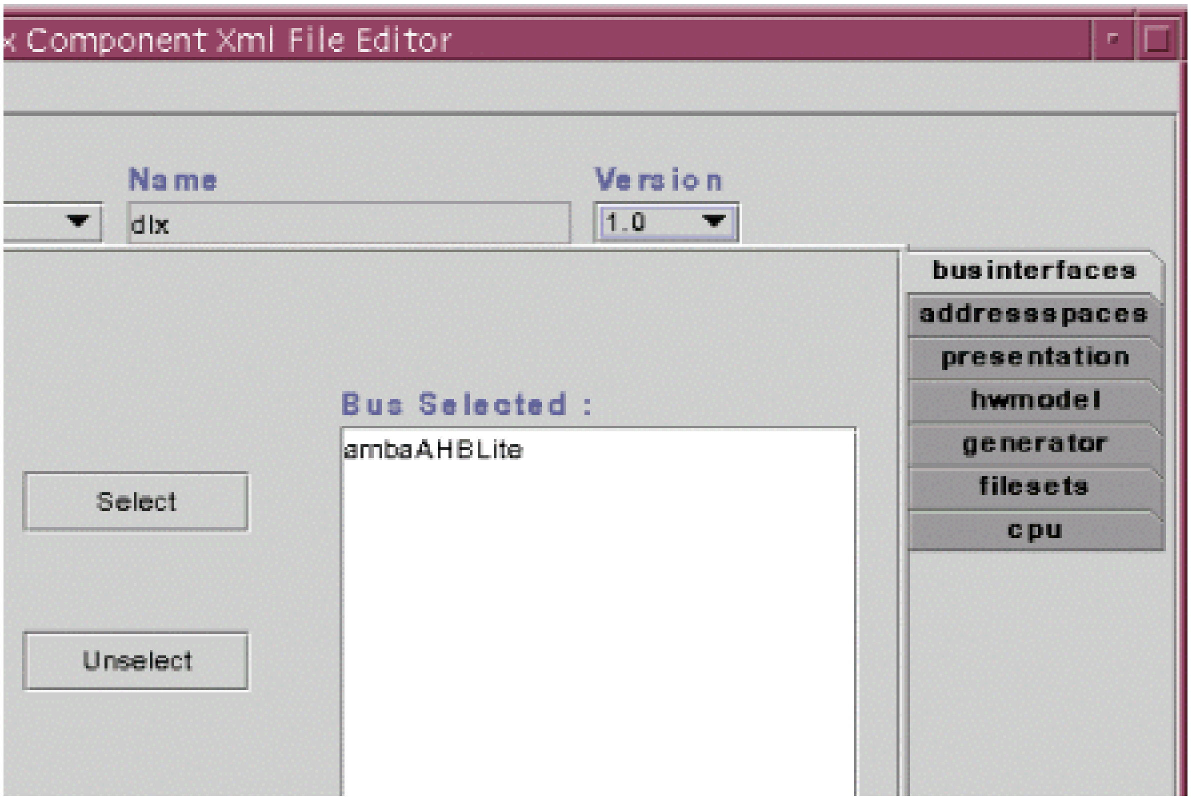


4. At this point, you can begin editing the information for the output XML file. You can start by selecting a Bus Interface. Shown below is the list of available buses. To choose a bus, click on the name in the Bus Available column and click Select. The bus name moves to the Bus Selected column

and the name appears at the left of the signal column. You can edit the signal list as necessary to match IP signal names to bus signals.



5. As shown below the tabbed input area of the main window presents a tab for each major element of the component definition file. Click on the tab to enter and edit subelement values for that element.



Some values are displayed in tables, as shown below. When entering values in these tables, remember to press Return after entering the value.

busType	master/s	interfaceId	baseAddr...
ambaAH...	master	dlxAHB	0

6. When you are through editing values for the component, select **File** > **Save** or **File** > **Save As** to save the information to an XML file. You need to add the *.xml* extension to the file name.

The pxedit application validates the saved XML file against the schema. If the file is not valid, an information message appears, pointing out any errors in the file.

Editing an Existing Component Definition File

To edit an existing component definition file, select **File > Open**. The pxedit application opens the file and displays the element data in the appropriate places. You can edit the values as described in the preceding section.

Note

pxedit cannot edit all elements of a component definition file, so some information may be lost when you open an existing file.

Packaging Components

Your components and component libraries must conform to a standard structure so that Platform Express can locate them. The overall structure of a component library is shown below.

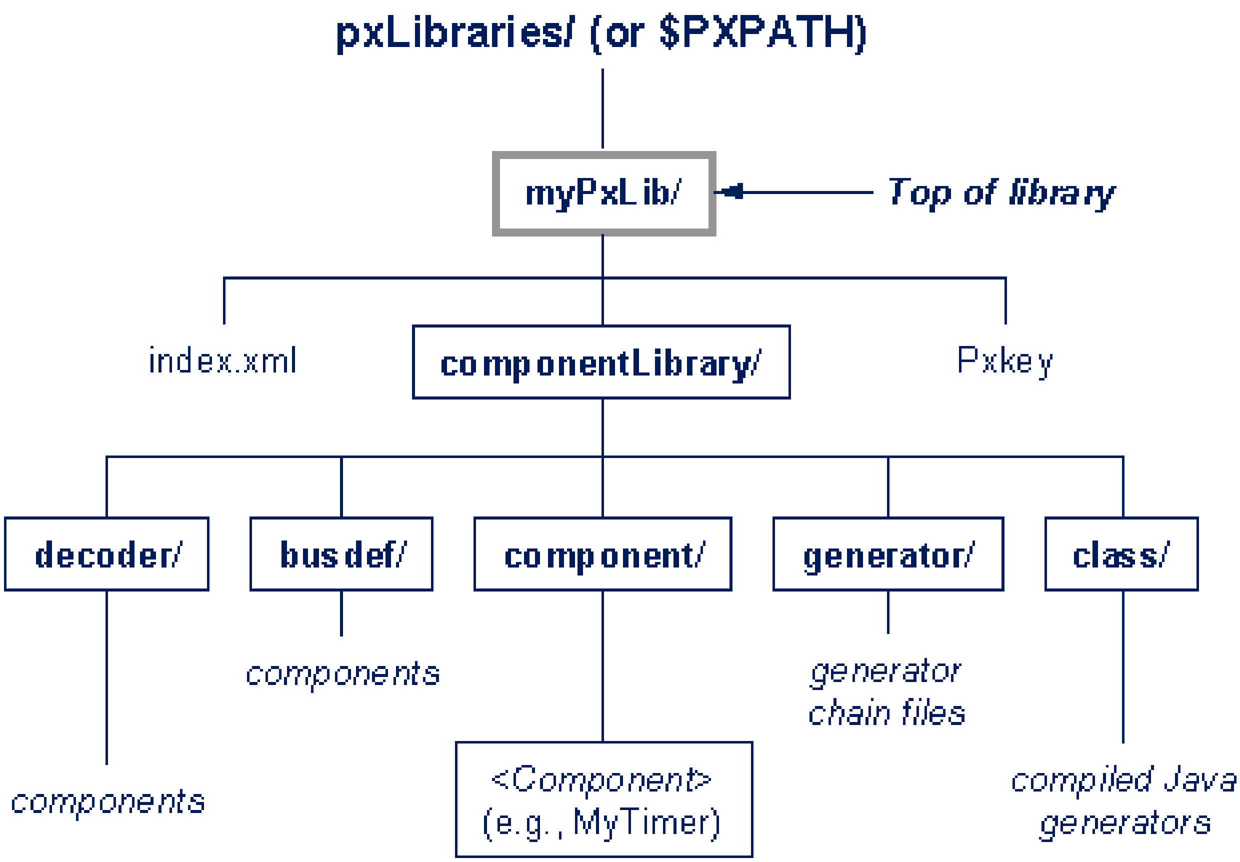
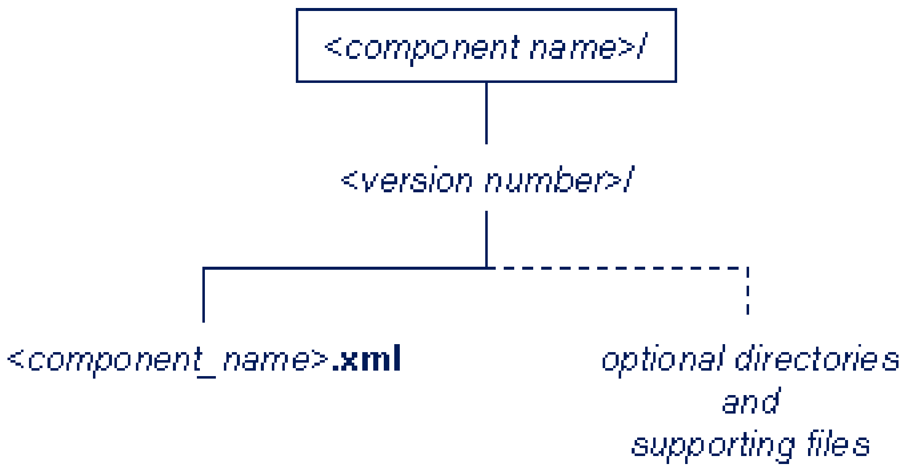


Figure 1-2. Component Library Structure

Platform Express locates component libraries through two different routes. The first, and primary, route is the PXPATH environment variable. The PXPATH variable is a colon-separated list of paths in which each path points to the top of a component library. If the PXPATH variable is not set, Platform Express looks in the current working directory for a *pxLibraries* directory, and then searches under that directory for component libraries. If Platform Express cannot find the *pxLibraries* directory in the current working directory, it looks for it in the parent directory of $PXHOME (at the same level as $PXHOME).

Within each library, Platform Express looks for components in their uniquely named directories under the *componentLibrary* directory. Each library must contain a *Pxkey* file, to license the library, and it may contain an *index.xml* file, which serves as an idex to the components in the library.

Shown below is the general structure for a component.



The minimum requirements for a component are a uniquely named directory with a *version number* directory containing a *component_name*.xml file. Most components will contain additional supporting files organized in subdirectories below the *version number* directory.

Running the mkIndex Utility

The mkIndex utility creates the file *index.xml* in the component library. This file is an index of all the components in the library. The utility also validates the component's *.xml* file against the schema.

Invoke the mkIndex utility as follows:

```
$PXHOME/tools/bin/mkIndex <path_to_component_library>
```

Correct errors and repeat this step until the *.xml* file is valid. The index file is not required to invoke Platform Express, but it can speed up the invocation. If the file does not exist, Platform Express creates an internal index of the library.

Note Be careful not to leave an out-of-date index in the component library. If the *index.xml* file is present, Platform Express will use it as-is, possibly ignoring any components added to the library.

Licensing a Library

To make a library available for Platform Express, it must contain a valid key file. A new key must be generated whenever anything within the *componentLibrary* subdirectory hierarchy is modified. You create a key file by running the Pxkeygen program. Three levels of licensing, each sold as a separate product, are available to Platform Express component developers:

- **Platform**, which licenses platform cores (CPUs plus core components).This level includes the following two.
- **IP**, which licenses peripheral components (memory, UARTs, timers, and so on). This level includes the following one.
- **EDA**, which licenses items such as generators and software that support components.

To use the Pxkeygen program, first set the PXHOME variable to point to the Platform Express installation. Also set either the MGLS_LICENSE_FILE or LM_LICENSE_FILE variable to point to the Platform Express license file. Invoke the Pxkeygen program as follows:

```
$PXHOME/tools/bin/Pxkeygen.sh [-h] [-e date] {pxLibrary1 ...}
```

where the -h option displays a help message, the -e *date* option specifies an expiration date for the library (or libraries), and {pxLibrary ...} is a list of paths to top-level directories of properly structured component libraries.

The program places a Pxkey file at the top level of the component library. The Pxkey file contains an expiration date and an encoded key.

Setting Up a Default Design

For platform cores, you can set up a default design configuration containing multiple components that will automatically be instantiated when the component is selected from the Component Library.

1. Prepare the platform (such as *a926_etm_tcm*, for example) and other components (RAM, ROM, and so on) for inclusion in libraries.

2. In Platform Express, create a design the has all the components instantiated.

3. Save the design. This results in a ".plx" file, such as *a926min.plx*, for the design that is saved in the design directory.

4. Move the .plx file to the library directory for the platform core. For example, move *a926min.plx* to *myPxLib/componentLibrary/component/a926_etm_tcm/1.0*.

5. Remove the saved design directory (from Step 3).

6. Edit the platform's .xml file (*myPxLib/componentLibrary/component/a926_etm_tcm/1.0/a926_etm_tcm .xml*) as follows.

   Near the end of the .xml file (after </cpu>) add the add the default design element:

```
 </cpu>

   <designFile> a926min </designFile>

</platformCore>
```

7. Run the Pxkeygen licensing utility, as described in "Licensing a Library," earlier in this chapter.

8. Reinvoke Platform Express, instantiate the platform, and verify that the default design is in place.

Chapter 2
User-Input Parameters and Configurators

Every component in any Platform Express component library is described by a component file, and every bus defined in a Platform Express library has a bus definition file. Component files and bus definition files are written using XML syntax. The meaning of each XML element is described in the XML schema, which is covered in “Creating the Component Definition File,” in Chapter 1

When a component or bus is instantiated in the design, an internal structure is created to represent the XML file. Multiple instantiations of the same library component or bus result in multiple structures. These structures can be modified by generators, or more commonly by user interaction managed by *configurators*. Configurators are discussed later in this chapter, in “Configurators.”

To preserve the integrity of the original XML file, only specially designated configurable elements can be modified. In the XML file, some elements, as directed by the schema, may be given a *resolve* attribute. The resolve attribute can take any of the following values: *immediate*, *user*, *dependent* or *generated*. These are described below.

- ***immediate* resolve**
  This is the same as if the element has no resolve attribute. The element is faithfully represented in the structure. If a generator or configurator attempts to modify the element through any standard or Platform Express API, an exception is thrown.

- ***user* resolve**
  This designates a configurable element (also called a property). Configurators and generators can modify the content of the element through either the PxProperty class or the ComponentDocument class. Textual

content can be replaced and child elements of arbitrary structure can be added. The attributes of a configurable element cannot be modified in any way, and any of its child elements from the original XML file cannot be removed. However, added child elements can be modified without restriction.

User resolved elements are persistent. Any modifications made to a user resolvable element are saved with the design and restored when the design is loaded. Elements with a *resolve=user* attribute must also have an *id* attribute set to an identifier that is unique within the XML file. The *id* attribute is used internally to support persistence and user configuration.

- ***dependent* resolve**
  The element's text value is dependent on the value of other elements. Typically the other elements will be user resolved elements. Elements with dependent resolve must have a *dependency* attribute which contains an expression used to evaluate the text value. The expression language is XPath with Platform Express extensions (see "Understanding XPath and Platform Express Extensions" in the Components chapter). The expression's context node is the dependent element itself. For the typical case where a dependent element evaluates to the setting of a user resolved element, you can use XPath's *id* function to access the user resolved property through its *id* attribute. For example, if the user resolved property has an id attribute of *baseAddress*, then the dependent property can assume the configurable property value by setting its dependency attribute to *id('baseAddress')*.

- ***generated* resolve**
  The element's value is typically written by generators. It can be modified just like a user resolved property, however the modifications are not persistent (they are not saved with the design). The id attribute is not required on elements of generated resolve.

Configurators

Configurators are Java classes that Platform Express executes to configure an object, usually through user interaction. For Platform Express to detect a configurator it has to be declared in the configurators section of an XML file, as described in “Creating the Component Definition File,” in Chapter 1.

Configurators are applied to various object types (components, busses, design settings, etc.). Platform Express provides several default configurators and declares them in PXHOME/etc/pxDefaultConfigurators.xml.

You can add new configurators and override default configurators by adding configurator elements to the configurators section of your component or bus definition file.

```
<configurator>
    <type>MyConfigurtor</type>
    <javaClass>MyConfiguratorJavaClass</javaClass>
    <presentation>
          <displayLabel>My Configurator</displayLabel>
    </presentation>
    <parameter name=”parameter1”>value1</parameter>
    <parameter name=”parameter2”>value2</parameter>
          .
          .
          .
 </configurator>
```

The *type* element can be used for component override of a default configurator of the same type. The *javaClass* element indicates the configurator class to be loaded and executed. The *displayLabel* element contains the text for a label that appears on a pop-up menu to invoke the configurator. The parameter elements supply name-value pairs to the configurator. The configurator class defines which, if any, parameters it uses.

The default configurators declared in pxDefaultConfigurators may be sufficient for most components. One of the default component configurators is declared as follows:

```
<configurator configureOnCreation="true">
    <type>Basic</type>
    <javaClass>com.mentor.px.configurator.DefaultConfigurator</javaClass>
    <presentation>
        <displayLabel>Basic</displayLabel>
        <displayLabel views="menu">Basics</displayLabel>
    </presentation>
    <parameter name="configGroup">requiredConfig</parameter>
</configurator>
```

This configurator has a *configureOnCreation* attribute to indicate that it is run when the component is first created. It invokes the DefaultConfigurator Java class, passing it a *configGroup* parameter with the value *requiredConfig*. The DefaultConfigurator class automatically builds a configuration dialog from information in the component XML file. The dialog presents input fields for all the component's configurable properties that have a *configGroups* attribute matching the passed in configGroup parameter value.

You can use the DefaultConfigurator class for more than just the basic configurator that is run on component instantiation. For example, you may want to use it on a secondary configurator that does not run on instantiation but does have a pop-up menu entry for explicit invocation. Or you may want to use it as a bus configurator. In either case, you can add the configurator declaration to your component or bus definition file, giving it the same Java class as the Basic configurator, but giving it a different configGroup parameter. Example:

```
<configurator>
    <type>MyDefault</type>
    <javaClass>com.mentor.px.configurator.DefaultConfigurator</javaClass>
    <presentation>
        <displayLabel>My Default</displayLabel>
    </presentation>
    <parameter name="configGroup">myGroup</parameter>
</configurator>
```

The above xml element will add a configurator to the component to configure the elements marked with *configGroup="**myGroup**"* with the default configurator rules.

Following sections describe how to use the default configurator and how to write and install your own configurators.

The Default Configurator

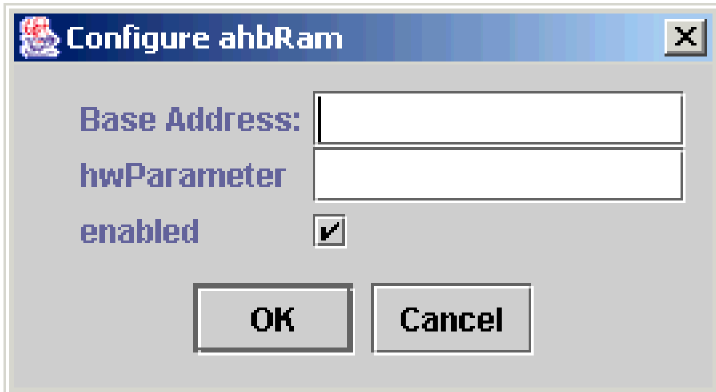
The default configurator supports general purpose configuration without having to write a configurator Java class. A configuration dialog is built automatically from information in the XML files. Its class name is *com.mentor.px.configurator.DefaultConfigurator* and it requires a *name* parameter. Unless overridden in the *configurators* section of their XML file, all components run the default configurator on component creation, passing in a name parameter of *requiredConfig*.

This section shows how to write a component file to take advantage of the default configurator that comes up when the component is added to the design.

Here is a partial listing of a memory component file with three user settable properties.

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
Copyright Mentor Graphics Corporation 2002
All Rights Reserved
-->
<component xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="schema/1.0/pxComponents.xsd">
    <name>ahbRam</name>
    <version>1.0</version>
   <busInterfaces>
        <busInterface interfaceId="ambaAHBLite">
            <busType>ambaAHBLite</busType>
            <slave>
                <memoryMap>
                    <addressBlock>
                        <baseAddress resolve="user”
                                     configGroups="requiredConfig"
                                     id="baseAddress"/>
                        <bitOffset>0</bitOffset>
                        <range resolve="dependent"
                               dependency="pow(2,id(’addrWidth’))"
                               id="addressRange"/>
                        <width>8</width>
 ...
        <hwParameters>
            <hwParameter name="addressSize" dataType="integer"
                         id="addrWidth" resolve="user"
                         configGroups="requiredConfig"/>
          </hwParameters>
 ...
        <fileSet fileSetId="fs-memtest">
 ...
            <swFunction>
                <fileRef>file-memtest</fileRef>
 ...
                <enabled id="diagsEnabled" resolve="user"
                         configGroups="requiredConfig">true</enabled>
            </swFunction>
        </fileSet>
...
```

Adding this component to the design causes the following dialog box to appear.



The default configurator found three configurable elements in the component XML file that belonged to the configuration group that matched its name parameter, *requiredConfig*. It created a form to allow input of all three values. The second and third input fields are labeled “hwParameter” and “enabled” which are the element names of the corresponding configurable elements. However, the first input field is labeled “Base address:” even though the element name is “baseAddress”.

The *memoryMap.xsd* file of the component schema contains an element description for the *baseAddress* element. It sets a default value for the prompt attribute with the following line:

```
<xs:attribute ref="prompt" default="Base Address:"/>
```

This shows that *baseAddress* elements have a *prompt* attribute. If none is specified in the XML file, then it takes “Base Address:” as the default value. The schema for the other two elements also allow a *prompt* attribute, but do not assign it a default value. Since no *prompt* attribute was supplied, the default configurator used the element name to prompt for the value.

Now notice that the *enabled* field is a check box instead of a text entry field like the other two fields. Its element definition found in *file.xsd* contains the line

```
<xs:attributeGroup ref="bool.prompt.att"/>
```

This refers to an attribute group defined in *common.xsd* which contains the following definition of the *format* attribute.

```
<xs:attribute name="format" type="formatType" default="bool">
```

Since this element has a *format* attribute set to “bool” by default, the configurator displayed a check box for obtaining the boolean value.

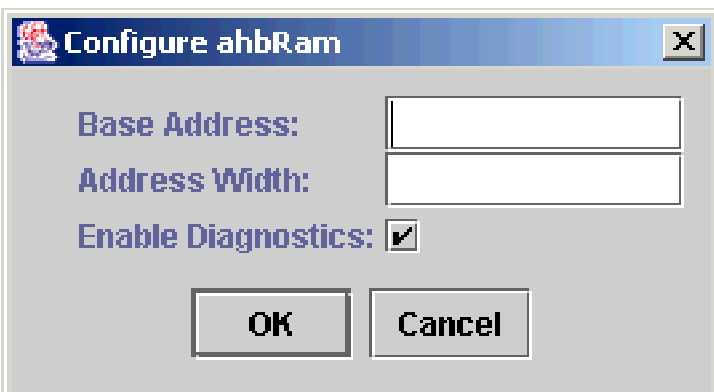
You can alter the displayed prompts by providing your own prompt attribute values in the component file. The following changes,

```
        <hwParameter name="addressSize" dataType="integer"
                     id="addrWidth" resolve="user"
                     configGroups="requiredConfig"
                     prompt="Address Width:"/>
```

and

```
<enabled id="diagsEnabled" resolve="user"
         configGroups="requiredConfig"
         prompt="Enable Diagnostics:">true</enabled>
```

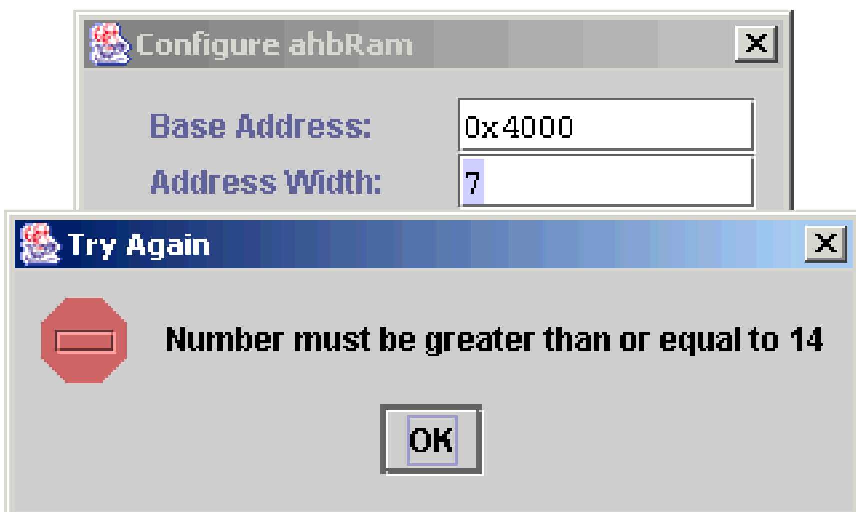
yield the following dialog.



Validators described in “Validators,” later in this chapter, allow you to check the validity of user input. However, you can use the XML component file to do some validity checking without having to write a validator. We have already seen the format attribute, which constrains the data type that may be entered. For numeric data, you can specify minimum and maximum values. For example, if the address width of this component must be between 14 and 20, you can constrain the input value with some additional attributes. Notice that the hwParameter element is not by default a numeric element, so to get minimum and maximum constraints to work you must explicitly give it a numeric format.

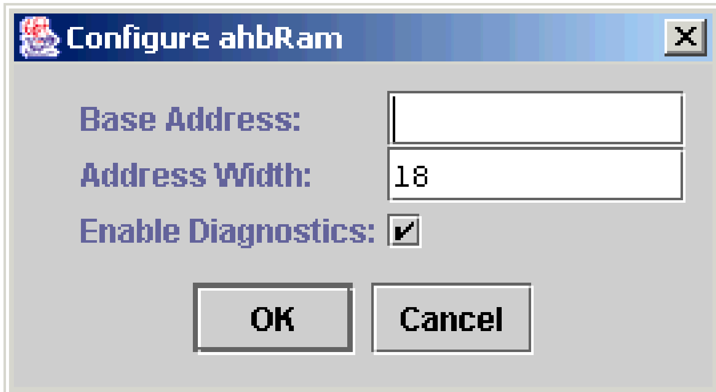
```
<hwParameter name="addressSize" dataType="integer" id="addrWidth"
             resolve="user" configGroups="requiredConfig"
             prompt="Address Width:"
             format="long" minimum="14" maximum="20"/>
```

Now if the user enters an invalid number, an error message appears when the form is submitted.



You may set the value of a user resolved element in the XML file. This provides users with a default value they can change if needed.

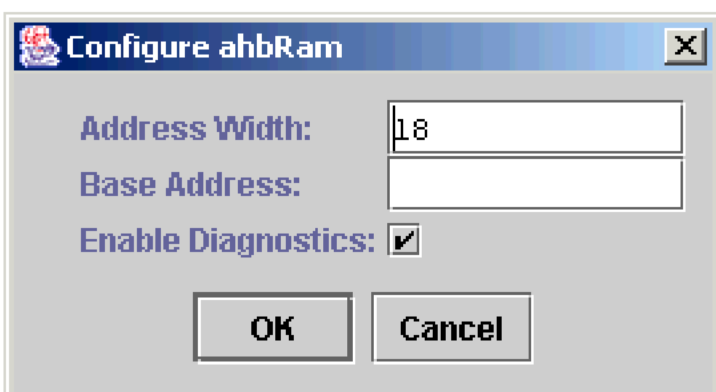
```
<hwParameter name="addressSize" dataType="integer" id="addrWidth"
             resolve="user" configGroups="requiredConfig"
             prompt="Address Width:"
             format="long" minimum="14" maximum="20">18</hwParameter>
```



You can control the order that the input fields appear in the form by using *order* attributes. Platform Express orders the fields according to ascending order attributes. It is tolerant of gaps and duplicates in the order number sequence.

As an example we will temporarily swap the first two fields of the dialog.

```
  <baseAddress resolve="user" configGroups="requiredConfig"
               id="baseAddress" order="2"/>
...
  <hwParameter name="addressSize" dataType="integer" id="addrWidth"
               resolve="user" configGroups="requiredConfig"
               prompt="Address Width:"
               format="long" minimum="14" maximum="20"
               order="1">18</hwParameter>
```



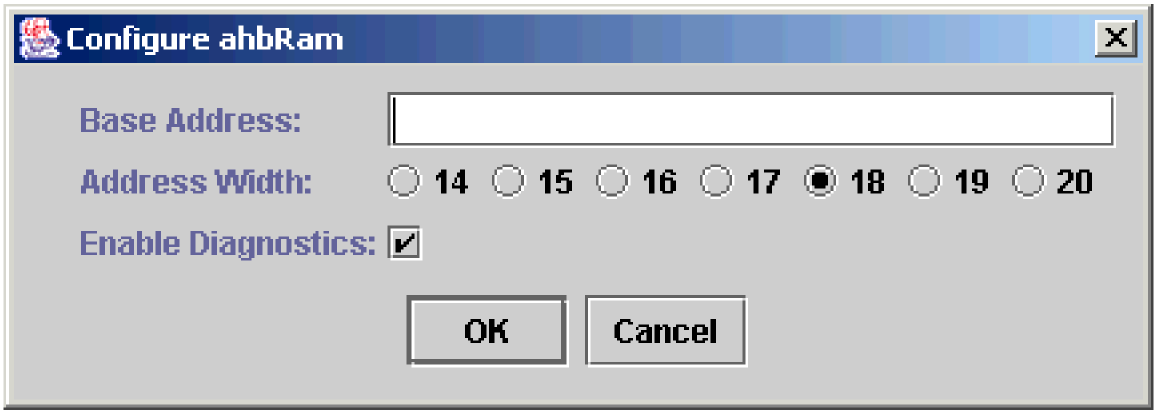
Note that we did not include an *order* attribute on *enabled* so it was placed after all the ordered fields.

Since there is a limited range of allowed values for the address width, it might be simpler for the user to choose a value rather than type one in. This can be accomplished in the component XML by adding a *ui* element to the component if it doesn't already exist, then adding a *uiChoice* element as a child of *ui*. Each *uiChoice* element has an *id* attribute so it can be referenced by properties with enumerated values. In this example, we add a *uiChoice* element that inclusively enumerates all numbers between the minimum and maximum allowed address widths.

```
...
<ui>
  <uiChoice id="widthOptions">
    <uiChoiceElement>14</uiChoiceElement>
    <uiChoiceElement>15</uiChoiceElement>
    <uiChoiceElement>16</uiChoiceElement>
    <uiChoiceElement>17</uiChoiceElement>
    <uiChoiceElement>18</uiChoiceElement>
    <uiChoiceElement>19</uiChoiceElement>
    <uiChoiceElement>20</uiChoiceElement>
  </uiChoice>
</ui>
```

The address width element's *format* attribute needs to change from its default value of *long* to *choice*. A *choiceRef* attribute must be added to refer to the *uiChoice* element listed above. Also note that the *minimum* and *maximum* attributes are no longer necessary.

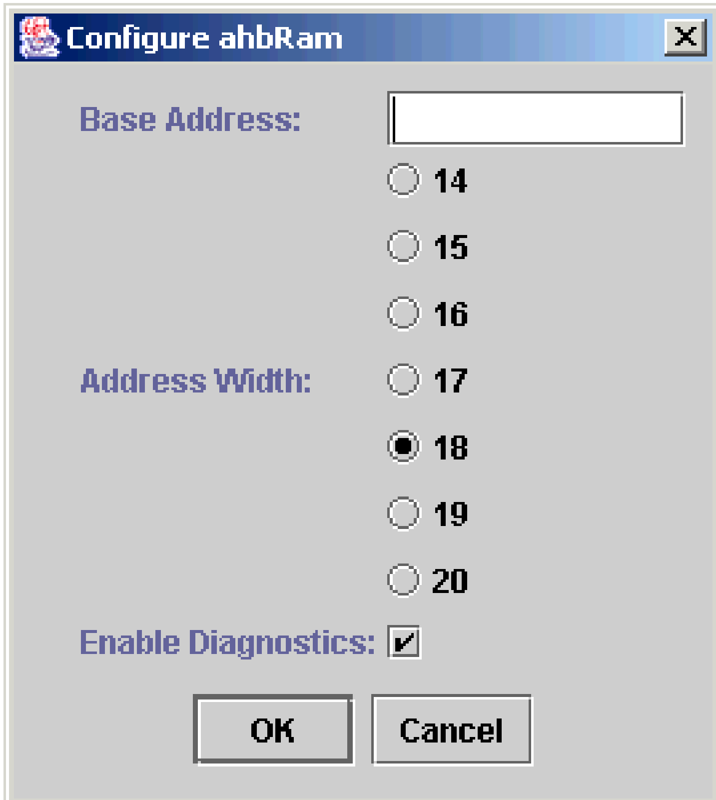
```
<hwParameter name="addressSize" dataType="integer" id="addrWidth"
             resolve="user" configGroups="requiredConfig"
             prompt="Address Width:"
             format="choice" choiceRef="widthOptions">18</hwParameter>
```



The default width value of 18 is preselected.

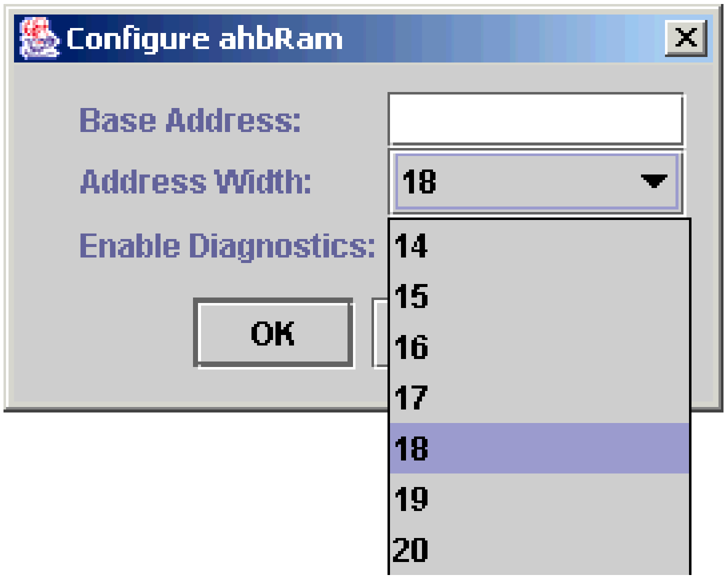
The display for a choice element is controlled by two attributes. The *choiceStyle* attribute has a default value of *radio* which causes radio buttons to be used. The *direction* attribute has a default value of *horizontal* which causes radio buttons to be laid out horizontally. You can change the layout direction by explicitly setting the *direction* attribute.

```
<hwParameter name="addressSize" dataType="integer" id="addrWidth"
             resolve="user" configGroups="requiredConfig"
             prompt="Address Width:" format="choice" direction="vertical"
             choiceRef="widthOptions">18</hwParameter>
```



You can change the choice style to a drop-down combo box.

```
<hwParameter name="addressSize" dataType="integer" id="addrWidth"
             resolve="user" configGroups="requiredConfig"
             prompt="Address Width:" format="choice" choiceStyle="combo"
             choiceRef="widthOptions">18</hwParameter>
```

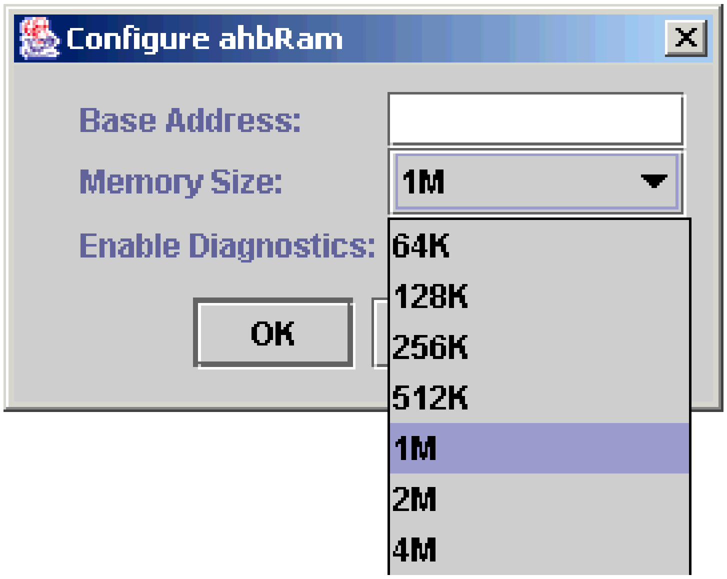


You can provide display text for radio or combo entries that differ from the corresponding element values. This is done by placing a *text* attribute in the *uiChoiceElement* elements. For example, instead of showing relatively meaningless bit counts in the combo box, you can display the memory size they translate to.

```
<uiChoice id="widthOptions">
  <uiChoiceElement text="64K">14</uiChoiceElement>
  <uiChoiceElement text="128K">15</uiChoiceElement>
  <uiChoiceElement text="256K">16</uiChoiceElement>
  <uiChoiceElement text="512K">17</uiChoiceElement>
  <uiChoiceElement text="1M">18</uiChoiceElement>
  <uiChoiceElement text="2M">19</uiChoiceElement>
  <uiChoiceElement text="4M">20</uiChoiceElement>
</uiChoice>
```

Since the meaning of the displayed values has changed, it is a good idea to change the label too.

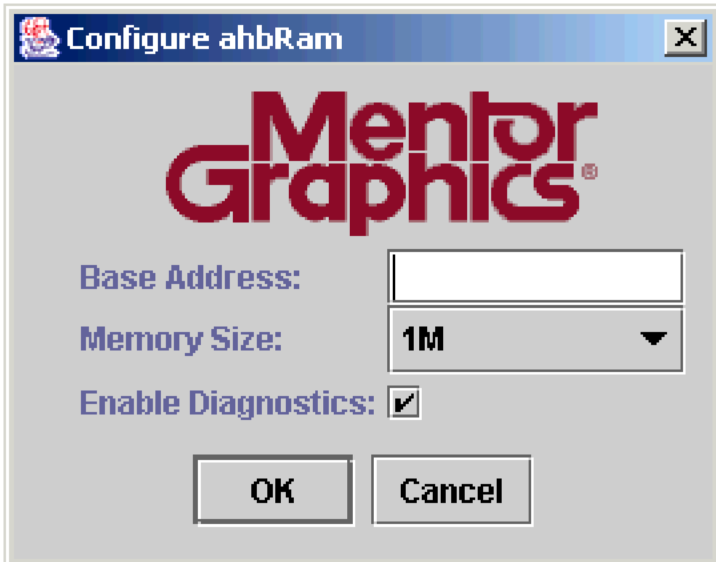
```
<hwParameter name="addressSize" dataType="integer" id="addrWidth"
             resolve="user" configGroups="requiredConfig"
             prompt="Memory Size:" format="choice" choiceStyle="combo"
             choiceRef="widthOptions">18</hwParameter>
```



Note that since 18 is the default value, the corresponding text of *1M* is initially selected.

You can add a decorative image or company logo to the form by placing a *uiIcon* element inside the *ui* element. The *uiIcon* element contains the pathname relative to the component directory, the component library or PXHOME of a gif or jpeg image file. In order to be picked up by the configurator, its configGroups attribute must contain the configurator's *name* parameter value.

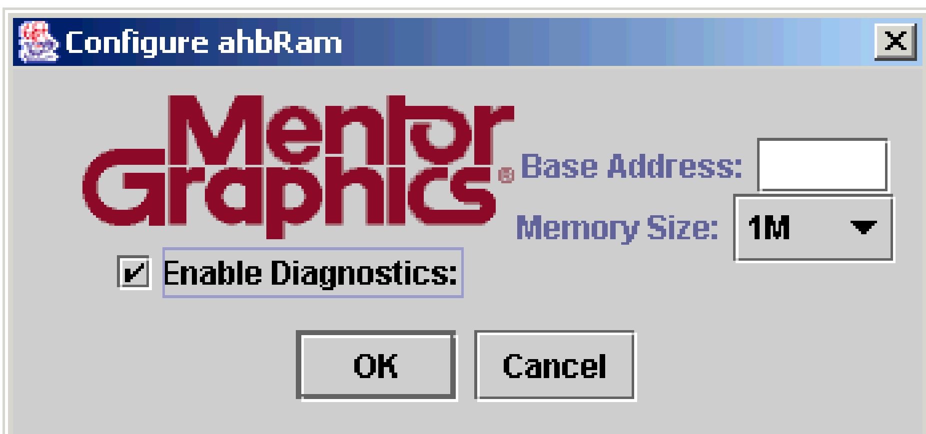
```
 <ui>
   <uiIcon configGroups=”requiredConfig>images/MGlogo_rg_sm.gif</uiIcon>
   <uiChoice id="widthOptions">
...
```



By default Platform Express lays out all elements of a form in a single column. You can control the layout with a *uiForm* element inside the *ui* element. The *uiForm* element must contain a *configGroup* attribute which equals the configurator's *name* parameter value. Configurable elements referenced in a *uiForm* do not require their own *configGroups* attribute. *UiForm* allows you to create any number of nested rows, columns and grids to hold the input fields. It also allows additional text and icons to be placed at any location within the form.

For example, you could lay out the above form into a row of two columns, the first column containing the logo and the check box, and the second column containing the other two fields. Use the *uiRow* element to define a row. This can contain nested *uiColumn* elements. The input fields are selected with the *uiProp* elements. These have *propId* attributes which refer to the user resolved properties.

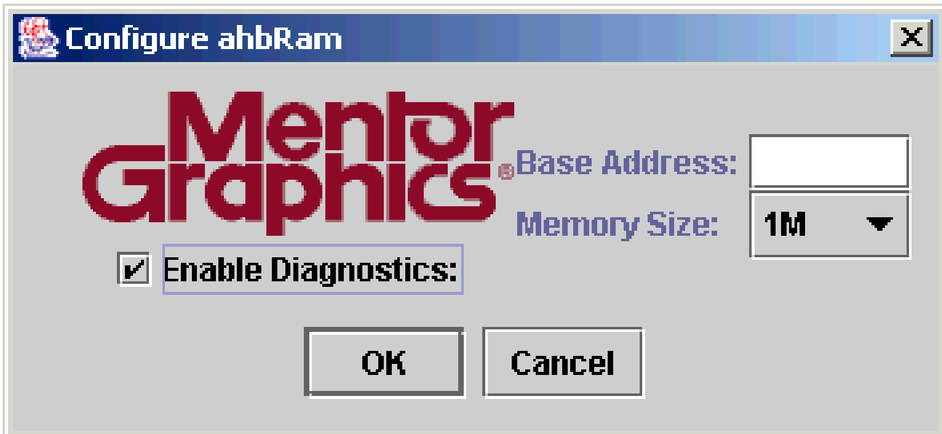
```
    <ui>
        <uiForm configGroup="requiredConfig">
            <uiRow>
                <uiColumn>
                    <uiIcon>images/MGlogo_rg_sm.gif</uiIcon>
                    <uiProp propId="diagsEnabled"/>
                </uiColumn>
                <uiColumn>
                    <uiProp propId="baseAddress"/>
                    <uiProp propId="addrWidth"/>
                </uiColumn>
            </uiRow>
        </uiForm>
        <uiChoice id="widthOptions">
...
```



Explicitly laid out forms usually require some tweaking to improve their appearance.

The first improvement would be better alignment of the base address and memory size. This can be accomplished using a grid. A grid contains either rows or columns of objects which are aligned with objects in the same position in adjacent rows or columns. Platform Express provides a shortcut such that a *uiProp* directly under a grid is considered to be a row consisting of one label on the left and one entry field on the right. We can try this by replacing the *uiColumn* element that contains the two *uiProp* entries with a *uiGrid*.

```
<uiGrid>
    <uiProp propId="baseAddress"/>
    <uiProp propId="addrWidth"/>
</uiGrid>
```

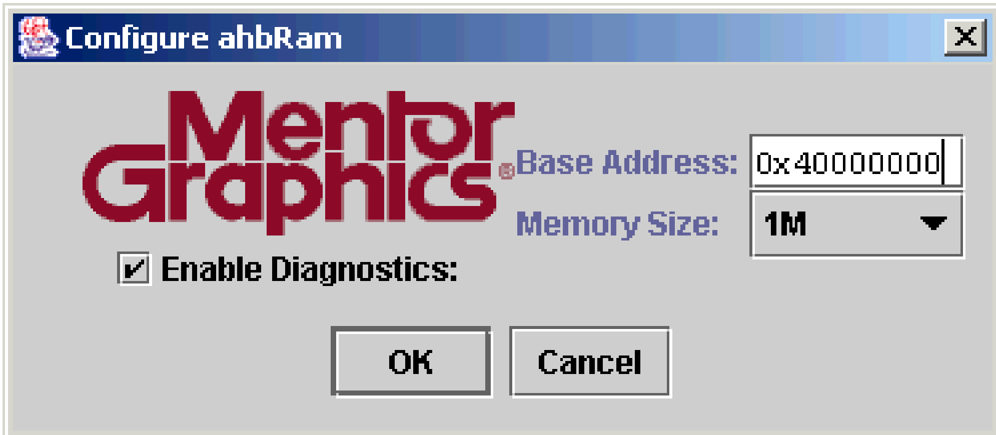


This is a slight improvement, but the entry field of the base address appears to be too narrow to accommodate a 32 bit address expressed in hexadecimal. This wasn't a problem when there was a single column because the minimum form width was wide enough for the entry to expand to a usable width.

Platform Express forms use the GridBag layout from the AWT package of the Java Runtime Environment. Most of the GridBag constraints can be used as attributes of the ui elements in a form. See http://java.sun.com/docs/books/tutorial/uiswing/layout/gridbagConstraints.html.

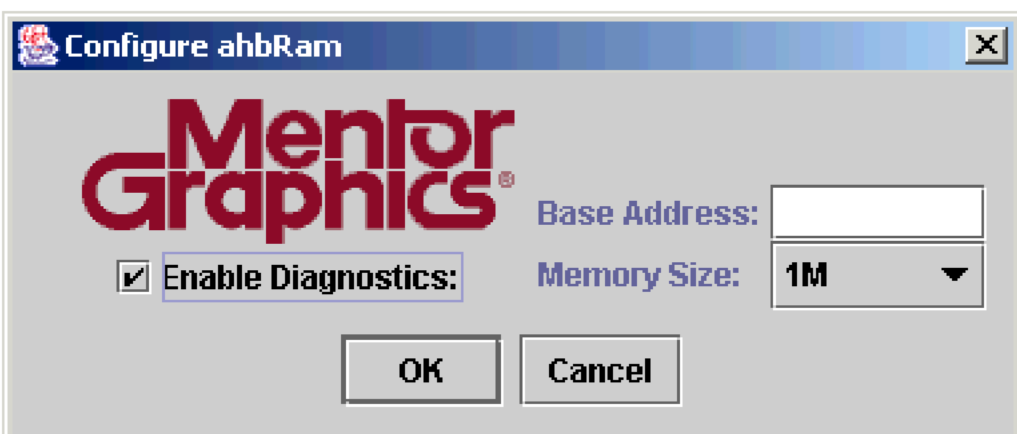
The *ipadx* constraint is used to increase the minimum width of an element. If we were to place this constraint on the affected *uiProp*, the whole row widens including the space used by the label. We instead place it on its container *uiGrid*. The contained *uiProp*s expand as they are designed to, with only the entry field filling out the remainder.

```
<uiGrid ipadx="20">
    <uiProp propId="baseAddress"/>
    <uiProp propId="addrWidth"/>
</uiGrid>
```



Some additional GridBag constraints can be placed on uiGrid to improve its placement in the form. In the above display the two entries are uncomfortably close to the logo. This can be remedied using an inset on the left side of the grid to move it away from the logo. They also look somewhat unsettled being vertically centered in their column. This can be improved by anchoring the grid to the bottom of the column so that it rests on the same level as "Enable Diagnostics".

```
<uiGrid ipadx="20" insetLeft="8" anchor="south">
    <uiProp propId="baseAddress"/>
    <uiProp propId="addrWidth"/>
</uiGrid>
```

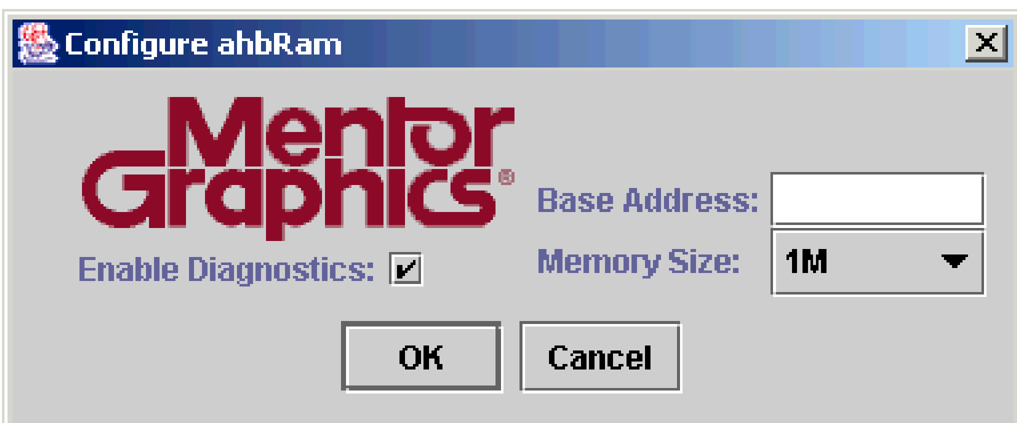


Now notice that "Enable diagnostics" appears much different than it did in the default layout. Platform Express uses components from the Swing package of the

Java Foundation Classes. Swing's check box component has its own text field. When a uiProp is used by itself to reference boolean user input, the default configurator puts the prompting text in the text field of the check box component. However when the configurator generates its own layout, the prompt is displayed in a label which is placed to the left of a text-less check box.

The *uiPropText* and *uiPropEntry* elements allow separate reference to a user defined property's prompt label and its prompt-less entry field. Both require a *propId* attribute to reference the user resolvable property. The following replacement for the "diagsEnabled" *uiProp*, lays out the entry as it appeared in the default form layout. Note that it includes a right inset on the label to separate it from the check box. It also includes an anchor on the row to left-align the entry.

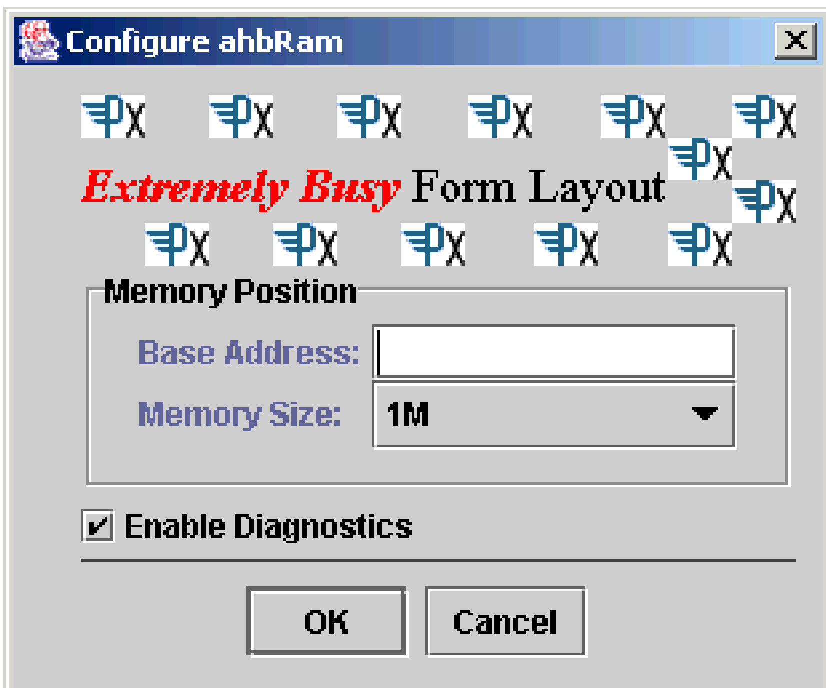
```
<uiColumn>
    <uiIcon>images/MGlogo_rg_sm.gif</uiIcon>
    <uiRow anchor="west">
        <uiPropText insetRight="5" propId="diagsEnabled"/>
        <uiPropEntry propId="diagsEnabled"/>
    </uiRow>
</uiColumn>
```



Since in this case we built the "Enable Diagnostics" entry to look like the default layout preferred by Platform Express, you can achieve the same result by placing the *uiProp* entry inside a *uiGrid* as we did with the base address and memory size.

```
<uiColumn>
    <uiIcon>images/MGlogo_rg_sm.gif</uiIcon>
    <uiGrid anchor="west">
        <uiProp propId="diagsEnabled"/>
    </uiGrid>
</uiColumn>
```

This has been a simple example with only three user inputs. More complex components may require more elaborate form layouts. You may want to group information into enclosing boxes and you may need to add explanatory text. You may also want to exploit the grid feature for more than just aligning labels and entries. The following form illustrates these features by adding unnecessary icons and text to the same three-input example.



The first element inside this *uiForm* contributes the bottom most visible feature. The black line just above the “OK” “Cancel” buttons is actually the visible part of a border that surrounds the body of the form. The *uiBorder* element supports most of the features of the Swing package’s Border Factory. See http://java.sun.com/docs/books/tutorial/uiswing/misc/border.html. The resulting border encloses every component represented by *uiBorder*’s child elements. By making *uiBorder* the parent of the entire *uiForm* content, the border encloses everything in the form except “OK” and “Cancel”.

```
        <uiForm configGroup="requiredConfig">
            <uiBorder type="line" bottom="1">
                <uiColumn>
...
                </uiColumn>
            </uiBorder>
        </uiForm>
```

The *type* attribute indicates that a line border is to be used. Line borders consist of a solid line of a specified width. The width of each side of the border is specified separately. The default width is zero, so the border is invisible by default. By setting the *bottom* attribute to “1”, we draw a one-pixel high line across the bottom of the form content.

To illustrate some of the additional capabilities of the *uiGrid*, a checkerboard of icons has been drawn. The icon is the same one used for the Platform Express file browser. Its image file is installed in the *images* subdirectory of PXHOME.

The grid contains rows which each contain alternating icons and empty cells. The empty cells are designated by *uiEmpty* elements.

```
    <uiForm configGroup="requiredConfig">
            <uiBorder type="line" bottom="1">
                <uiColumn>
                    <uiGrid>
                        <uiRow insetBottom="2">
                            <uiIcon>images/pxicon.gif</uiIcon>
                            <uiEmpty/>
                            <uiIcon>images/pxicon.gif</uiIcon>
                            <uiEmpty/>
                            <uiIcon>images/pxicon.gif</uiIcon>
                            <uiEmpty/>
                            <uiIcon>images/pxicon.gif</uiIcon>
                            <uiEmpty/>
                            <uiIcon>images/pxicon.gif</uiIcon>
                            <uiEmpty/>
                            <uiIcon>images/pxicon.gif</uiIcon>
                        </uiRow>
```

```
                        <uiRow insetBottom="2">
                            <uiText gridwidth="9"
                                    gridheight="2"
                                   anchor="west"><![CDATA[<html><font size=+1
color="black"><font color="red"><I><B>Extremely Busy</B></I></font> Form
Layout</font>]]></uiText>
                            <uiIcon>images/pxicon.gif</uiIcon>
                            <uiEmpty/>
                        </uiRow>
                        <uiRow insetBottom="2">
                            <uiEmpty/>
                            <uiIcon>images/pxicon.gif</uiIcon>
                        </uiRow>
                        <uiRow insetBottom="24">
                            <uiEmpty/>
                            <uiIcon>images/pxicon.gif</uiIcon>
                            <uiEmpty/>
                            <uiIcon>images/pxicon.gif</uiIcon>
                            <uiEmpty/>
                            <uiIcon>images/pxicon.gif</uiIcon>
                            <uiEmpty/>
                            <uiIcon>images/pxicon.gif</uiIcon>
                            <uiEmpty/>
                            <uiIcon>images/pxicon.gif</uiIcon>
                            <uiEmpty/>
                        </uiRow>
                    </uiGrid>
```

The checkerboard pattern is broken up with text which spans multiple rows and columns. The *uiText* element inserts the text label. Its GridBag constraints of *gridwidth* and *gridheight* allow it to span multiple grid cells.

The text element is created from a Swing label. Swing labels accept simple text strings, but they can also accept HTML. HTML text is designated by a leading *<html>* tag in the text string. In this example, an XML *CDATA* section is used so that the XML parser will not try to interpret HTML markup as XML markup. Instead, everything between *<![CDATA[* and *]]>* is passed to the Swing label which then interprets the HTML tags.

Below the checkerboard are the base address and memory size entries, grouped into a box labeled “Memory Position”. The box is created by two nested borders. The outer border is the one you see. It is an etched border with the “Memory Position” title. The inner border is an empty border used to insert space around its contents. Like the line border, the thickness of the empty border is individually specified for each side.

```
<uiBorder type="etched" title="Memory Position"
          titleColor="black">
    <uiBorder type="empty"
              left="16" right="16" bottom="10">
        <uiGrid fill="horizontal">
            <uiProp propId="baseAddress"/>
            <uiProp propId="addrWidth"/>
        </uiGrid>
    </uiBorder>
</uiBorder>
```

Note that the *uiGrid* that contains the two *uiProp*s has a GridBag *fill* constraint set to *horizontal*. This allows the width of the two entries to expand to fill the space below the checkerboard.

The final part of the layout places the “Enable Diagnostics” check box.

```
                <uiProp anchor="west" propId="diagsEnabled"/>
            </uiColumn>
        </uiBorder>
    </uiForm>
```

It intentionally displays the prompt as the check box text rather than as an adjacent label. However, the trailing colon was removed. This required a change in the *prompt* attribute of the *enable* element.

```
<enabled id="diagsEnabled" resolve="user" configGroups="requiredConfig"
         prompt="Enable Diagnostics">true</enabled>
```

Writing a Configurator Java Class

As we have seen previously, many configuration tasks can be performed without writing a single line of code. However, there are times when the need for a more customized configurator arises, such as when the configuration task is too complex for the default configurator to handle. The configurator might contain multiple panels (as, for example, in a required driver configurator for the buses), or the configurator may be a legacy Java class, or even an external program.

All configurators must have a Java class that implements the PxConfigurator interface, or extend a class that does. Platform Express searches the following paths in sequence to locate the configurator class: The class directory under the component location, the class directory under the component library, and finally the paths used by the Platform Express application.

The PxConfigurator interface is initialized passing in a Configurator object. This object allows access to some of the following data.

1. **Owner.**
   The Owner has the following characteristics:

   - It is the configurable object that has properties that need user configuration.
   - It has a Document Object Model (DOM) XML structure wrapped in a PxProperty tree.
   - It provides getter methods to retreive important information for the configurator, such as the root of the PxProperty tree.

2. **Validators.**
   A validator validates the result of a configuration. (See “Validators,” later in this chapter.) It is ancillary to any validation performed implicitly by the configurator, and it is not essential that the configurator run the validation. If it never accesses the validators, the Configurator object will perform the validation after the configuration is complete. However, if the configurator chooses to execute the validators, it can assist the user in correcting configuration errors detected by the validators while it is still active.

3. **Configuration Activity.**
   Configuration Activity is an action that needs to run after the the initial configuration (such as connecting a new component to the design). This is passed in by the Platform Express application. The activity runs before any validation so that the result of the activity can be validated. It is not essential that the configurator perform this activity. If it never accesses the activity, the Configurator object will perform it after the configuration is complete. However, if the configurator performs the activity followed by the validation, it can assist the user in correcting configuration errors detected by the validators while it is still active.

Platform Express provides convenience classes to support three different ways of implementing the PxConfigurator interface and addressing the three needs mentioned at the begining of this section:

1. Single Panel Configurators.
2. Legacy Configurators.
3. MultiPanel Configurators.

Single Panel Configurators

Single Panel Configurators are the most common configurators. They provide an easy minimum-coding-required way to perform both simple and complex configuration tasks that require only one GUI panel.

Minimum Implementation for Single Panel Configurators

The minimum implementation of the PxSingleConfigurator is as follow:

```
import com.mentor.px.api.*;

public class MinimumImplementationConfigurator extends PxSingleConfigurator
{

    /** Creates new MinimumImplementationConfigurator */
    public MinimumImplementationConfigurator() {
    }

    /**
       * Creates the content of the panel. This method gets called when
     * the panel is first created, so any initialization code should
     * go here.
     *
    * @throws  PxConfiguratorException if any initialization errors
     *         occurr.
     */
    protected void initializeComponents() throws PxConfiguratorException {
    }

    /**
     * Saves the values entered in this panel by the user to
```

```
      * the owner's properties.
      */
     public void savePanel() {
     }

     /**
      * Fills the value fields of the panel from the owner properties.
      */
     public void fillPanel() {
     }
 }
```

The constructor is an empty one and shouldn't contain any code. (It can contain some initialization code, but the initialization code should go in **initializeComponents()** along with the GUI creation code.) This method is called only once, when the configurator is first created.

The **fillPanel()** method is called whenever the the configurator is invoked; it should contain the code to fill the GUI elements from the owner's DOM.

The **savePanel()** method is called whenever the user presses Ok in the configuration dialog, and *before* any action or validator is invoked. It should contain the code to update the owner's DOM with the values entered by the user in the configuration dialog.

The following example demonstrates how to display a configurator dialog using PxSingleConfigurator. The example shows how to display a value from the owner's DOM in a text field:

```
 /* Import needed classes */
 import com.mentor.px.api.*;
 import com.mentor.px.design.PxProperty;
 import javax.swing.*;

 /* extends PxSingleConfigurator */
public class MyMinimumImplementationConfigurator extends PxSingleConfigurator
{
 /* Declare Variables */
     JTextField text;
          .
          .
          .

     /** Creates new MyMinimumImplementationConfigurator */
     public MyMinimumImplementationConfigurator() {
     }
```

```
/**
 * Creates the content of the panel. This method gets called when
 * the panel is first created, so any initialization code should
 * go here.
 *
 * @throws  PxConfiguratorException if any initialization errors
 *         occurr.
 */
protected void initializeComponents() throws PxConfiguratorException {
    /* Set the dialog label */
    this.setPanelLabel(" My Minimum Implementation ");

    /* Initialize variables */
    this.text = new JTextField();
     .
     .
     .

    /* Add GUI elements to the configurator
     * Note: PxSingleConfigurator extends JPanel so we use all the
     * Swing code to add GUI components to the configurator.
     */
    this.add(text);
     .
     .
     .

}

/**
 * Saves the values entered in this panel by the user to
 * the owner's properties.
 */
public void savePanel() {
    PxProperty root = this.getOwner().getConfigurableRoot();

    /* Saves the user entered values in the owner's DOM */
    String value = this.text.getText();
    root.getChildProperty("myProp").setValue(value);
     .
     .
     .

}

/**
 * Fills the value fields of the panel from the owner properties.
 */
public void fillPanel() {
    PxProperty root = this.getOwner().getConfigurableRoot();
```

```
        /* Fill the GUI components with values from the owner's DOM */
        String value = root.getChildValue("myProp");
        this.text.setText(value);
         .
         .
         .

    }
}
```

Optional Methods for Single Panel Configurators

Other methods can be implemented to further control the behavior of the configurator, such as:

- `public boolean` **`isConfigurable()`**
  Determines whether the configurator has any data to configure. It is used to gray out the unconfigurable configurators. The default implementation to this method returns *true*.

- `public boolean` **`requiresConfig()`**
  Determines whether this configurator is required for proper building of the design (or the proper operation of any generator in general). This method must return true if the configurator misses some data important for the build, or requires user confirmation during building. The default implementation ensures that the configurator is shown to the user at build time for the first design build, or when the configurator has missing data. (Note: Optional configurators must override this method to return *false*.)

Legacy Configurators

Legacy configurators, which do not follow the PxConfigurator GUI scheme, can be easily incorporated using **PxAbstractConfigurator**. This class provides a wrapper around the legacy configurator.

PxAbstractConfigurator provides default implementations for all the methods of the *PxConfigurator* interface except **configure()**. It also provides utility methods the derived configurator may call.

To implement a class that extends PxAbstractConfigurator, only one method, **configure()**, needs to be implemented, although other methods such as initialize(Configurator), and requiresConfig() will be overriden in most typical implementations.

Here is a sample implementation of a legacy configurator that extends PxAbstractConfigurator:

```
import com.mentor.px.api.*;

public class MyLegacyConfigurator extends PxAbstractConfigurator {

    /** Creates new MyLegacyConfigurator */
    public MyLegacyConfigurator() {
    }

    /**
     * Initializes the configurator.
     *
     * @param   config      The <code>Configurator</code> reference to
     *                      this configurator.
     *
     * @throws  PxConfiguratorException if errors are encountered in
     *          any of the configurator’s initial data.
     */
    public void initialize(com.mentor.px.configurator.Configurator config) {
        /* Must call super for proper initialization */
        super.initialize(config);

        /* Put Legacy Initialization Code here */
            .
            .
            .


  }

  /**
   * This method configures the <code>Configurable</code>.  It
   * should perform any necessary user interaction, set the
   * configurable properties as directed by the user, and return
   * <code>true</code>.  If the user cancels the configuration it
   * should return <code>false</code>.
   * <p>
   * In the event of a cancellation or exception, Px will
   * automatically restore any properties set by the confiugrator to
   * their value before this method was called.  The only exception
   * is if the configurator sets properties between calls to
   * <code>DesignModel.editor.startConfigure()</code> and
```

```
     * <code>DesignModel.editor.endConfigure()</code>.
     *
     * @return  <code>true</code> if the configuration was completed,
     *         <code>false</code> if it was cancelled.
     *
     * @throws  PxConfiguratorException if any error was encountered
     *         during the configuration.
     *
     * @see com.mentor.px.gmodel.ModelEditor#startConfigure
     * @see com.mentor.px.gmodel.ModelEditor#endConfigure
     * @see com.mentor.px.gmodel.DesignModel#editor
     */
    public boolean configure() throws PxConfiguratorException {
        /* Put Legacy Configuration code here */
            .
            .
            .
    }

    /**
     * Indicates whether or not elements managed by this
     * configurator require configuration.  This method may be called
     * by generators to determine if they need to run a configurator
     * before they can proceed.
     * <p>
     * The default implementation always returns <code>false</code>.
     *
     * @return  <code>false</code> if all elements have been
     *          configured and validated to the satisfaction of this
     *          configurator or if the default values are known to be
     *          sufficient.  <code>true</code> if user interaction is
     *          required.
     */
    public boolean requiresConfig() {
        /* Note : The default implementation of this method in
         * PxAbstractConfigurator returns false, so If this configurator is
         * required for proper execution of generators, this method should
         * be overriden to implement its required function.
         */
    }
}
```

MultiPanel Configurators

A MultiPanel Configurator is a general scheme for writing configurators in Px. You could perform the configuration tasks described in the preceding sections with the MultiPanelConfigurator API, but that requires nearly double the effort.

Obviously, the MultiPanel API is suited to creating configurators that require multiple panels. Consider, for example, the task of configuring special pins in a component for which the user is allowed to specify special code to be used with those pins (such as HDL code), where each pin needs a separate panel. If you are sure that all components to which the configurator applies contain only one special pin, then the PxSingleConfigurator API is more appropriate. However, if some components might contain multiple pins, then a MultiPanel configurator is needed.

The task of writing a MultiPanel configurator involves three classes:

1. The Configurator (**PxMultiConfigurator**).
2. The Panel (**PxConfigurationPanel**).
3. The Validator (**PxConfigurationValidator**)

Note The validator class is not required, as the panel implements the validator interface, but it is used to share a single validator implementation among multiple panels. This validator class which applys only to multi-panel configurators is not to be confused with the general PxValidator interface described later in this chapter, in "Validators."

The Configurator

The Configurator class is responsible for:

- Calculating and setting the number of panels required.
- Collecting the properties for configuration from the owner's DOM.
- Creating the panels and validators required.

Accordingly, the PxMultiConfigurator has four methods that all the derived classes must implement; this will be illustrated in the following example:

```
import com.mentor.px.api.*;

public class MyMultiPanelConfigurator extends PxMultiConfigurator {
```

```
/** Creates new MyMultiPanelConfigurator */
public MyMultiPanelConfigurator() {
}


/**
 * Returns the number of panels that this configurator uses.
 * For most cases this will return 1, as most configurators will not use
 * more than one panel.
 *
 * @return
 *       an integer representing the number of panels that this
 *       configurator uses. This number <B>Can't</B> be 0.
 */
protected int getNumberOfPanels() {
    /* Return a fixed number, or calculate the required number according
     * to the owner's status.
     */
        .
        .
        .
      return number;
}

/**
 * Returns the configurable properties specific to the passed panel
 * index.
 * <P>
 * A possible implementation for this method, is a switch-case statement
 * that switches on the panelIndex to select elements specific for each
 * panel from the owner's DOM Tree.
 *
 * @param panelIndex
 *       The panel index that we want to get properties for.
 *       This must be >= 0 and < number of panels of this configurator.
 *
 * @return
 *       The properties specific to the passed panel index.
 *
 * @throws PxConfiguratorException
 *       If any error occurs while getting the data from the owner.
 */
protected PxProperty[] getConfigurableProperties(int panelIndex) throws
  PxConfiguratorException {
    /* collect/return the properties required for each panel.
     * Properties might be collected and stored internally in the
     * getNumberOfPanels() method, and all is needed here is to return
     * the properties specific for each panel.
     */
        .
```

```
        .
        .
}

/**
 * Returns the configuration panel object specific to the passed panel
 * index.
 * <P>
* A possible implementation for this method, is a switch-case statement
 * that switches on the panelIndex to create the specific panel of that
 * index.
 *
 * @param panelIndex
 *       The panel index that we want to create a panel for.
 *       This must be >= 0 and < number of panels of this configurator.
 *
 * @return
 *       The PxConfigurationPanel specific for the passed panel index
 *
 * @throws ClassNotFoundException
 *       If the panel class was not found at runtime.
 *       The configurator author shouldn't worry himself with handling
 *       the exception unless he wants to override the default handling
 *       mechanism, or he wants to specify a special message.
 */
protected PxConfigurationPanel getConfigurationPanel(int panelIndex)
  throws ClassNotFoundException {
  /* creates classes for configuration panels, the below implementation
   * is for the case when all the panels have the same class, if this is
   * not the case then this method can be implemented as a switch-case
   * statement as stated in the method description.
   */
    return new MyMultiPanel();
}

/**
 * Returns the configuration validator specific to the passed panel
 * index.
 * <P>
* A possible implementation for this method, is a switch-case statement
 * that
 * switches on the panelIndex to create the validator specific for each
 * panel.
 *
 * @param panelIndex
 *       The panel index that we want to create a validator for.
 *       This must be >= 0 and < number of panels of this configurator.
 *
 * @return
 * The PxConfigurationValidator specific for the passed panel index
*
```

```
     * @throws ClassNotFoundException
     *       If the validator class was not found at runtime.
     *       The configurator author shouldn't worry himself with handling
     *       the exception unless he wants to override the default handling
     *       mechanism, or he wants to specify a special message.
     */
  protected PxConfigurationValidator getValidator(int panelIndex) throws
     ClassNotFoundException {
        /* creates classes for configuration validators, the below
         * implementation
         * is for the case when all the validators have the same class, if
         * this is
         * not the case then this method can be implemented as a switch-case
         * statement as stated in the method description.
         */
          return new MyMultiPanelValidator();
    }
}
```

The Panel

The Panel is responsible for:

- Displaying the configurable properties to the user.
- Saving the user configured values in the owner's DOM.
- Validating the user configuration. *

* The validation method in the panel can suffice for the need of a validator, or the opposite may occur, as Px calls the two, so any of them can do the job, or both can be used for separate checks.

The Panel methods were introduced before in the PxSingleConfigurator. (Remember that PxSingleConfigurator extends PxConfigurationPanel—see "Single Panel Configurators.") The following example shows how the panel code will look:

```
import com.mentor.px.api.*;

public class MyMultiPanel extends PxConfigurationPanel {

    /** Creates new MyMultiPanel */
```

```
    public MyMultiPanel() {
    }

    /**
     * Creates the content of the panel. This method gets called when
     * the panel is first created, so any initialization code should
     * go here.
     *
     * @throws  PxConfiguratorException if any initialization errors
     *          occur.
     */
    protected void initializeComponents() throws PxConfiguratorException {
        /* Insert GUI creation code here */
            .
            .
            .
    }

    /**
     * Fills the value fields of the panel from the owner properties.
     */
    public void fillPanel() {
    }


    /**
     * Saves the values entered in this panel by the user to
     * the owner's properties.
     */
    public void savePanel() {
    }
}
```

The panel might include also a **validateSettings()** method, to check that the user configuration is correct, as well as a **highlightProperty(PxProperty)** to highlight the faulty property in case the error can be traced back to a single property:

```
  /**
   * Called while the panel is still visible but after the
   * properties have been set to validate this panel's settings.
   * <p>
   * The default implementation performs no validation.
   *
   * @throws  PxConfiguratorException if an invalid setting is
   *          found.
   */
  public void validateSettings() throws PxConfiguratorException {
  }


  /**
```

```
 * Highlights a property in the dialog to indicate the source of
 * an error.  Configurator panels may override this method, but in
 * general it is better to override
 * <code>getPropertyInputField</code>.
 * <p>
 * The default implementation calls
 * <code>getPropertyInputField</code> to obtain the input field to
 * be highlighted.  It then directs focus to the field.  If it is
 * a text input field, the text contents are selected so they will
 * be overwritten when the user types.
 *
 * @param   property    The property to highlight.
 *
 * @return  <code>true</code> if something was highlighted.
 *          <code>false</code> means the dialog box may be closed
 *          because there's no additional information to display.
 */
public boolean highlightProperty(PxProperty property) {
}
```

Tip:

The Default Configuration can be used as one of the panels in a MultiPanelConfigurator by putting the following code in the **initializeComponents()** method:

```
/* defaultPanel is declared before as PxConfigurationPanel or DefaultPanel
*/
defaultPanel = new DefaultPanel();
defaultPanel.setConfigGroup("myGroup");
defaultPanel.setConfigurableRoot(this.getOwner().getConfigurableRoot());
try {
       defaultPanel.initialize(this.configuration);
} catch (PxConfiguratorException pce) {
     /* Handle initialization errors */
}

this.add(defaultPanel, java.awt.BorderLayout.CENTER);
```

Add the following line to the **fillPanel()** method:

```
defaultPanel.fillPanel();
```

The Validator:

The Validator is responsible for checking the correctness of the user configuration and reporting back any errors. As seen in the Panel, the validation task can be performed totally inside the Panel. However, in some cases the task needs to be isolated (to share the validator for example).

The PxConfigurationValidator class contains only one method to override, **isValid(),** which does the same thing as the Panel's **validateSettings()** method. A sample validator will look like the following:

```
import com.mentor.px.api.*;

public class MyConfigurationValidator extends PxConfigurationValidator {

    /** Creates new MyConfigurationValidator */
    public MyConfigurationValidator() {
    }

    /**
     * Checks that the properties of this ConfigurationValidator are valid.
     *
     * This method returns true if the properties are valid, If any property
     * is invalid this Method should throw an exception indicating which
     * property
     * has the error and describing the error for the caller.
     *
     * @return
     *     <CODE>true</CODE> if the properties of this validator are valid.
     *       This method shouldn't return <CODE>false</CODE> if there is an
     *       invalid property, instead it should throw an exception.
     *
     * @throws PxConfiguratorException
     *    If any property holds an invalid information, The exception should
     *    indicate which property was errored, and describe the error.
     *
     */
    public boolean isValid() throws PxConfiguratorException {
        /* Do some checking */
            .
            .
            .
```

```
        if (error) {
            throw new PxConfiguratorException("Error in ... due to ...",
                                                                    erroredProperty);
        }

        return true;
    }

}
```

Summary: Selecting a Base Class

The following table describes how to select the base class from which to derive a configurator:

	Case	Examples from Px
Default Configuration	Most Common Simple configuration tasks Prompts the user to enter values, select from values, ...	Basic Configuration. Project Settings.
PxSingleConfigurator	A more complex configuration task, possibly requires complex GUI or pre-calculations. Needs only *one* GUI panel	Scatter Loader Configurator.
PxAbstractConfigurator	Legacy GUI; doesn't conform to Platform Express APIs	None
PxMultiConfigurator	Simple or complex configuration tasks that require *more than one* panel.	Required Drivers Configurator.

Validators

Validators are designated with a *validator* element under the *configurators* element of a component file, bus definition file, or in the default configurators file. Validators are associated with specific configurators by matching the type name of the validator to the type name of the configurator. They validate the result of the configuration after the user has accepted the input.

If a validator detects an error, it should throw a PxConfiguratorException that includes a descriptive message. If the user can remedy the error by reconfiguring one of the properties, the Exception should be constructed with a reference to the property. Depending on the capabilities of the configurator, this may cause focus to be given to the erroneous user input on the configuration form.

The following validator example uses a hypothetical case of a memory with user settable data and address widths. The component XML file can be used to place many constraints on user input. However we want to place an additional constraint of maximum memory capacity. This means that as the user-selected data width doubles, the maximum allowable address width decreases by 1 bit. This cannot be expressed in the component XML so we write a validator to programmatically impose this constraint.

The following is a partial listing of the XML file showing the two width parameters and their XML-imposed constraints. It also shows the validator declaration.

```
<component xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="schema/1.0/pxComponents.xsd">
    <name>ram</name>
    <version>1.0</version>
    <busInterfaces>
        <busInterface interfaceId="ambaAHBLite">
            <busType>ambaAHBLite</busType>
            <slave>
                <memoryMap>
```

```
                    <addressBlock>
                        <baseAddress resolve="user"
                                     configGroups="requiredConfig"
                                     id="baseAddress" order="1"/>
                        <bitOffset>0</bitOffset>
                        <range resolve="dependent"
                               dependency="pow(2,id('addrWidth'))"
                               id="addressRange"/>
                        <width resolve="user" id="dataWidth"
                               prompt="Data Width:"
                               format="choice" choiceRef="dataWidthOptions"
                               configGroups="requiredConfig"
                               order="3">8</width>
...
    <hwModel>
...
        <hwParameters>
            <hwParameter name="addressSize" dataType="integer" id="addrWidth"
                         resolve="user" configGroups="requiredConfig"
                         prompt="Address Width:" format="long" minimum="14"
                         order="2">18</hwParameter>
        </hwParameters>
    </hwModel>
    <configurators owner="component">
        <validator>
            <type>Basic</type>
            <javaClass>ExampleValidator</javaClass>
        </validator>
    </configurators>
    <ui>
        <uiChoice id="dataWidthOptions">
            <uiChoiceElement>8</uiChoiceElement>
            <uiChoiceElement>16</uiChoiceElement>
            <uiChoiceElement>32</uiChoiceElement>
            <uiChoiceElement>64</uiChoiceElement>
        </uiChoice>
    </ui>
  ...
```

The validator defined by this component is of type *Basic* associated with object type *component*. When the *Basic* configurator for components as defined in the default configurators file is run on this component, the *ExampleValidator* class will be loaded and its *validateSettings* method will be called.

The classes declared by a component are searched for in the following locations:

- Component directory

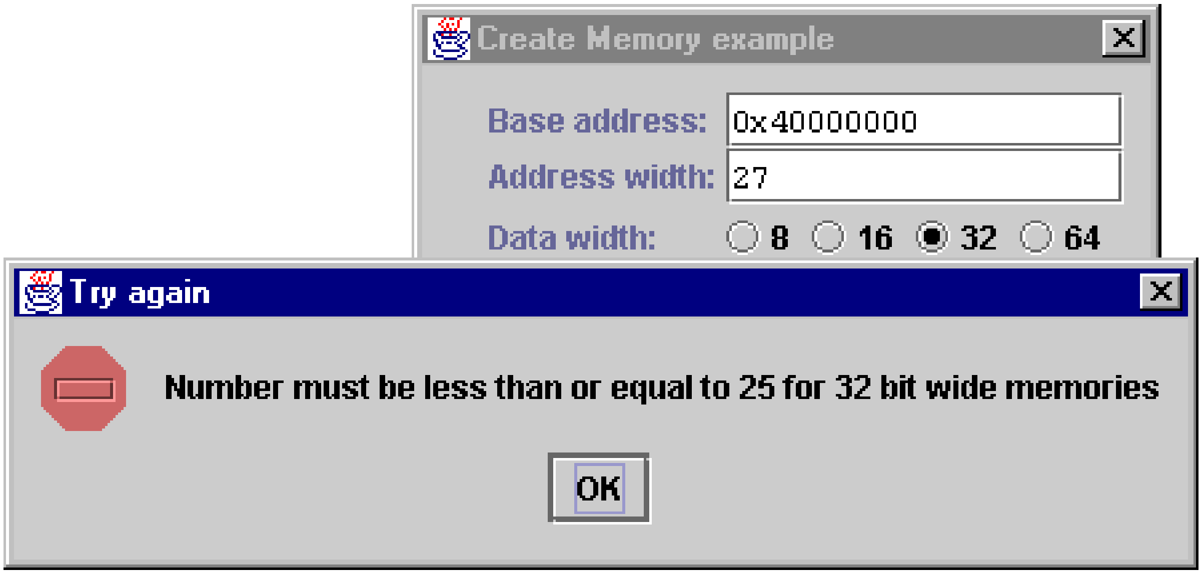
- Class directory of the library the component belongs to
- Class directory under PXHOME.

Valdators derive from the abstract class *com.mentor.api.PxAbstractValidator* and must implement the validateSettings method.

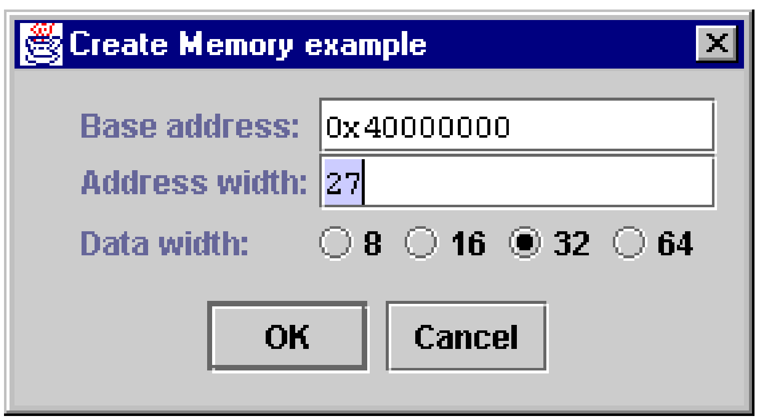
The full implementation for our hypothetical example is as follows.

```
import com.mentor.px.api.*;
import com.mentor.px.design.PxProperty;

public class ExampleValidator extends PxAbstractValidator {

    public void validateSettings() throws PxConfiguratorException {
        PxProperty prop = getOwner().getConfigurableRoot();
        PxProperty addrWidthProp = prop.resolveId("addrWidth");
        long dataWidth = prop.resolveId("dataWidth").getLongValue();
        long byteWidth = dataWidth / 8;
        long maxAddrWidth = 27;

        // Decrement maximum allowable address width for every
        // doubling of the byte width
        while (byteWidth > 1) {
            maxAddrWidth--;
            byteWidth /= 2;
       }

        // Compare requested address width to maximum allowed.
        if (addrWidthProp.getLongValue() > maxAddrWidth) {
            throw new PxConfiguratorException(
                "Number must be less than or equal to " +
                maxAddrWidth + " for " +
                dataWidth + " bit wide memories",
                addrWidthProp);
        }
    }
}
```

When the component is added and configured, the validator is executed. If too large a number is entered for the address width, the error message from the exception appears in a pop-up box, as shown in the following example.



Since the exception was constructed with the address width property, when the user clicks OK, the erroneous entry is selected and highlighted.



Chapter 3
Generators

Introduction

This chapter discusses topics related to creating generators. Generators are Java classes that create HDL code, software, simulation environment scripts, simulation stimulus or anything else that contributes to building and verifying a design in Platform Express. When you integrate a component into a Platform Express component library, you may have to supply generators to support that component. You extend the PxGenerator class built into Platform Express to create generators.

You specify all the generators for a component in the component's *component*.xml file. As shown in the following example, you use the generator element to do this.

```
<generator>
  <name>CodeAddressVldGenerator</name>
  <generatorPhase>0</generatorPhase>
 </generator>
```

Design Database

Platform Express constructs a design database (see Figure 3-1) for each design. This database includes detailed information on the properties of each component in the design, as well as global project settings and compiled object code and supporting files used for verification. Platform Express reads the component properties to determine which generators it needs to run. Generators themselves interact with the design database both to obtain component and project information and to create supporting files for verification. Generators access the design database through the Platform Express API, which is discussed in the following section.

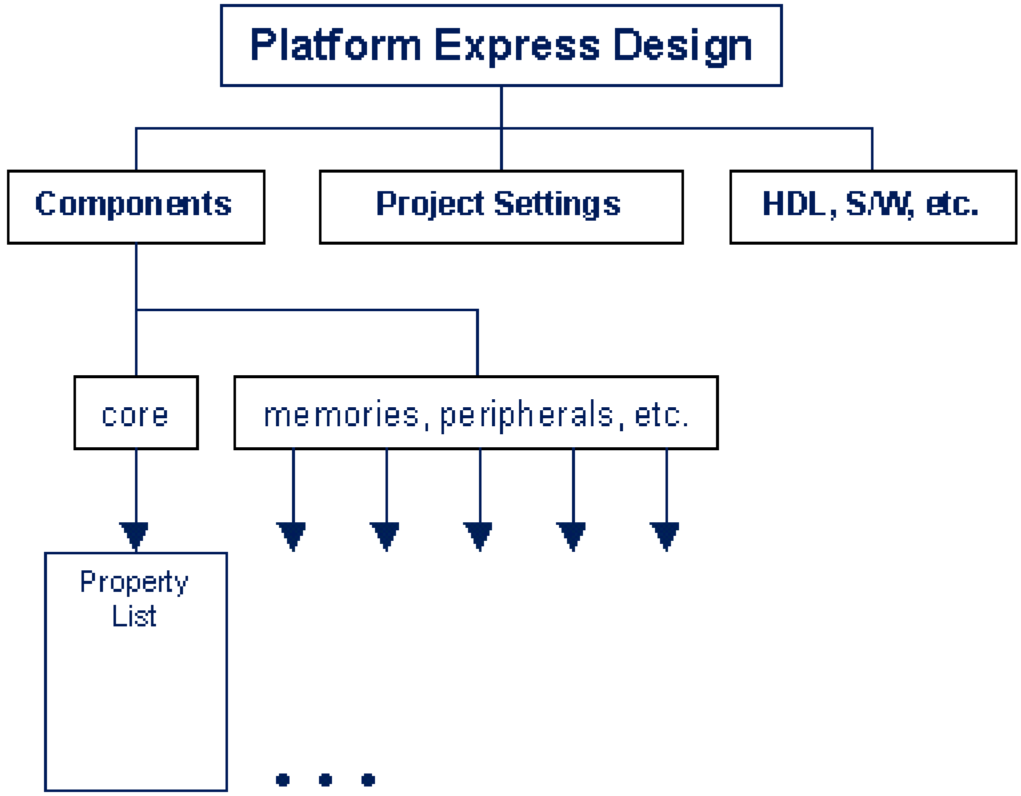


Figure 3-1. Design Database

The Platform Express API

Creating a generator amounts to creating an extension to the Platform Express application. You do this by means of the Platform Express API, which makes available several Java classes for this purpose:

- **PxGenerator** class
  is the abstract base class for Platform Express generators. To create a generator, you extend this class by implementing the generate() method. The PxGenerator class has access to the design database, allowing the generator to retrieve any information necessary to perform its generation function.

You declare Platform Express generators in component definition files (*component*.xml file). Platform Express loads the PxGenerator class at runtime and executes the generate() method when a user generates a design.

- **PlatformDesign** class
  represents the complete encapsulation of the user's design. There is one active instance at any given time. As the user builds up the design (from the GUI or batch input), objects are added to or removed from the data members contained in this class

- **PxBus** class
  acts as a wrapper for the Platform Express bus information and object methods.

- **PxComponent** class
  is the base class for all components in a hierarchical design.

- **PxConfigurationPanel** class
  is an extension of JPanel and is the base class for a single configuration panel. It can be used inside a single configuration dialog box, or as part of a multi-panel configurator.

- **PxConfigurationValidator** class
  handles the validation task for the configurator panels. There should be a validator for each configuration panel. The Configurator author should extend this class and provide the implementation for the isValid() method to check for thevalidity of the data.

- **PxConfigurator** class
  is the base class for configurators, which enable component parameters to be configured at instantiation in a design.

- **PxDef** class
  provides static definitions for Platform Express.

- **PxLog** class
  issues log messages to System.err and optionally to a file.

- **PxProperty** class
  A Px property is a hierarchical piece of information about a Px design component. A property may have a string value, or it may contain additional ordered child properties. All properties have a name. Some may have attributes to indicate such things as how to obtain their value if it has not been pre-assigned.

- **PxGeneratorException** class
  provides exception handling.

- **PxConfiguratorException** class
  provides exception handling.

Detailed Javadoc information for the Platform Express API classes, constructors, and methods, can be found in the Platform Express installation directory, under the *api* directory. You can read this documentation using any HTML browser. Invoke the browser on *index.html*.

Creating a Generator Class

To create a generator you write a Java class that extends the PxGenerator class and provide an implementation of the *generate* method. You can download java development tools from http://java.sun.com/j2se/1.3. You should compile the class into the component directory that references the generator. When you compile, use a class path that includes the class directory under the Platform Express installation. Here is a compilation example.

```
javac -classpath $PXHOME/class MyGenerator.java
```

In the XML file, the element

generators/componentGenerator/javaClass/className

should contain the name of the class you just created. When Platform Express loads generator classes, it searches for them in the following locations, in the specified order:

1. <component_directory>*/class*

2. *componentLibrary/class*

3. *$PXHOME/class*

Generator Chains

Since some generators need to run before others, the component definition file provides a generatorPhase element to control their sequence of operation. The range of generator phases runs from 0 to 15. Multiple generators may run in the same phase. Although the order in which generators run within a given phase cannot be controlled, all generators assigned that phase run before generators in the next phase.

Generators are scheduled and invoked through generator chain files. The Platform Express generator chain that is activated by the **Tools > Build** menu selection runs all component generators that have an element generators/componentGenerator/group equal to value "build".

Component generators are compiled java classes. These classes are loaded at runtime and are searched for, in the order given, at these locations:

1. <path_to_component_directory>*/class*

2. <path_to_component_library>*/class*

Soft Paths and Generators

Px often will need to generate text that refers to file pathnames. All file pathnames should appear as either relative paths or as soft paths. No absolute paths should be used. Px will generate a file that assigns absolute paths to the "soft path" variables. This file will be executed before Px executes any local command. The filename "pxenv" is reserved for this purpose.

Generator Author Responsibility

Generator authors are responsible for resolving variables, converting them to the appropriate syntax, and generating soft paths for all file locations. The Px API contains utilities to assist with this. Here is a sampling of generation utilities. Refer to the Px API for details.

- **com.mentor.px.api.PxDef**
  contains many static definitions that can be used by generator authors. PxDef.PXVAR_COMPONENT_LIBRARY and PxDef.PXVAR_COMPONENT_LOCATION are strings defining the reserved variable names.

- **com.mentor.px.design.PlatformDesign.getSoftPath()**
  gets the soft path data base for the design.

- **String SoftPath.get(String directoryName, String prefix);**
  A generator can get a soft path name with this method.

- **com.mentor.px.common.PlatformDependent**
  contains methods to handle platform dependencies.

  Some methods are:

  - **writeShellScript(...)** writes command text to a file.

  - **execShellScript(...)** executes a shell script, first assigning the soft path variables

  - **resolveVariables(...)** filters text to resolve and format variables. This will also handle the Px reserved variables by resolving the pathnames and creating soft paths for them.

- **com.mentor.px.design.PlatformDesign.getVenvLocation()**
  gets the top-level directory of the current verification environment. Most generated files will be written to this directory or one of its subdirectories.

Chapter 4
Decoder Templates

Introduction

Platform Express connects components (core platforms, peripherals, memories and bus bridges) that share the same bus interface. A designer should be able to connect any peripheral that implements a certain bus interface to a core platform that implements the same bus interface. In practice, however, some additional logic usually has to be created to allow the peripheral to function properly on that bus. For instance, many components have a select signal that has to be asserted whenever the value of the address bus is within a certain range. This requires some logic in the bus interface.

A bus decoder template file specifies all the logic and connections that will be created for a particular bus within Platform Express. A Platform Express generator uses the decoder template and the current design database to create an HDL module/entity which connects and interfaces all of the components that belong to that bus.

Pins: Logical and Physical, Master and Slave

In Platform Express, bus descriptions in the *busdef* directories that reside in the component libraries contain a list of logical signals for particular buses. These signals, and their functions, are described in the specification for each bus. This specification is usually available from the designer of the bus.

The component descriptions within Platform Express identify the physical pins on each component. These are identified in the <signal> element of the component description file. The physical names of the pins on the components do not always match the logical name of the signal in the bus specification. The component descriptions within Platform Express identify bus interfaces implemented on each component, and they map the physical pins of the component onto the logical signals of the bus specification. Most components will not implement all of the logical signals specified in a bus definition.

In addition to the logical and physical pin mapping, each bus interface on a component is identified as a master or a slave. A slave is a component that is capable of responding to requests for bus transactions, but is not capable of initiating them. A master is a component that is capable of initiating a bus transaction. Typically, a memory or peripheral is a slave device, while a DMA controller or core platform is often a bus master.

Within the bus decoder template files, all definitions are made in terms of the logical signals, and master and slave connections. When the bus decoder generator is run, it creates an HDL source file where the physical names of the component signals are substituted in the text for the logical pin description. Bus decoder template files are located in <library_name>*/componentLibrary/decoder*.

Some Basic Concepts and Syntax

Bus decoder template files are written in XML. The whole definition must be contained within a *<decoderTemplate>* tag, which defines the name of the bus.

The decoder definition also contains HDL code, so the language being used must be declared. Legal choices are "vhdl" and "verilog". The <language> tag is used by Platform Express to select the correct way of compiling the generated decoder.

```
<!DOCTYPE decoderTemplate SYSTEM
"px:/dtd/decoderTemplate.dtd" >
    <decoderTemplate name="pVCI">
    <language name="vhdl"/>

        ... bus decoder definition
</decoderTemplate>
```

Code Sections

Within the decoder defintion, <code> tags can be included. The decoder generator is responsible for creating the entity(architecture) and module skeleton. The text within a <code> tag is a snippet of HDL code that will be included within the generated decoder. A bus decoder template can have any number of code sections.

The following code would always be included in a generated decoder.

```
<code>
   assert false
     report &quot;Bus Decoder Active&quot;
     severity note;
</code>
```

It's also worthwhile noting that some symbols used in HDLs are reserved XML characters. In the above example, the double quote character that delimits the "Bus Decoder Active" string cannot be used directly. Instead, the character has to be replaced with ". Other special characters are:

HDL Text	XML required
<	<
>	>
&	&
'	&pos;
"	"

Creating connections within a decoder is quite straightforward. However, at the time of creating a decoder template, the writer has no idea of how many master or slave devices might be connected to the bus. So, we can create a <code> section, setting the *loop* attribute to be “slave”. Instead of creating a single block of code (like the first example), this next example will iterate for each slave device connected to the bus, customising the generated code for each slave.

The decoder template uses the <lPin> tag (logical pin) to specify the signals used in the bus decoder. By setting the 'master' parameter to be 'true' or 'false', the decoder generator will insert the physical signal name of the version of the bus signal attached to the bus master or current bus slave.

The 'nopin' parameter is a method of controlling how the decoder generator works when a component does not have a physical pin mapped to the logical pin specified in the <lpin> element. If an <lPin> is marked as "required", then the decoder writer has decided that a correct design cannot created without the presence of that pin.

In this example, if the bus master did not implement the "WRITE_EN" logical pin, then the decoder generator (and the whole design build) would be aborted. However, if the slave did not implement the "WRITE_EN" pin, only that iteration of the code would not be created. So this <code> section would still connect up the master "WRITE_EN" pin to all of the "WRITE_EN" pins on the slave devices that supported that pin.

```
            <code loop="slave">
              <lPin name="WRITE_EN" master="false"
nopin="continue"/> &lt;= <lPin name="WRITE_EN" master="true"
nopin="required"/>;
            </code>
```

```
                        WRITE_EN_Slave_1 <= WRITE_EN;
                        WRITE_EN_Slave_2 <= WRITE_EN;
```

Sometimes, these control signals need to be qualified by other signals such as clocks (perhaps this bus is synchronous) and addresses. It is possible to add other signals to create more complex code structures. Usefully, the <addressDecodeExpression/> tag will return a complex expression checking that the value on the address bus is within the range of addresses supported by each slave device. The bus decoder knows which logical signal is the address bus because this signal is tagged with the <address/> attribute in the bus definition file.

```
           <code loop="slave">
              process (<lPin name="CLOCK" master="true"
nopin="required"/> )
                 begin
```

```
                   if (<lPin name="RESET" master="true"
nopin="required"/> = &apos;0&apos; then
                     <lPin name="WRITE_EN" master="false"
nopin="continue"/> &lt;= &apos;0&apos;
                   else
                     if (<lPin name="CLOCK" master="true"
nopin="required"/> &lt;= &apos;1&apos;  and
                         <addressDecodeExpression/>) THEN
                        <lPin name="WRITE_EN" master="false"
nopin="continue"/> &lt;= &apos;1&apos; else
                        <lPin name="WRITE_EN" master="false"
nopin="continue"/> &lt;= &apos;0&apos;;
                     end if;
                   end if;
                 end process;
            </code>

        process (CLOCK_master_1)
           begin
             if (RESET_master_1 ='0') then
               WRITE_EN_slave_1 <= '0'
             else
               if (CLOCK_master_1 ='1' AND
                   ADDRESS_master_1 >= 16#fff00000# AND
ADDRESS_master_1 < 16#fff00010#) then
                  WRITE_EN_slave_1 <= '1';
               else
                  WRITE_EN_slave_1 <= '0';
               end if;
             end if;
           end process;

    process (CLOCK_master_1)
           begin
             if (RESET_master_1 ='0') then
               WRITE_EN_slave_2 <= '0'
             else
               if (CLOCK_master_1 ='1' AND
                   ADDRESS_master_2 >= 16#fff00100# AND
ADDRESS_master_1 < 16#fff001ff#) then
                  WRITE_EN_slave_2 <= '1';
               else
                  WRITE_EN_slave_2 <= '0';
```

```
                end if;
              end if;
            end process;
```

One common requirement is to merge together signals from different slaves into one signal that is connected to the master (perhaps using a multiplexer to channel the correct signal back). One way of doing this is to create an intermediate signal (this has to be declared before it can be used, so the declaration is put in a <code decl="true"> section. If the signal requires any libraries or packages to be made visible, these declarations can be put in a <header> section. As the decoder is generated, the generator will work out the appropriate places for these code segments to be inserted.

```
<header>
library ieee;
use ieee.std_logic_1164.all;
</header>

<code decl="true">
         signal select : std_logic_vector (<noSlaves/>-1
downto 0);
</code>

<code>
    <code loop="slave">
        select(<currentSlaveIndex>-1) &lt;= &apos;1&apos when
<addressDecodeExpression/>
                                                     else
&apos;0&apos;

        <lPin name="RDATA" master="true" nopin="required"/>
&lt;=
           <code loop="slave" separator=" else ">
                     <lPin name="RDATA" master="false"
nopin="continue"/> when select

(<currentSlaveIndex/>-1)=&apos;1&apos;
           </code> else (others =;&gt; &apos;0&apos);
    </code>
</code>
```

```
architecture PlatformExpress of design is
       signal select : std_logic_vector (3-1 downto 0);
begin

       select(1-1) <=  '1' when ADDRESS_master_1 >=
16#fff00000# AND ADDRESS_master_1 < 16#fff00010#;
       select(2-1) <=  '1' when ADDRESS_master_1 >=
16#fff00000# AND ADDRESS_master_1 < 16#fff00010#;
       select(3-1) <=  '1' when ADDRESS_master_1 >=
16#ffe00000# AND ADDRESS_master_1 < 16#ffe00010#;

       RDATA_master_1 <= RDATA_slave_1 when select(1-1) =
'1' else
                                         RDATA_slave_2 when
select(2-1) = '1' else
                                         (others => '0');
  end PlatformExpress;
```

This style is very useful because some slaves (like slave 3 in the above example) may not implement an RDATA bus (perhaps a slave with write-only registers), so the "RDATA_slave_2 when select(x-1) = '1' " will be omitted for those slaves. However, the default "others" clause will ensure that a well-controlled value is placed on the RDATA_master_1 bus signal when that slave is being accessed.

There are alternative ways to achieve a similar result. One is to specify a default value for a signal if it does not exist.

```
      <lPin name="RDATA" master="false" nopin="continue"/>
when select (<currentSlaveIndex/>-1)=&apos;1&apos;
```

could be rewritten as :

```
     <lPin name="RDATA" master="false" nopin="default"
default="01010101" /> when select (<currentSlaveIndex/>-
1)=&apos;1&apos;
```

generating

```
                                         RDATA_master_1 <=
RDATA_slave_1 when select(1-1) = '1' else
```

```
RDATA_slave_2 when select(2-1) = '1' else

"01010101"       when select(3-1) = '1' else

(others => '0');
```

Another is to specify an <alternativeCode> section which would substitute alternative code if the signal did not exist for that slave.

```
          <code decl="true">
                 constant RDATA_DEFAULT : std_logic_vector (7
downto 0) := "11110000";
         </code>

         <code>
             <lPin name="RDATA" master="false"
nopin="alternative" /> when select (<currentSlaveIndex/>-
1)=&apos;1&apos;
             <alternativeCode>
                  <code>
                       RDATA_DEFAULT when select
(<currentSlaveIndex/>-1)=&apos;1&apos;
                  </code>
             </alternativeCode>
         <code>
```

generating

```
RDATA_master_1 <= RDATA_slave_1   when select(1-1) = '1' else

RDATA_slave_2   when select(2-1) = '1' else

RDATA_DEFAULT when select(3-1) = '1' else

(others => '0')
```

Handling Data Busses Of Differing Widths

The <lPin> element has a “fill” attribute where you may specify a fill character (usually 0) to resolve busses of different sizes. This attribute should be used on an <lPin> on the right-hand side of an assignment expression. Additional details can be found in the *decoderTemplate.dtd* file.

Some Tips For Bus Decoder Template Writers

The first and most important tip is to assume the worst. Mark all <lPins> as "required" unless it is certain that the generated code will handle conditions where that pin is not implemented in the master and slave devices being connected together.

It is also important to think about statements where a pin will be implemented for some instances but not for others.

Examples

The specific syntax and semantics of the decoder template files is given in:

$PXHOME/dtd/decoderTemplate.dtd

This file contains comments and gives a great deal of additional information about decoder templates.

Decoder template files are located in:

<component_library>*/componentLibrary/decoder*

where <component_library> is the path to the top of a library. Not all libraries will contain decoder templates.

Mentor Graphics Trademarks

The following names are trademarks, registered trademarks, and service marks of Mentor Graphics Corporation:

3D Design™, A World of Learning℠, ABIST™, Arithmetic BIST™, AccuPARTner™, AccuParts™, AccuSim®, ADEPT™, ADVance™ MS, ADVance™ RFIC, AMPLE™, Analog Analyst™, Analog Station™, AppNotes℠, ARTgrid™, ArtRouter™, ARTshape™, ASICPlan™, ASICVector Interfaces™, Aspire™ Assess2000℠, AutoActive®, AutoCells™, AutoDissolve™, AutoFilter™, AutoFlow™, AutoLib™, AutoLinear™, AutoLink™, AutoLogic™, AutoLogic BLOCKS™, AutoLogic FPGA™, AutoLogic VHDL®, AutomotiveLib™, AutoPAR®, AutoTherm®, AutoTherm Duo™, AutoThermMCM™, AutoView™, Autowire Station™, AXEL™, AXEL Symbol Genie™, BISTArchitect™, BIST Compiler℠, BIST-In-Place℠, BIST-Ready℠, Board Architect™, Board Designer™, Board Layout™, Board Link™, Board Process Library™, Board Station®, Board Station Consumer™, BOLD Administrator™, BOLD Browser™, BOLD Composer™, BSDArchitect™, BSPBuilder™, Buy on Demand™, Cable Analyzer™, Cable Station™, CAECO Designer™, CAEFORM™, Calibre®, Calibre CB™, Calibre DESIGNrev™, Calibre DRC™, Calibre DRC-H™, Calibre FRACTUREh™, Calibre FRACTUREj™, Calibre FRACTUREk™, Calibre FRACTUREm™, Calibre FRACTUREt™, Calibre Interactive™, Calibre LITHOview™, Calibre LVS™, Calibre LVS-H™, Calibre MDPview™, Calibre MGC™, Calibre OPCpro™, Calibre OPCsbar™, Calibre ORC™, Calibre PRINTimage™, Calibre PSMgate™, Calibre PSMcheck™, Calibre RVE™, Calibre TDopc™, Calibre WORKbench™, Calibre xRC™, CAM Station™, Capture Station®, CAPITAL™, CAPITAL Analysis™, CAPITAL Bridges™, CAPITAL Documents™, CAPITAL H™, CAPITAL Harness™, CAPITAL Harness Systems™, CAPITAL H the complete desktop engineer®, CAPITAL Insight™, CAPITAL Integration™, CAPITAL Manager™, CAPITAL Manufacturer™, CAPITAL Support™, CAPITAL Systems™, Cell Builder™, Cell Station®, CellFloor™, CellGraph™, CellPlace™, CellPower™, CellRoute™, Centricity™, CEOC™, ChaseX™, CheckMate™, CHEOS™, Chip Station®, ChipGraph™, CommLib™, CommLib BMC™, Concurrent Board Process℠, Concurrent Design Environment™, Connectivity Dataport™, Continuum™, Continuum Power Analyst™, CoreAlliance™, CoreBIST™, Core Builder™, Core Factory™, Co-Verification Environment™, CTIntegrator™, DataCentric Model™, DataFusion™, Datapath™, Data Solvent™, dBUG™, Debug Detective™, DC Analyzer™, Design Architect®, Design Architect Elite™, DesignBook®, Design Capture™, Design Manager™, Design Station®, DesignView™, DesktopASIC™, Destination PCB®, DFTAdvisor™, DFTArchitect™, DFTInsight™, DirectConnect℠, DSV™, Direct System Verification™, Documentation Station™, DSS (Decision Support System)™, DSV™, E3LCable™, ECO Immunity℠, EDGE (Engineering Design Guide for Excellence)℠, EDT™, Eldo™, EldoNet™, ePartners™, EParts®, Empowering Solutions™, Engineer’s Desktop™, EngineerView™, ENRead™, ENWrite™, ESim™, Exemplar™, Exemplar Logic™, Expedition™, Expert2000℠, Explorer CAECO Layout™, Explorer CheckMate™, Explorer Datapath™, Explorer Lsim™, Explorer Lsim-C™, Explorer Lsim-S™, Explorer Ltime™, Explorer Schematic™, Explorer VHDLsim™, ExpressI/O™, FabLink™, Falcon®, Falcon Framework®, FastScan™, FastStart™, FastTrack Consulting℠, First-Pass Design Success™, First-Pass success℠, FlexSim™, FlexTest™, FDL (Flow Definition Language)™, FlowTabs™, FlowXpert™, FORMA™, FormalPro™, FPGA Advantage®, FPGAdvisor™, FPGA BoardLink™, FPGA Builder™, FPGASim™, FPGA Station®, FrameConnect™, Galileo®, Gate Station®, GateGraph™, GatePlace™, GateRoute™, GDT®, GDT Core®, GDT Designer™, GDT Developer™, GENIE™, GenWare™, Geom Genie™, HDL2Graphics™, HDL Architect™, HDL Architect Station™, HDL Author™, HDL Designer™, HDL Designer Series™, HDL Detective™, HDL Inventor™, HDL Pilot™, HDL Processor™, HDL Sim™, HDLWrite™,Hardware Modeling Library™, HIC rules™, Hierarchical Injection™, Hierarchy Injection™, HotPlot®, Hybrid Designer™, Hybrid Station®, IBD™, IC Design Station™, IC Designer™, IC Layout Station™, IC Station®, ICbasic™, ICblocks™, ICcheck™, ICcompact™, ICdevice™, ICextract™, ICGen™, ICgraph™, ICLink™, IClister™, ICplan™, ICRT Controller Lcompiler™, ICrules™, ICtrace™, ICverify™, ICview™, ICX™, ICX Active™, ICX Custom Model™, ICX Custom Modeling™, ICX Plan™, ICX Pro™, ICX Project Modeling™, ICX Sentry™, ICX Standard Library™, ICX Verify™, ICX Vision™, IDEA Series™, Idea Station®, INFORM®, IFX™, Inexia™, Integrated Product Development®, Integra Station™, Integration Tool Kit™, INTELLITEST®, Interactive Layout™, Interconnect Table™, Interface-Based Design™, IntraStep℠, Inventra™, InventraIPX™, Inventra Soft Cores™, IP Engine ™, IP Evaluation Kit™, IP Factory™, IP -PCB™, IP QuickUse™, IPSim™, IS_Analyzer™, IS_Floorplanner™, IS_MultiBoard™, IS_Optimizer™, IS_Synthesizer™, ISD Creation℠, ITK™, It's More than Just Tools℠, Knowledge Center℠, Knowledge-Sourcing℠, LAYOUT™, LNL™, LBIST™, LBISTArchitect™, Language Neutral Licensing™, Lc™, Lcore™, Leaf Cell Toolkit™, Led™, LED LAYOUT™, Leonardo®, LeonardoInsight™, LeonardoSpectrum™, LIBRARIAN™, Library Builder™, Logic Analyzer on a Chip℠, Logic Builder™, Logical Cable™, LogicLib™, *logio*™, Lsim™, Lsim DSM™, Lsim-Gate™, Lsim Net™, Lsim Power Analyst™, Lsim-Review™, Lsim-Switch™, Lsim-XL™, Mach PA™, Mach TA™, Manufacture View™, Manufacturing Advisor™, Manufacturing Cable™, MaskCompose™, MaskPE®, MBIST™, MBISTArchitect™, MBIST Full-Speed™, MBIST Flex™, MBIST Manager™, MCM Designer™, MCM Station®, MDV™, MegaFunction™, Memory Builder™, Memory Builder Conductor™, Memory Builder Mozart™, Memory Designer™, Memory Model Builder™, Mentor™, Mentor Graphics®, Mentor Graphics Support CD℠, Mentor Graphics SupportBulletin℠, Mentor Graphics SupportCenter℠, Mentor Graphics SupportFax℠, Mentor Graphics SupportNet-Email℠, Mentor Graphics SupportNet-FTP℠, Mentor Graphics SupportNet-Telnet℠, Mentor Graphics We Mean Business™, MicroPlan™, MicroRoute™, Microtec®, Mixed-Signal Pro™, ModelEditor™, Model*Sim*®, Model*Sim* LNL™, Model*Sim* VHDL™, Model*Sim* VLOG™, Model*Sim* SE™, ModelStation®, Model Technology™, ModelViewer™, ModelViewer*Plus*™, *MODGEN*™, Monet®, Mslab™, Msview™, MS Analyzer™, MS Architect™, MS-Express™, MSIMON™, MTPI℠, Nanokernel®, NetCheck™, NETED™, Nucleus™, Online Knowledge Center℠, OpenDoor℠, Opsim™, OutNet™, P&RIntegrator™, PACKAGE™, PARADE™, ParallelRoute-Autocells™, ParallelRoute-MicroRoute™, PathLink™, Parts SpeciaList™, PCB-Gen™, PCB-Generator™, PCB IGES™, PCB Mechanical Interface™, PDLSim™, Personal Learning Program™, Physical Cable™, Physical Test Manager:SITE™, PLA Lcompiler™, Platform Express™, PLDSynthesis™, PLDSynthesis II™, Power Analyst™, PowerAnalyst Station™, Power To Create®, Precision™, Precision Synthesis™, Precision HLS™, Precision PNR™, Precision PTC™, Pre-Silicon™, ProjectXpert™, ProtoBoard™, ProtoView™, QNet™, QualityIBIS™, QuickCheck™, QuickConnect™, QuickFault™, QuickGrade™, QuickHDL™, QuickHDL Express™, QuickHDL Pro™, QuickPart Builder™, QuickPart Tables™, QuickParts™, QuickPath™, QuickSim™, QuickSimII™, QuickStart™, QuickUse™, QuickVHDL®, RAM Lcompiler™, RC-Delay™, RC-Reduction™, RapidExpert™, REAL Time Solutions!™, Registrar™, Reinstatement 2000℠, Reliability Advisor™, Reliability Manager™, REMEDI™, Renoir™, RF Architect™, RF Gateway™, RISE™, ROM Lcompiler™, RTL X-Press™, Satellite PCB Station™, ScalableModels™, Scaleable Verification™, SCAP™, Scan-Sequential™, Scepter™, Scepter DFF™, Schematic View Compiler, SVC™, Schemgen™, SDF™ (Software Data Formatter), SDL2000 Lcompiler™, Seamless®, Seamless C-Bridge™, Seamless Co-Designer™, Seamless CVE™, Seamless Express™, Selective Promotion™, SignaMask OPC™, Signal Spy™, Signal Vision™, Signature Synthesis™, Simulation Manager™, SimExpress™, SimPilot™, SimView™, SiteLine2000℠, SmartMask™, SmartParts™, SmartRouter™, SmartScripts™, Smartshape™, SNX™, SneakPath Analyzer™, SOS Initiative™, Source Explorer™, SpeedGate™, SpeedGate DSV™, SpiceNet™, SST Velocity®, Standard Power Model Format (SPMF)™, Structure Recovery™, Super C™, Super IC Station™, Support Services BaseLine℠, Support Services ClassLine℠, Support Services Latitudes℠, Support Services OpenLine℠, Support Services PrivateLine℠, Support Services SiteLine℠, Support Services TechLine℠, Support Services RemoteLine℠, Symbol Genie™, Symbolscript™, SYMED™, SynthesisWizard™, System Architect™, System Design Station™, System Modeling Blocks™, Systems on Board Initiative™, System Vision™, Target Manager™, Tau®, TeraCell™, TeraPlace™, TeraPlace-GF™, TechNotes™, The Ultimate Tool for HDL Simulation™, TestKompress™, Test Station®, Test Structure Builder™, The Ultimate Site For HDL Simulation™, TimeCloser™, Timing Builder™, TNX™, ToolBuilder™, TrueTiming™, Vlog™, V-Express™, V-Net™, VHDLnet™, VHDLwrite™, Verinex™, ViewCreator™, ViewWare®, Virtual Library™, Virtual Target™, Virtual Test Manager:TOP™, VR-Process℠, VRTX®, VRTXmc™, VRTXoc™, VRTXsa™, VRTX32®, Waveform DataPort™, We Make TMN Easy™, Wiz-o-matic™, WorkXpert™, xCalibre™, xCalibrate™, Xconfig™, XlibCreator™, Xpert™, Xpert API™, XpertBuilder™, Xpert Dialogs™, Xpert Profiler™, XRAY®, XRAY MasterWorks®, XSH®, Xtrace®, Xtrace Daemon™, Xtrace Protocol™, Zeelan®, Zero Tolerance Verification™, Zlibs™

Third-Party Trademarks

The following names are trademarks, registered trademarks, and service marks of other companies that appear in Mentor Graphics product publications:

Adobe, the Adobe logo, Acrobat, the Acrobat logo, Exchange, FrameMaker, FrameViewer, PostScript,and Reader are registered trademarks of Adobe Systems Incorporated in the United States and/or other countries.

Altera, ByteBlaster, Excalibur, and Quartus are trademarks or registered trademarks of Altera Corporation in the United States and other countries.

AM188, AMD, AMD-K6, and AMD Athlon Processor are trademarks of Advanced Micro Devices, Inc.

Apple and Laserwriter are registered trademarks of Apple Computer, Inc.

ARIES is a registered trademark of Aries Technology.

AMBA, ARM, ARMulator, ARM7TDMI, ARM7TDMI-S, ARM9TDMI, ARM9E-S, ARM946E-S, ARM966E-S, EmbeddedICE, StrongARM, TDMI, and Thumb are trademarks or registered trademarks of ARM Limited.

ASAP, Aspire, C-FAS, CMPI, Eldo-FAS, EldoHDL, Eldo-Opt, Eldo-UDM, EldoVHDL, Eldo-XL, Elga, Elib, Elib-Plus, ESim, Fidel, Fideldo, GENIE, GENLIB, HDL-A, MDT, MGS-MEMT, MixVHDL, Model Generator Series (MGS), Opsim, SimLink, SimPilot, SpecEditor, Success, SystemEldo, VHDeLDO and Xelga are registered trademarks of ANACAD Electrical Engineering Software, a unit of Mentor Graphics Corporation.

Avant! and Star-Hspice are trademarks of Avant! Corporation.

AVR is a registered trademark of Atmel Corporation.

Cadence, Affirma signalscan, Allegro, Analog Artist, Composer, Concept, Design Planner, Dracula, GDSII, GED, HLD Systems, Leapfrog, Logic DP, NC-Verilog, OCEAN, Physical DP, Pillar, Silicon Ensemble, Spectre, Verilog, Verilog XL, Veritime, and Virtuoso are trademarks or registered trademarks of Cadence Design Systems, Inc.

CAE+Plus and ArchGen are registered trademarks of Cynergy System Design.

CalComp is a registered trademark of CalComp, Inc.

Canon is a registered trademark of Canon, Inc. BJ-130, BJ-130e, BJ-330, and Bubble Jet are trademarks of Canon, Inc.

Centronics is a registered trademark of Centronics Data Computer Corporation.

ColdFire and M-Core are registered trademarks of Motorola, Inc.

Ethernet is a registered trademark of Xerox Corporation.

Foresight and Foresight Co-Designer are trademarks of Nu Thena Systems, Inc.

FLEXlm is a trademark of Globetrotter Software, Inc.

GenCAD is a trademark of Teradyne Inc.

Hewlett-Packard (HP), LaserJet, MDS, HP-UX, PA-RISC, APOLLO, DOMAIN and HPare registered trademarks of Hewlett-Packard Company.

HCL-eXceed and HCL-eXceed/W are registered trademark of Hummingbird Communications. Ltd.

HyperHelp is a trademark of Bristol Technology Inc.

Installshield is a registered trademark and service mark of InstallShield Corporation.

IBM, PowerPC, and RISC Systems/6000 are trademarks of International Business Machines Corporation.

I-DEAS and UG/Wiring are registered trademarks of Electronic Data Systems Corporation.

IKON is a trademark of Tahoma Technology.

IKOS and Voyager are registered trademarks of IKOS Systems, Inc.

Imagen, QMS, QMS-PS 820, Innovator, and Real Time Rasterization are registered trademarks of MINOLTA-QMS Inc. imPRESS and UltraScript are trademarks of MINOLTA-QMS Inc.

ImageGear is a registered trademark of AccuSoft Corporation.

Infineon, TriCore, and C165 are trademarks of Infineon Technologies AG.

Intel, i960, i386, and i486 are registered trademarks of Intel Corporation.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc.

Linux is a registered trademark of Linus Torvalds.

MemoryModeler MemMaker are trademarks of Denali Software, Inc.

MIPS is a trademark of MIPS Technologies, Inc.

MS-DOS, Windows 95, Windows 98, Windows 2000, and Windows NT are registered trademarks of Microsoft Corporation.

MULTI is a registered trademark of Green Hills Software, Inc.

NEC and NEC EWS4800 are trademarks of NEC Corp.

Netscape is a trademark of Netscape Communications Corporation.

Novas, Debussy, and nWave are trademarks or registered trademarks of Novas Software, Inc.

OakDSPCore is a registered trademark for DSP Group, Inc.

Oracle, Oracle8i, and SQL*Plus are trademarks or registered trademarks of Oracle Corporation.

OSE is a registered trademark of OSE Systems.

PKZIP is a registered trademark of PKWARE, Inc.

Pro/CABLING and HARNESSDESIGN are trademarks or registered trademarks of Parametric Technology Corporation.

Quantic is a registered trademark of Quantic EMC Inc.

QUASAR is a trademark of ASM Lithography Holding N.V.

IMPORTANT - USE OF THIS SOFTWARE IS SUBJECT TO LICENSE RESTRICTIONS CAREFULLY READ THIS LICENSE AGREEMENT BEFORE USING THE SOFTWARE

This license is a legal "Agreement" concerning the use of Software between you, the end-user, either individually or as an authorized representative of the company purchasing the license, and Mentor Graphics Corporation, Mentor Graphics (Ireland) Limited, Mentor Graphics (Singapore) Private Limited, and their majority-owned subsidiaries ("Mentor Graphics"). USE OF SOFTWARE INDICATES YOUR COMPLETE AND UNCONDITIONAL ACCEPTANCE OF THE TERMS AND CONDITIONS SET FORTH IN THIS AGREEMENT. If you do not agree to these terms and conditions, promptly return or, if received electronically, certify destruction of Software and all accompanying items within 10 days after receipt of Software and receive a full refund of any license fee paid

END-USER LICENSE AGREEMENT

1. **GRANT OF LICENSE**. The software programs you are installing, downloading, or have acquired with this Agreement, including any updates, modifications, revisions, copies, and documentation ("Software") are copyrighted, trade secret and confidential information of Mentor Graphics or its licensors who maintain exclusive title to all Software and retain all rights not expressly granted by this Agreement. Mentor Graphics or its authorized distributor grants to you, subject to payment of appropriate license fees, a nontransferable, nonexclusive license to use Software solely: (a) (in machine-readable, object-code form; (b) for your internal business purposes; and (c) on the computer hardware or at the site for which an applicable license fee is paid, or as authorized by Mentor Graphics. A site is restricted to a one-half mile (800 meter) radius. Mentor Graphics' then-current standard policies, which vary depending on Software, license fees paid or service plan purchased, apply to the following and are subject to change: (a) relocation of Software; (b) use of Software, which may be limited, for example, to execution of a single session by a single user on the authorized hardware or for a restricted period of time (such limitations may be communicated and technically implemented through the use of authorization codes or similar devices); (c) eligibility to receive updates, modifications, and revisions; and (d) support services provided. Current standard policies are available upon request.

2. **ESD SOFTWARE**. If you purchased a license to use embedded software development (ESD) Software, Mentor Graphics or its authorized distributor grants to you a nontransferable, nonexclusive license to reproduce and distribute executable files created using ESD compilers, including the ESD run-time libraries distributed with ESD C and C++ compiler Software that are linked into a composite program as an integral part of your compiled computer program, provided that you distribute these files only in conjunction with your compiled computer program. Mentor Graphics does NOT grant you any right to duplicate or incorporate copies of Mentor Graphics' real-time operating systems or other ESD Software, except those explicitly granted in this section, into your products without first signing a separate agreement with Mentor Graphics for such purpose.

3. **BETA CODE**

   3.1. Portions or all of certain Software may contain code for experimental testing and evaluation ("Beta Code"), which may not be used without Mentor Graphics' explicit authorization. Upon Mentor Graphics' authorization, Mentor Graphics grants to you a temporary, nontransferable, nonexclusive license for experimental use to test and evaluate the Beta Code without charge for a limited period of time specified by Mentor Graphics. This grant and your use of the Beta Code shall not be construed as marketing or offering to sell a license to the Beta Code, which Mentor Graphics may choose not to release commercially in any form.

   3.2. If Mentor Graphics authorizes you to use the Beta Code, you agree to evaluate and test the Beta Code under normal conditions as directed by Mentor Graphics. You will contact Mentor Graphics periodically during your use of the Beta Code to discuss any malfunctions or suggested improvements. Upon completion of your evaluation and testing, you will send to Mentor Graphics a written evaluation of the Beta Code, including its strengths, weaknesses and recommended improvements.

3.3. You agree that any written evaluations and all inventions, product improvements, modifications or developments that Mentor Graphics conceives or makes during or subsequent to this Agreement, including those based partly or wholly on your feedback, will be the exclusive property of Mentor Graphics. Mentor Graphics will have exclusive rights, title and interest in all such property. The provisions of this subsection shall survive termination or expiration of this Agreement.

4. **RESTRICTIONS ON USE**. You may copy Software only as reasonably necessary to support the authorized use. Each copy must include all notices and legends embedded in Software and affixed to its medium and container as received from Mentor Graphics. All copies shall remain the property of Mentor Graphics or its licensors. You shall maintain a record of the number and primary location of all copies of Software, including copies merged with other software, and shall make those records available to Mentor Graphics upon request. You shall not make Software available in any form to any person other than your employer's employees and contractors, excluding Mentor Graphics' competitors, whose job performance requires access. You shall take appropriate action to protect the confidentiality of Software and ensure that any person permitted access to Software does not disclose it or use it except as permitted by this Agreement. Except as otherwise permitted for purposes of interoperability as specified by the European Union Software Directive or local law, you shall not reverse-assemble, reverse-compile, reverse-engineer or in any way derive from Software any source code. You may not sublicense, assign or otherwise transfer Software, this Agreement or the rights under it without Mentor Graphics' prior written consent. The provisions of this section shall survive the termination or expiration of this Agreement.

5. **LIMITED WARRANTY**

5.1. Mentor Graphics warrants that during the warranty period Software, when properly installed, will substantially conform to the functional specifications set forth in the applicable user manual. Mentor Graphics does not warrant that Software will meet your requirements or that operation of Software will be uninterrupted or error free. The warranty period is 90 days starting on the 15th day after delivery or upon installation, whichever first occurs. You must notify Mentor Graphics in writing of any nonconformity within the warranty period. This warranty shall not be valid if Software has been subject to misuse, unauthorized modification or installation. MENTOR GRAPHICS' ENTIRE LIABILITY AND YOUR EXCLUSIVE REMEDY SHALL BE, AT MENTOR GRAPHICS' OPTION, EITHER (A) REFUND OF THE PRICE PAID UPON RETURN OF SOFTWARE TO MENTOR GRAPHICS OR (B) MODIFICATION OR REPLACEMENT OF SOFTWARE THAT DOES NOT MEET THIS LIMITED WARRANTY, PROVIDED YOU HAVE OTHERWISE COMPLIED WITH THIS AGREEMENT. MENTOR GRAPHICS MAKES NO WARRANTIES WITH RESPECT TO: (A) SERVICES; (B) SOFTWARE WHICH IS LOANED TO YOU FOR A LIMITED TERM OR AT NO COST; OR (C) EXPERIMENTAL BETA CODE; ALL OF WHICH ARE PROVIDED "AS IS."

5.2. THE WARRANTIES SET FORTH IN THIS SECTION 5 ARE EXCLUSIVE. NEITHER MENTOR GRAPHICS NOR ITS LICENSORS MAKE ANY OTHER WARRANTIES, EXPRESS OR IMPLIED, WITH RESPECT TO SOFTWARE OR OTHER MATERIAL PROVIDED UNDER THIS AGREEMENT. MENTOR GRAPHICS AND ITS LICENSORS SPECIFICALLY DISCLAIM ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

6. **LIMITATION OF LIABILITY**. EXCEPT WHERE THIS EXCLUSION OR RESTRICTION OF LIABILITY WOULD BE VOID OR INEFFECTIVE UNDER APPLICABLE STATUTE OR REGULATION, IN NO EVENT SHALL MENTOR GRAPHICS OR ITS LICENSORS BE LIABLE FOR INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES (INCLUDING LOST PROFITS OR SAVINGS) WHETHER BASED ON CONTRACT, TORT OR ANY OTHER LEGAL THEORY, EVEN IF MENTOR GRAPHICS OR ITS LICENSORS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. IN NO EVENT SHALL MENTOR GRAPHICS' OR

ITS LICENSORS' LIABILITY UNDER THIS AGREEMENT EXCEED THE AMOUNT PAID BY YOU FOR THE SOFTWARE OR SERVICE GIVING RISE TO THE CLAIM. IN THE CASE WHERE NO AMOUNT WAS PAID, MENTOR GRAPHICS AND ITS LICENSORS SHALL HAVE NO LIABILITY FOR ANY DAMAGES WHATSOEVER.

7. **LIFE ENDANGERING ACTIVITIES**. NEITHER MENTOR GRAPHICS NOR ITS LICENSORS SHALL BE LIABLE FOR ANY DAMAGES RESULTING FROM OR IN CONNECTION WITH THE USE OF SOFTWARE IN ANY APPLICATION WHERE THE FAILURE OR INACCURACY OF THE SOFTWARE MIGHT RESULT IN DEATH OR PERSONAL INJURY. YOU AGREE TO INDEMNIFY AND HOLD HARMLESS MENTOR GRAPHICS AND ITS LICENSORS FROM ANY CLAIMS, LOSS, COST, DAMAGE, EXPENSE, OR LIABILITY, INCLUDING ATTORNEYS' FEES, ARISING OUT OF OR IN CONNECTION WITH SUCH USE.

8. INFRINGEMENT

8.1. Mentor Graphics will defend or settle, at its option and expense, any action brought against you alleging that Software infringes a patent or copyright in the United States, Canada, Japan, Switzerland, Norway, Israel, Egypt, or the European Union. Mentor Graphics will pay any costs and damages finally awarded against you that are attributable to the claim, provided that you: (a) notify Mentor Graphics promptly in writing of the action; (b) provide Mentor Graphics all reasonable information and assistance to settle or defend the claim; and (c) grant Mentor Graphics sole authority and control of the defense or settlement of the claim.

8.2. If an infringement claim is made, Mentor Graphics may, at its option and expense, either (a) replace or modify Software so that it becomes noninfringing, or (b) procure for you the right to continue using Software. If Mentor Graphics determines that neither of those alternatives is financially practical or otherwise reasonably available, Mentor Graphics may require the return of Software and refund to you any license fee paid, less a reasonable allowance for use.

8.3. Mentor Graphics has no liability to you if the alleged infringement is based upon: (a) the combination of Software with any product not furnished by Mentor Graphics; (b) the modification of Software other than by Mentor Graphics; (c) the use of other than a current unaltered release of Software; (d) the use of Software as part of an infringing process; (e) a product that you design or market; (f) any Beta Code contained in Software; or (g) any Software provided by Mentor Graphics' licensors which do not provide such indemnification to Mentor Graphics' customers.

8.4. THIS SECTION 8 STATES THE ENTIRE LIABILITY OF MENTOR GRAPHICS AND ITS LICENSORS AND YOUR SOLE AND EXCLUSIVE REMEDY WITH RESPECT TO ANY ALLEGED PATENT OR COPYRIGHT INFRINGEMENT BY ANY SOFTWARE LICENSED UNDER THIS AGREEMENT.

9. **TERM**. This Agreement remains effective until expiration or termination. This Agreement will automatically terminate if you fail to comply with any term or condition of this Agreement or if you fail to pay for the license when due and such failure to pay continues for a period of 30 days after written notice from Mentor Graphics. If Software was provided for limited term use, this Agreement will automatically expire at the end of the authorized term. Upon any termination or expiration, you agree to cease all use of Software and return it to Mentor Graphics or certify deletion and destruction of Software, including all copies, to Mentor Graphics' reasonable satisfaction.

10. **EXPORT**. Software is subject to regulation by local laws and United States government agencies, which prohibit export or diversion of certain products, information about the products, and direct products of the products to certain countries and certain persons. You agree that you will not export in any manner any Software or direct product of Software, without first obtaining all necessary approval from appropriate local and United States government agencies.

11. **RESTRICTED RIGHTS NOTICE**. Software has been developed entirely at private expense and is commercial computer software provided with RESTRICTED RIGHTS. Use, duplication or disclosure by the U.S. Government or a U.S. Government subcontractor is subject to the restrictions set forth in the license agreement under which Software was obtained pursuant to DFARS 227.7202-3(a) or as set forth in subparagraphs (c)(1) and (2) of the Commercial Computer Software - Restricted Rights clause at FAR 52.227-19, as applicable. Contractor/manufacturer is Mentor Graphics Corporation, 8005 Boeckman Road, Wilsonville, Oregon 97070-7777 USA.

12. **THIRD PARTY BENEFICIARY**. For any Software under this Agreement licensed by Mentor Graphics from Microsoft or other licensors, Microsoft or the applicable licensor is a third party beneficiary of this Agreement with the right to enforce the obligations set forth in this Agreement.

13. **CONTROLLING LAW**. This Agreement shall be governed by and construed under the laws of Ireland if the Software is licensed for use in Israel, Egypt, Switzerland, Norway, South Africa, or the European Union, the laws of Japan if the Software is licensed for use in Japan, the laws of Singapore if the Software is licensed for use in Singapore, People's Republic of China, Republic of China, India, or Korea, and the laws of the state of Oregon if the Software is licensed for use in the United States of America, Canada, Mexico, South America or anywhere else worldwide not provided for in this section

14. **SEVERABILITY**. If any provision of this Agreement is held by a court of competent jurisdiction to be void, invalid, unenforceable or illegal, such provision shall be severed from this Agreement and the remaining provisions will remain in full force and effect.

15. **MISCELLANEOUS**. This Agreement contains the entire understanding between the parties relating to its subject matter and supersedes all prior or contemporaneous agreements, including but not limited to any purchase order terms and conditions, except valid license agreements related to the subject matter of this Agreement which are physically signed by you and an authorized agent of Mentor Graphics. This Agreement may only be modified by a physically signed writing between you and an authorized agent of Mentor Graphics. Waiver of terms or excuse of breach must be in writing and shall not constitute subsequent consent, waiver or excuse. The prevailing party in any legal action regarding the subject matter of this Agreement shall be entitled to recover, in addition to other relief, reasonable attorneys' fees and expenses.

(10/99 rev B)