Celoxica

DK2

DK Design Suite User Guide

For DK2.0

Celoxica, the Celoxica logo and Handel-C are trademarks of Celoxica Limited.

All other products or services mentioned herein may be trademarks of their respective owners.

The product described in this document is subject to continuous development and improvement. All particulars of the product and its use contained in this document are given by Celoxica Limited in good faith. However, all warranties implied or express, including but not limited to implied warranties of merchantability, or fitness for purpose, are excluded.

This document is intended only to assist the reader in the use of the product. Celoxica Limited shall not be liable for any loss or damage arising from the use of any information in this document, or any incorrect use of the product.

The information contained herein is subject to change without notice and is for general guidance only.

Authors: SB

Document number: UM-2005-4.1

Table of contents

Table of contents

Table of contents

Table of contents

Table of contents

Table of contents

Conventions

A number of conventions are used in this document. These conventions are detailed below.

Warning Message. These messages warn you that actions may damage your hardware.

Handy Note. These messages draw your attention to crucial pieces of information.

Hexadecimal numbers will appear throughout this document. The convention used is that of prefixing the number with '0x' in common with standard C syntax.

Sections of code or commands that you must type are given in typewriter font like this:

```
void main();
```

Information about a type of object you must specify is given in italics like this:

copy ***SourceFileName DestinationFileName***

Optional elements are enclosed in square brackets like this:

```
struct [type_Name]
```

Curly brackets around an element show that it is optional but it may be repeated any number of times.

string ::= "{***character***}"

Assumptions

This manual assumes that you:

- have used Handel-C or have the Handel-C Language Reference Manual
- are familiar with common programming terms (e.g. functions)
- are familiar with MS Windows

Omissions

This manual does not include:

- instruction in VHDL
- instruction in the use of place and route tools

1. Getting started with DK

1.1 Starting DK

To start the DK, do one of the following:

- Select **Start**>**Programs**>**DK Design Suite**>**DK**
- Double-click on an existing DK workspace file (files with the extension `.hw`)
- Double-click the DK icon on the desktop

1.2 Creating a new file

1. Select **File**>**New**, and click the **Source File** tab.
2. Select the type of file you want to create in the left-hand pane. (Note that the default is a text file.)
3. Check the **Add to project** box if you want to add the file to an existing project. Select one of the projects in your current workspace from the drop-down box.
4. Type the name of the file in the **Filename** box. You do not need to add a file extension if you have set the file type.
5. Set the location (the directory path where the file is stored), by typing the path name in the box, or by selecting a directory by clicking the **...** button.
6. Press OK.
7. The code editor window opens.

1.3 Writing source code

You write Handel-C source code in the source code editor. Code is indented at the same level as the line above it and is syntax highlighted.

Having a file open in the source code editor does not mean that it is part of your project. The only files that will be compiled and built are those that you have added to your project.

1.3.1 Build configuration types

There are several default types of configuration that you can select from to build your application:

- Debug (default)
- Release
- Generic (This option is only available for library projects)
- VHDL (This option is not available in Nexus PDK.)
- Verilog (This option is not available in Nexus PDK.)
- EDIF (This option is not available in Nexus PDK.)

Debug mode is used to build a configuration that can be simulated and debugged on the PC. In debug mode, you can view the contents of registers and step through the program’s source code.

Release mode creates compiled code that has no debug messages and can be used in another program. Release mode can also be used for high-speed simulation.

Generic mode is used to create Handel-C intellectual property (libraries) which are not targeted at a particular output format. It creates compiled code that has no debug messages and can be used in another program. Generic mode can be linked for simulation, EDIF 2.0.0, Verilog IEEE Std 1364-1995 or VHDL 1987.

In EDIF mode, you get a list of gates, ready to be placed and routed on a device.

In VHDL mode, you get a collection of VHDL files, which can be simulated using any VHDL simulator (such as ModelSim) and synthesized and placed and routed using the appropriate RTL tools.

In Verilog mode, you get a collection of Verilog files, which can be simulated using any Verilog simulator (such as ModelSim) and synthesized and placed and routed using the appropriate RTL tools.

You can also define your own configuration types to store a particular set of project settings.

1.4 Project development sequence

The normal development sequence for a single-chip project is:

1. Create a new project.
2. Configure the project.
3. Add empty source code files to the project.
4. Create source code.
5. Link to any required libraries.
6. Set up the files for debug.

7. Compile the project for debug.
8. Use the debugger and simulator.
9. Optimize the project.
10. Compile the project for the target chip. (This step is not available in Nexus PDK.)
11. Export the target file to a place and route tool.
12. Place and route. There is no information on placing and routing within the DK documentation. Consult your place and route tool's documentation.

2. DK environment

The DK environment is a standard Windows development environment with dockable windows and customizable toolbars. The environment is in four main parts.

Workspace window	The area where you organize each project: the files you need, plus information about the target. When you start DK, the default position of the window is on the left.
Code editor window	Where you create and edit Handel-C source files. When you create or open a file, the default position of the window is on the right.
Output window	The area that displays error messages and warnings when you compile a file. The default position of the window is at the bottom of the screen. The output window has tabs for build messages and debug messages.
Debug windows	Windows which show information when you simulate the operation of a compiled program. The **View>Debug Windows** command determines which windows are displayed. The simulation steps the program through clock cycles, and allows you to look at the contents of any variables that are in scope. These are displayed in the Variables window. You can select variables to display in the debug Watch window. The default position of the Watch window is the bottom left-hand corner of the screen. The call stack (the route by which you have called a function) is displayed in the Call Stack window. You can see clock cycles and current executing threads in the Clocks/Threads window.

2.1 Workspace window

The Workspace window contains workspaces and projects.

A workspace is allows you to organize the files that you need for each project. You would generally use one workspace per system (a system describes the hardware configuration that you are targeting).

A project consists of everything you need to create one or more netlist files ready to be placed and routed on a device, together with the project settings. Project settings provide information about where the files for the project are stored, the target chip for the project, how the compilation will work, and optimization requirements.

The Workspace window has two views:

- File view
- Symbol view

2.1.1 File view

File view shows the workspace, its projects, and their source files and folders. The current project name is in bold.

File view shows the structure of files in the project, not how they are stored on disk. It allows you to set up dependencies (what files are needed for this project, and what files or projects they depend upon) and to manage your project.

- Double-click on a source file name to open the file in the code editor. Double-clicking on anything else expands or contracts that branch of the workspace tree.
- Right-click on a file name or directory to display a menu of commonly used options.

Context menu - File View window

The context menus in the **File View** window are accessed by right-clicking on a file or project.

Context menu for files

Item	Description
Open	Opens file in Code Editor window
Compile	Compiles file
Delete	Removes file from project
Settings	Opens **Project Settings** dialog. Allows you to specify file settings.
Properties	Opens **Properties** dialog. Displays information and allows you to change the language specified for the file.

Context menu for projects

Item	Description
Build	Builds the selected project
Clean	Deletes all the files that are created by building the project (doesn't affect source files)
New Folder	Allows you to specify the name of a new folder, and the extensions of the files associated with it
Add files to Folder	Allows you to add files to the project
Set as Active Project	Sets selected project to be the active one
Settings	Opens **Project Settings** dialog
Properties	Opens **Properties** dialog

File view icons

DK workspace (.hw file)

DK system project (.hp file)

DK board project (.hp file)

DK chip project (.hp file)

DK core project (.hp file)

Library project (.hp file)

Handel-C source file (.hcc file)

Handel-C header file (.hch file)

C++ source file (.cpp file)

ANSI-C source file (.c file)

C/C++ header file (.h file)

Text file (.txt file)

Folder

Folder (open)

2.1.2 Symbol view

A symbol is a logical or architectural construct that you define such as a function, variable, macro, `typedef` or enum. Symbol view allows you to see the logical content of a project.

- To create the symbol view, build the project with the option **Save browse info (-b)** enabled in the project settings (**Linker** tab, or **Library** tab for a library project). This option is set by default in the Debug configuration.
- To see the symbol view, select the **Symbol View** tab in the Workspace window.

Symbol view shows a tree of icons representing the logical and architectural components. Each icon is identified by its definition and use (references). External symbols (external variables and function names) appear in alphabetical order. Local symbols appear in alphabetical order within the function or procedure where they are defined.

Double-click on a symbol to expand it, or (if it is not expandable) to open the relevant source code file with the appropriate line tagged.

Symbol view icons

Icon	Meaning
	Shared function, procedure or expression
	In-line function or macro
	Variable
	Memory (RAM, ROM, WOM or MPRAM)
	Channel (chan, chanin or chanout)
	External interface
	Semaphore (sema)
	Signal
	Stacked position containing the related object (e.g. recursive macro)
	Position in the file containing the definition of the object

2.1.3 Code editor window

The code editor is a simple editor that resides in its own window. If you right-click in the code editor window, you get a context-sensitive menu.

2.1.4 Code editor icons

	Current active point
	Other statements executed in current thread on current clock cycle
	Active point in different thread
	Position of current error/browse symbol
	Enabled breakpoint(s) on this line
	Disabled breakpoint(s) on this line
	Enabled and disabled breakpoint(s) on this line
	Bookmark on this line

2.1.5 Context menu - code editor window

Item	Description
Undo	Removes the last word or line break that you typed
Redo	Restores a word or line break (after using **Undo**)
Cut	Cuts selected text
Copy	Copies selected text
Paste	Pastes text copied from elsewhere
Select All	Selects everything in the Code editor window
Toggle Bookmark	Allows you turn bookmarks on and off
Insert Breakpoint	Allows you to specify which lines of code the simulator will pause at

2.1.6 Syntax colour codes

The syntax in a displayed file is colour coded.

The default colour codes are:

green:	comments
blue:	Handel-C and supported C/C++ keywords
red:	unsupported C/C++ keywords
brown:	number
brown:	string
purple:	operator

You can change the colour codes by selecting the **Format** tab from the **Tools**>**Options** dialog box.

2.2 Output window icons

	Information		User assert statement
	Warning about your program		
	Error in your program		Position stack
	Internal error in the compiler		Position

2.3 Debugger interface

The debugger interface consists of the debug windows and menu commands, and their associated buttons. When you start a simulation, the **Debug** menu appears. You use the **Debug** menu commands to control the simulation.

Debug information is presented in the following windows. To open or close windows, use the following shortcuts or use the **View**>**Debug Windows** menu options.

Window	Shortcut	Function
Code editor	Appears by default	The editor window for the source code that you are debugging. Its title will be the file name. The code is marked by debug symbols to show the current execution points and breakpoints.
Call Stack	Alt+7	Shows the calling path to the current function.
Clocks/Threads	Alt+5	Identifies all current threads, and allows you to select one to follow. Also identifies each clock in use, and allows you to view its definition in the code.
Variables	Alt+4	Shows the variables used in the latest statements in the current thread, and those local to the current macro or function.
Watch	Alt+3	Showing the contents of variables that you select. You select the variables to show on four separate tabs

Buttons

Restart			Stop debugging
Break			
Step into		Alt+7	Show/hide the Call Stack window
Step over		Alt+5	Show/hide the Clock/Threads window
Step out		Alt+4	Show/hide the Variables window
Run to cursor		Alt+3	Show/hide the Watch window
Advance		Alt+0	Show/hide the Workspace window

Icons

Clock in Clocks/Threads window

Thread in Clocks/Threads window

2.3.1 Call Stack window

During debug the **Call Stack** window lists the functions and macro procedures called on the way to the current function. The current function or macro procedure appears at the top of the list, followed by those that have not yet completed. You can open the **Call Stack** window by selecting **View**>**Debug Windows**>**Call Stack** or clicking the **Call Stack** window button 

2.3.2 Clocks/Threads window

The **Clocks/Threads** window shows a tree view of the simulators, clocks and threads in operation during debug. Entries for the current clock and thread appear in bold type.

- To open the Clocks/Threads window, select **View**>**Debug windows**>**Clocks/Threads**, or press Alt + 5.

Details are shown in three columns.

Clock/Thread Identifies each clock and the threads that are executing on it.

 Clock entry is in the form ***clockno line***
clockno is the number used by the simulator to identify the clock
line is the source file name and line where the clock is defined.

To view the definition, right-click on the clock icon and select **Show Definition** from the shortcut menu.

 Thread entry is in the form ***threadno context***
threadno is the number used by the simulator to identify the thread
context indicates the source code context in which the thread executes.

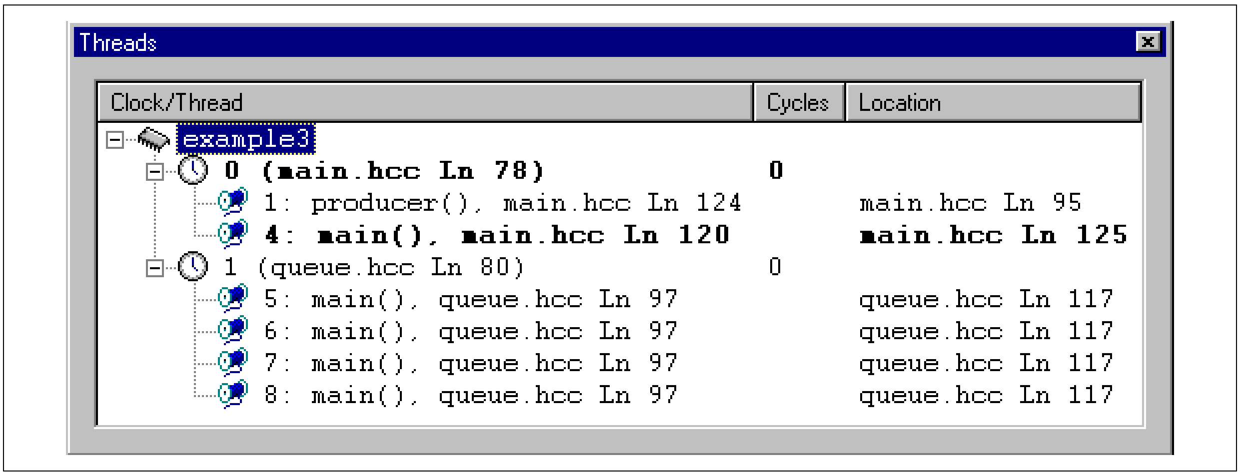
Right-click on the thread icon to display a menu with two options:

Show Location to view the source code for the thread

Follow Thread to make the selected thread the current thread

Cycles Shows the number of cycles executed for each clock.

Location Shows the source code file name and line number currently executing in the thread.



SAMPLE CLOCKS / THREADS WINDOW

2.3.3 Variables window

The Variables window shows the variables that are important in the program's current context. When their values change, the colour changes from black to red. Only the last value to change will be shown in red. You can open the Variables window by selecting **View**>**Debug Windows**>**Variables** or clicking the Variables window button

You can change the base that variables are displayed in by right-clicking the Variables window and selecting a new base. Binary format variables are displayed with leading zeroes. You can also change the default base for variables in the Variables window: select **Tools**>**Options**>**Debug**, and set the required base in the **Base for numbers** box.

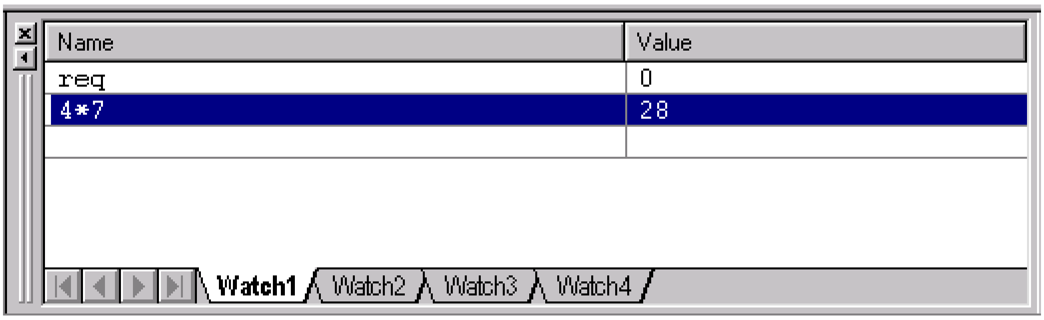
The default maximum number of elements displayed in the Variables window is 16. To change this, select **Tools**>**Options**>**Debug**, and change the number in the **Maximum number of visible elements** box. If you increase the number of elements, the simulation will be slower.

The Variables window has two tabs, **Auto** and **Locals**.

- The **Auto** tab shows variables that have been automatically selected. These are variables used in the current statement and in the previous statement. (If you have just swapped threads, the "previous statement" will be the last one you looked at in the other thread.)
  Variables that have changed since the previous step are shown in red. The **Auto** tab also displays return values when you come out of or step over a function. If you switch threads, you will see variables from the previous step in the other thread.
- The **Locals** tab shows the variables that are in scope in the current function or macro.

2.3.4 Watch window

The Watch window has four tabs:



Each goes to a different Watch window. You can select variables to be displayed in each window, and look at their values at any breakpoint or as you step through the program.

The default maximum number of elements displayed in the Watch window is 16. To change this, select **Tools**>**Options**>**Debug**, and change the number in the **Maximum number of visible elements** box. If you increase the number of elements, the simulation will be slower.

You can add a variable to the Watch window by typing its name. You can delete a variable from the Watch window but selecting its name and deleting it.

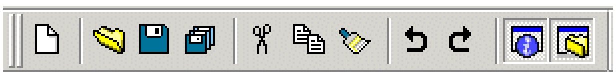
You can change the base that variables are displayed in by right-clicking the Watch window and selecting a new base. Binary format variables are displayed with leading zeroes. You can also change the default base for variables in the Watch window: select **Tools**>**Options**>**Debug**, and set the required base in the **Base for numbers** box.

The Watch window has an expression evaluator. If you type in an expression, it will be evaluated and the result will be displayed. It cannot display expressions containing: function calls, `let`, `select`, `trysema`, strings, &, `assert`.

2.4 Toolbars

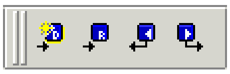
When you start DK, toolbars appear under the menu bar.

Standard toolbar

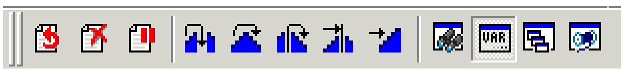


Build toolbar

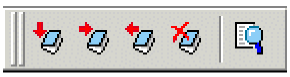
Browse toolbar



Debug toolbar



Edit toolbar



2.4.1 Standard toolbar buttons

The buttons on the standard toolbar give a subset of options from the File, Edit and View menus.

Toolbar icons

New ...	Show Workspace window
Open	Show Output window
Save	Show properties
Save All	Compile one file
Print	Build project
Cut	Stop a build in progress
Copy	Run program in simulator
Paste	Add or remove breakpoint
Undo	Show the About box
Redo	
Find	
Find in files	

2.4.2 Status bar

The status bar is visible at the bottom of the DK window. It displays:

- Information about items when the mouse is over them
- The current line and column number within the current code editor window
- Status and progress messages
- Keyboard states
  - CAP : Caps lock on
  - NUM : Num lock on
  - SCRL : Scroll lock on
  - OVR : Overwrite on (i.e. insert key pressed)

You can toggle its display by selecting **View**>**Status bar**.

2.5 Customizing the DK GUI

2.5.1 Customizing windows

The DK user interface has standard scrollable windows.

You can customize:

- The position and size of the workspace, code editor, output and debug windows. The settings will affect all DK projects.
- How document windows are laid out (this is specific to each workspace)

Re-sizing windows

Document windows are movable within the DK window. You can resize them and drag them about.

Docking windows can either be docked at one of the window margins, or can float above the other windows.

- To float a docked window, double-click its border.
- To dock a floating window, either double-click its border, or drag its title bar to a docking position.

Splitting windows

You can split a text window in two ways:

- Use the **Split** command on the Window menu
- Drag the split box (shown in the graphic below). It is the small box immediately above the vertical scroll bar in the text window:

Full screen display

The **Full Screen** command on the **View** menu displays the code editor pane at maximum size. The normal menu bars and toolbars are not visible. To return to a normal view, click the Close Full Screen button .

2.5.2 Customizing toolbars

The **Command** tab in the **Customize** dialog allows you to add or remove buttons on any toolbar. The right-hand pane displays the buttons available.

Adding or removing buttons on a toolbar

Select **Tools**>**Customize Toolbars**, and then select the **Toolbars** tab.

To add a button to a toolbar, select the button from the Commands right-hand pane and drag it to the toolbar.

To remove a button from a toolbar, drag the button off the toolbar.

Restoring a toolbar

To reset a toolbar to its previous state, select **Tools**>**Customize Toolbars** and then select the **Toolbars** tab. Select the toolbar name in the Toolbars list and click the **Reset** button.

Placing toolbars

The toolbars in DK are dockable. They can be docked at one of the edges of the DK window, or they can float.

- You can change a toolbar from docked to floating and back by double clicking on it.
- You can move a toolbar by dragging the title bar or the double bar.

Changing toolbar appearance

The **Toolbars** tab in the **Customize** dialog allows you to change the display of toolbars.

Check a toolbar in the toolbar pane to display it, uncheck it to hide it.

Show Tooltips	Check this to popup the purpose of a button when your mouse cursor is over it
Cool Look	Check this to make the buttons appear two-dimensional
Large Buttons	Check this to increase the button size

2.5.3 Customizing menus

The **Command** tab in the **Customize** dialog allows you to add buttons to the toolbar and menus to the menu bar. The right-hand pane displays the buttons and menu commands available.

Select **Tools**>**Customize Toolbars**, and then select the **Command** tab.

To add a menu to the menu bar:

1. From the Categories list select **Menu**.
2. Select the menu name from the right-hand list and drag it to the menu bar.

If you drag a menu name to a toolbar, it appears as a button. If you drag it to an empty area, it appears as a new floating window.

To remove a menu from the menu bar, drag the menu name off the menu bar.

3. Commands

3.1 File menu

Command		Shortcut	Function
	**New...**	Ctrl+N	Display the **New** dialog to create: • A project • A file • A workspace
	**Open...**	Ctrl+O	Display the **File Open** dialog
	**Save**	Ctrl+S	Save the active document
	**Print**	Ctrl+P	Print the active document
	Save As...		Save the active document under a new name
	Save All		Saves all active documents.
	Page Setup		Set up for printing
	Open Workspace		Display the **Workspace Open** dialog
	Close Workspace		Close the current workspace
	Save Workspace		Save the current workspace
	Recent Files>		List of recently used files. Select one to open it.
	Recent Workspaces>		List of recently used workspaces. Select one to open it.
	Exit		Quit DK

3.1.1 New dialog (File>New)

The **New** dialog allows you to create

- new files
- new projects
- new workspaces

3.2 Edit menu

Command		Shortcut	Function
	Undo	Ctrl+Z	Reverse a recent change to the active document or to the workspace
	Redo	Ctrl+Y	Reverse a recent undo
	Cut	Ctrl+X	Copy the current selection to the clipboard and delete it
	Copy	Ctrl+C	Copy the current selection to the clipboard
	Paste	Ctrl+V	Copy the contents of the clipboard to the current selection
	Delete	Del	Delete the current selection
	Find	Ctrl+F	Find a string or regular expression in the current file. Use F3 to Find next occurrence, Shift F3 to find previous occurrence.
	Find in files...		Find a string or regular expression in selected files
	Replace	Ctrl+H	Replace one string or regular expression with another in current file
	Bookmarks>		Set, remove or move through bookmarks in the document
	Breakpoints...		Display the project's breakpoints dialog box
	Browse>		Find definitions and references for variables or other symbols in the document

3.2.1 Find commands

DK has simple **Find** and **Replace** commands that allow you to search for text in the current file, and the **Find in Files** command, which allows you to search for a string in all the files in a directory. The shortcut F3 finds the next occurrence, and Shift F3 finds the previous occurrence.

The output from **Find in Files** can be sent to two different window panes, allowing you to view the results of two searches. To choose which pane is selected, check or uncheck the **Output to pane 2** box in the **Find in Files** dialog.

These searches work line by line. Therefore you cannot match text that spans more than one line.

You can also search using regular expressions. To do this, check **Regular expression** in the **Find** or **Find in Files** dialog box.

Finding using regular expressions

You can search files for text by using regular expressions. To do this, check **Regular expression** in the **Find** or **Find in Files** dialog box. You can use any of the expressions listed below.

Regular expression	Description
(***x***)	The characters or expressions between the parentheses.
.	(Period.) Any single character.
^	Start of line.
$	End of line.
\t	Tab character.
x|***y***	A match for either ***x*** or *y*. For example, a(team|class) will match either ateam or aclass.
x*	Zero, one or many copies of ***x.*** For example, ba*c matches bac, baac, baaac and bc.
x?	None or one ***x***. For example, ba?c matches bac or bc.
x+	At least one or more of ***x.*** For example, ba+c matches bac, baac, baaac, but not bc.
[***xyz***] [***x***-***y***]	Matches one character from the set in the brackets. Use a dash (-) to include all characters in a range; for example, [_A-Za-z] matches an underscore or any letter, and [_A-Za-z][_A-Za-z0-9]* matches an alphanumeric string that can include underscores. Use [***xyz***-] or [-***xyz***] if you want to include a dash in the set. If you need a] in the set use [] ***xyz***].
[^***xyz***]	Matches one character that is not in the brackets. For example, x[^0-9] matches xa, but not x0 or x2.
x	Matches the character *x*, even if *x* is one of the magic characters ^\$[].*+? listed above. For example, ^pig matches pig at the start of a line, but \^pig matches the string ^pig anywhere on a line.

3.2.2 Bookmarks

The **Bookmarks** submenu in the **Edit** menu allows you to set and clear bookmarks within files.

Once you have set one or more bookmarks in a file, you can move through the bookmarks by selecting **Next Bookmark** (F2) or **Previous Bookmark** (Shift F2).

Setting bookmarks

1. Select the line where you wish to place the bookmark.
2. Press the Toggle Bookmark button

   OR

   Right-click the line and select **Toggle Bookmark** from the shortcut menu

   OR

   Select **Edit**>**Bookmarks**>**Toggle Bookmark** (Ctrl F2).

Moving to a bookmark

To move forward through the bookmarks	Select **Edit**>**Bookmarks**>**Next Bookmark** (F2) or press the Next Bookmark button
To move backwards through the bookmarks	Select **Edit**>**Bookmarks**>**Previous Bookmark** (Shift F2) or press the Previous Bookmark button

Clearing a bookmark

1. Select the line where you wish to clear the bookmark.
2. Press the Toggle Bookmark button

   OR

   Right-click the line and select Toggle Bookmark from the shortcut menu

   OR

   Select **Edit**>**Bookmarks**>**Toggle Bookmark** (Ctrl + F2).

Clearing all bookmarks in a file

To clear all bookmarks:

- Select **Edit**>**Bookmarks**>**Clear All Bookmarks** (Ctrl + Shift + F2)

  OR

- Press the Clear All Bookmarks button

3.2.3 Breakpoints dialog

The **Breakpoints** dialog appears when you select the **Edit**>**Breakpoints** command. The dialog gives a list of currently set breakpoints. You can:

- View all breakpoints
- Delete breakpoints
- Make a breakpoint conditional
- Disable or enable a breakpoint
- View code where the breakpoint is set
- Add a (duplicate) breakpoint
- Edit a breakpoint

View all breakpoints

When the dialog box opens, it displays a list of all current breakpoints, identified by file name and line number.

Deleting breakpoint(s)

Select a breakpoint in the breakpoint list and click **Remove**. To delete all breakpoints, click **Remove All**.

Making a breakpoint conditional

On condition: Select the breakpoint in the breakpoint list and enter the condition on which it will be active in the **Break when** box. This condition can be any valid Handel-C expression. For example, `y == 4` or `x[7]! = 0`. Note that statements are not allowed, so you cannot use `y = 4`.

On repetition: Select the breakpoint in the breakpoint list and enter the number of times that it must be passed before it is active in the **Break after** box. For example, if you enter '5', the breakpoint will be triggered on the 6th pass through the code.

You can also use the **Break after** box to specify how many times a condition should be passed before it is active.

Disabling and enabling breakpoints

To disable a breakpoint, clear the box by its entry in the list of breakpoints. To enable it, check the box.

View code where breakpoint is set

Select the breakpoint in the list and click the **Edit code** button.

Add a (duplicate) breakpoint

Select the blank box at the end of the breakpoint list. Type the file name and line number (separated by a comma) in the **Break at** box. This allows you to have two breakpoints on the same line with different conditions.

Edit a breakpoint

Select the breakpoint in the list. Edit the file name and line number in the **Break at** box. The file name and line number must be separated by a comma, e.g. `parmult.hcc,112`.

3.2.4 Using browse commands

The **Edit**>**Browse** command allows you to find definitions of and references to selected variables or other symbols. If you make a change to a variable, this is a quick way of finding everywhere that the variable is used.

To find the definition of a variable or other symbol

1. Select the symbol name in an edit window.
2. Select **Edit**>**Browse**>**Go to Definition** or click the button.

To find the first reference to a variable or other symbol

1. Select the symbol name in an edit window.
2. Select **Edit**>**Browse**>**Go to Reference** or click the button.

To move through the references to and definitions of a variable or other symbol

1. Select the symbol name in an edit window.
2. To move forward, select **Edit**>**Browse**>**Next Definition Reference** or click the button.
3. To move backward, select **Edit**>**Browse**>**Previous Definition Reference** or click the button.

Browse commands summary

If you select a symbol name in a source file, you can use the browse commands and buttons to find its definitions and references in all the files used in a project. If the symbol name is defined more than once, a **Resolve Ambiguity** dialog appears, giving you the list of symbols with that name, and which files they are in.

Button	Command	Function
	Go to Definition	Jump to the source code line where the variable is defined
	Go to Reference	Jump to the first source code line where the variable is used
	Previous Definition/Reference	Jump to the previous definition or reference
	Next Definition/Reference	Jump to the next definition or reference

3.3 View menu

Command		Shortcut	Function
	Status bar		Show/hide the status bar
	Full screen		Show the code editor pane at maximum size
	Workspace	Alt+0	Show/hide the Workspace window
	Output	Alt+2	Show/hide the Output window
	Debug Windows>		Control the windows in the debugger
	Properties	Alt+Enter	Display the **Properties** dialog for the current file or selection

3.4 Project menu

Command	Shortcut	Function
Set Active Project>		Select a project from the workspace to make current
Add to Project>		Add a file or folder to the project
Dependencies...		Select projects on which the project depends
Settings...	Alt+F7	Open the **Project Settings** dialog box to do one of these tasks: • Use the logic estimator • Create independent settings for a file • Set the output directory for generated files • Set preprocessor settings
Insert Project into Workspace...		Add a project to the workspace

3.4.1 Project settings

Project settings define how your files and projects are compiled and built. Select **Project**>**Settings** to see the **Project Settings** dialog box. The different settings are available via tabs. The tabs available will depend on the project type. For example, the **Library** tab is only available for a library-type project.

The tabs available are:

- General
- Chip
- Preprocessor
- Compiler
- Optimizations
- Linker
- Debugger
- Build commands
- Library

If you can't see the tab you want, then scroll the tabs by clicking on the arrows at the end of the tabs. Some tabs may only be visible if you have selected a Handel-C file in the left window.

3.5 Build menu

Command		Shortcut	Function
	Compile	Ctrl+F7	Run the compiler on the active document (which must be a Handel-C code file), to generate its .hco file.
	Build *project*	F7	Build this project: run the compiler on all .hcc files that are newer than their object (.hco) files, then run the linker on the object files to make the .dll, .hcl, EDIF, Verilog or VHDL files. (EDIF, VHDL and Verilog are not available in Nexus PDK.)
	Stop Build	Ctrl+Break	Cancel a build in progress.
Rebuild All			Rebuild all files in this project: like Build, except that all .hcc files are compiled.
Clean			Delete all the files that are created by Build.
Start Debug			Pop-up menu giving three options:
	Go	F5	(Build project if not built.) Run the simulator at full speed until a breakpoint or other stop is reached.
	Step Into	F11	(Build project if not built.) Run to the first statement in the function or macro invoked in the current line. If the current line is not a function or macro invocation, run to the next statement.
	Run to Cursor	Ctrl+F10	(Build project if not built.) Run to the line containing the text cursor.
Set Active Configuration			Choose the active build configuration for the current project.
Configurations...			Add or remove configurations.

Selecting a configuration

Select **Set Active Configuration** from the **Build** menu. The **Set Active Project Configuration** dialog appears. Select the configuration that you wish to use, and click **OK**.

3.6 Debug menu

The Debug menu appears when you build a project in Debug mode and then start the debugger by pressing F5 (Go) or F11 (Step into).

Command	Shortcut	Function
Go	F5	Runs the simulator until it reaches a breakpoint or other stop.
Restart	Ctrl+Shift+F5	Runs the simulator, starting at the first line of the program.
Stop Debugging	Shift+F5	Stops the simulation.
Break		Pauses the simulation when it is running.
Step Into	F11	Moves to the end of the next clock edge executed within the current thread, or to the next function call, or to the next breakpoint. If the current line is a function or macro call, it runs to the end of the clock cycle invoked by the call.
Step Over	F10	If the current line is a function or macro call, it runs to the end of the clock cycle after the function (steps over the function). Else as for **Step Into**.
Step Out	Shift+F11	Executes the rest of a function or macro, and steps to the end of the clock cycle after the line which invoked the function (steps out of a function).
Run to Cursor	Ctrl+F10	Runs until the line containing the text cursor is reached.
Advance	Ctrl+F11	Moves forward a single execution point rather than a complete clock cycle.

3.7 Tools menu

Command	Shortcut	Function
Source Browser	Alt+F12	Use the source browser dialog to find definitions and references for variables and functions in your code.
Customize Toolbars...		Customize your copy of DK: change the display of toolbars, and add menus and buttons to toolbars and the menu bar.
Keyboard Shortcuts...		Redefine the available keyboard shortcuts.
Options...		Set options for: Editor; Tabs; Debug; Format; Workspace; Directories

3.7.1 Source browser

The Source Browser command allows you to search for names of variables and functions in your code. It directs you to their definition and lists references to them.

1. Build your project (Press F7). You will need to re-build if you have changed your code since a previous build.
2. From the **Tools** menu, select the **Source Browser** command.
3. In the **Browse** dialog box, enter the symbol name to view its definition and references.

You can also browse for definitions and references using symbol view.

3.7.2 Customize Toolbars... command

The **Customize Toolbars...** command on the **Tools** menu allows you to change the DK user interface in the following ways:

- Change the appearance of toolbars
- Add or remove toolbar buttons
- Add or remove menus and buttons on the menu bar

3.7.3 Tools Options dialog

Command	Function
Editor	Set the window options for the editor. Define when files are saved.
Tabs	Define how tabs are handled and whether Auto-Indent is used.
Debug	Set the default base used to display numbers in the debug windows. This information is over-ruled by the Handel-C base specification.
Format	Define the colour and font of text and markers in windows.
Workspace	Set the number of recently opened workspaces in the workspace list.
Directories	Set the directories that will be searched for and library files used in projects.

Editor tab (Tools options)

Item	Function when checked
Selection margin	Use a selection margin in the editor window to enable you to view breakpoints and debug symbols to the left of your source code.
Drag and drop text editing	Edit by selecting an area, and dragging it to a new position
Save before running tools	Save files before running tools defined in the **Tools** menu
Prompt before saving files	Ask before saving

Format tab (Tools options)

Command	Function
Category	Select window type(s) to modify
Font	Select font to display text in
Size	Select display font size
Colours	Select text type to modify:
	Foreground: Set foreground colour
	Background: Set background colour
Sample	Display sample text in selected settings
Reset All	Return to default settings

Workspace tab (Tools options)

Command	Function
Default workspace list	Set number of recent workspaces listed in the **File**>**Recent Workspaces** command.

Tabs tab (Tools options)

Command	Function
File type	Define settings for specified file types or define default settings.
Tab size	Equivalent number of spaces per tab
Insert spaces/Keep tabs	Select whether to use spaces or tabs in file. Existing spaces/tabs will not be changed.
Auto indent	Check to auto-indent text to above line's indent

3.7.4 Debug tab (Tools options)

Command	Function
Base for numbers	Select default display base in debug windows
Maximum number of visible elements	Specify maximum number of array or memory elements to be shown in Watch and Variables windows during simulation. Default is 16. If you increase the number of elements, the simulation will be slower.

Directories tab (Tools options)

Command	Function
Show directories for	From the dropdown list, select **Include files** path list or **Library modules** path list. Add or remove directory paths to search for files. You can select directories individually, or enter multiple paths separated by commas.

3.8 Window menu

The Window menu allows you to control the size and display of editing windows.

Command	Function
New window	Create a copy of the current window
Split	Split the window into two or four views
Close	Close the current window
Close All	Close all windows
Cascade	Cascade all open windows with title bars visible
Tile Horizontally	Display all windows, splitting the viewing area horizontally
Tile Vertically	Display all windows, splitting the viewing area vertically
Arrange Icons	Arrange minimized window icons at the bottom of the viewing area
Windows...	List and control the open edit windows
List of files	A list of files currently open for editing appears after the Windows option. The file currently selected is marked by a tick.

3.8.1 Windows dialog

The **Windows** dialog (**Window**>**Windows**) gives the names of all open edit windows. You can make one of them the current window, or select a group of windows to be saved, closed or tiled.

3.9 Help menu

Command	Shortcut	Function
Help Topics	F1	List the Help topics
About DK Design Suite	-	Give version etc.

3.10 Keyboard shortcuts

This table gives a list of the default keyboard shortcuts. You can change them using the **Tools**> **Keyboard Shortcuts** command.

Command	Shortcut	Function
File		
New...	Ctrl+N	Display the **New** dialog to create: • A project • A file • A workspace
Open...	Ctrl+O	Display the **File Open** dialog
Save	Ctrl+S	Save the active document
Print	Ctrl+P	Print the active document
Edit		
	Alt+drag	Select rectangular area
Undo	Ctrl+Z	Reverse the most recent change to the active document or to the workspace
Redo	Ctrl+Y	Reverse the most recent undo
Cut	Ctrl+X	Copy the current selection and delete it
Copy	Ctrl+C	Copy the current selection to the clipboard
Paste	Ctrl+V	Copy the clipboard to the current selection
Delete	Del	Delete the current selection
Find	Ctrl+F	Find string or regular expression
	F3	Find next string or regular expression
	Shift+F3	Find previous string or regular expression
Replace	Ctrl+H	Replace found selection
Bookmarks...	Alt+F2	Display the project's bookmarks dialog box
	Ctrl + F2	Toggle selected bookmark on or off
	F2	Go to next bookmark
	Shift + F2	Go to previous bookmark
	Ctrl Shift + F2	Clear all bookmarks
Breakpoints...	Alt+F9	Display the project's breakpoints dialog box

3. Commands

View

Workspace	Alt+0	Hide or show the **Workspace** window
Output	Alt+2	Hide or show the **Output** window
Debug windows:		
Watch	Alt+3	Hide or show the **Watch** window
Call Stack	Alt+7	Hide or show the **Call Stack** window
Variables	Alt+4	Hide or show the **Variables** window
Clocks/ Threads	Alt+5	Hide or show the **Clocks/Threads** window
Properties	Alt+Enter	Display the **Properties** dialog for the current document or selection

Project

Settings...	Alt+F7	Shows the **Project Settings** dialog box

Build

Compile	Ctrl+F7	Compiler selected file
Build	F7	Build this project

Debug

Go	F5	Run the simulator at full speed (until a breakpoint etc.)
Restart	Ctrl+Shift+ F5	Run the simulator from the beginning
Stop Debugging	Shift+F5	Stop the simulation
Step Into	F11	Run to the first statement in the function invoked in the current line. If the current line is not a function invocation, just run until the next statement
Step Over	F10	Run until the start of the next statement
Step Out	Shift+F11	Run until the start of the statement after the line which invoked the current function
Run to Cursor	Ctrl+F10	Run until the line containing the text cursor is reached
Advance	Ctrl+F11	Advance a partial clock cycle, to the next code line.

Tools

Source Browser	Alt+F12	Show a symbol browser dialog box

3. Commands

Help		
Help Topics	F1	List the Help topics
About		Gives the DK and compiler versions
Output Window		
	Double click	Takes you to line in source code
	F4	Next error
	Shift+F4	Previous error
Windows control		
	F6	Next pane in split window
	Shift+F6	Previous pane in split window

4. Project development

4.1 Project types

When you start a new project, you need to define its type. A new project may be:

a chip	Not targeted to a particular product. Will not use device-specific resources. Cannot be built as Generic mode.
a board	Allows you to have multiple chip projects within a board project. Targeted to chips defined within board. Cannot be built as Generic mode.
a system	Allows you to have multiple board projects within a system project. Targeted to chips defined within boards. Cannot be built as Generic mode.
a core	A discrete piece of code, compiled to a specific architecture, which may be used as part of a larger design. Cannot be built as Generic mode.
a library	Pre-compiled Handel-C code that may be re-used or sold elsewhere. If built in Generic mode can be rebuilt to target EDIF, VHDL or Verilog. If built in other mode can only be linked with projects in that format.
a pre-defined chip, system or board	Targeted to a known product. These systems will be optimized for that product, and should only be placed and routed onto that product. Cannot be built as Generic mode.

Common pre-defined project types are supplied with DK.

4.1.1 Creating a project

1. Select **New** from the **File** menu.
2. Select the **Project** tab in the dialog that appears.
3. Enter the name and location (path name for the directory that it will be stored in) for your project. You can look for a directory by clicking the **...** button to the right of the **Location** box.
4. Select the appropriate project type from the types listed in the **Project** pane.
5. Click **OK**.

By default, a new workspace is created for your project in the same directory as the project. Workspace files have **.hw** extensions. Project files have **.hp** extensions.

4.2 Managing your project files

You can order the files within your project into folders. These folders are only used to organize the files. They do not exist as folders on your hard disk and have no effect on your directory structure.

1. Select **Project**>**Add to Project**>**New Folder**
2. Type the name of the folder in the dialog box that appears.
3. Type the extension for the file types it should contain. You can leave the box blank.
4. Click **OK**. A new folder appears in the **File View** window.
5. Drag the files that you wish to move across to the folder.

4.2.1 What files are generated for a project?

The table below lists the files built for a workspace `WSpace.hw`, containing a project `Proj`, consisting of one Handel-C file `Code.hcc` that has been built for debug. `Code.hcc` #includes the file `Incl.hch`. Output and Intermediate files will be stored in the `Debug` folder.

Directory	File name	File type
Workspace directory	`WSpace.hw`	Workspace
	`WSpace.pref`	Contains window layout preferences
Project directory	`build.log`	Records command line sent to the compiler (determined by project settings / command line options) and any feedback from the compiler during a build, e.g. errors, NAND count
	`Code.hcc`	Source file
	`Incl.hch`	Header file
	`Proj.hp`	Your project file
Intermediate directory	`Code.hb`	A program browse file used for symbol view
	`Code.hco`	Handel-C object file built during compilation
Output directory	`Proj.dll`	Part of the simulator
	`Proj.exp`	Part of the simulator
	`Proj.hb`	A program browse file used for symbol view
	`Proj.lib`	Part of the simulator

The default extensions for Handel-C files are now `.hch`, `.hcc`, `.hcl` and `.hco`, rather than `.h`, `.c`, `.lib` and `.obj`.

Files and paths

The current directory is the directory containing the current project's .hp file. All relative path names are calculated from that current directory.

4.2.2 Adding files to a project

When developing a DK project you can add a file that you have already written or create a new, empty one.

If you have existing Handel-C files which use the old extensions (`.c`, `.h`) you should rename them. The new extensions for Handel-C files are `.hcc` and `.hch`. Files with old extensions should still be recognized.

Adding a file to an existing project

1. Select **Project**>**Add to Project**>**Files**

   OR

   Right-click the mouse on the project, and select **Add Files to Folder** from the shortcut menu.

   The **Add Files** dialog box appears.
2. Select the type of file you wish to browse for from **Files of type** pull-down list. You can search for Handel-C files, ANSI-C/C++ files or all types of files.
3. Select one or more files to add and click **Open**.

Opening an existing source code file does not add it to the project. It will not be built or compiled. You must explicitly add files to the project.

Setting the language of a file

- When you are adding a file to a project, browse for the file using the appropriate file type in the **Files of type** box.

OR

- Click on the file in the **File View** window. Then access the file properties (**View**>**Properties**, or right-click on the file in the **File View** window). You can select the language on the **General** tab.

4.2.3 Multi-file projects

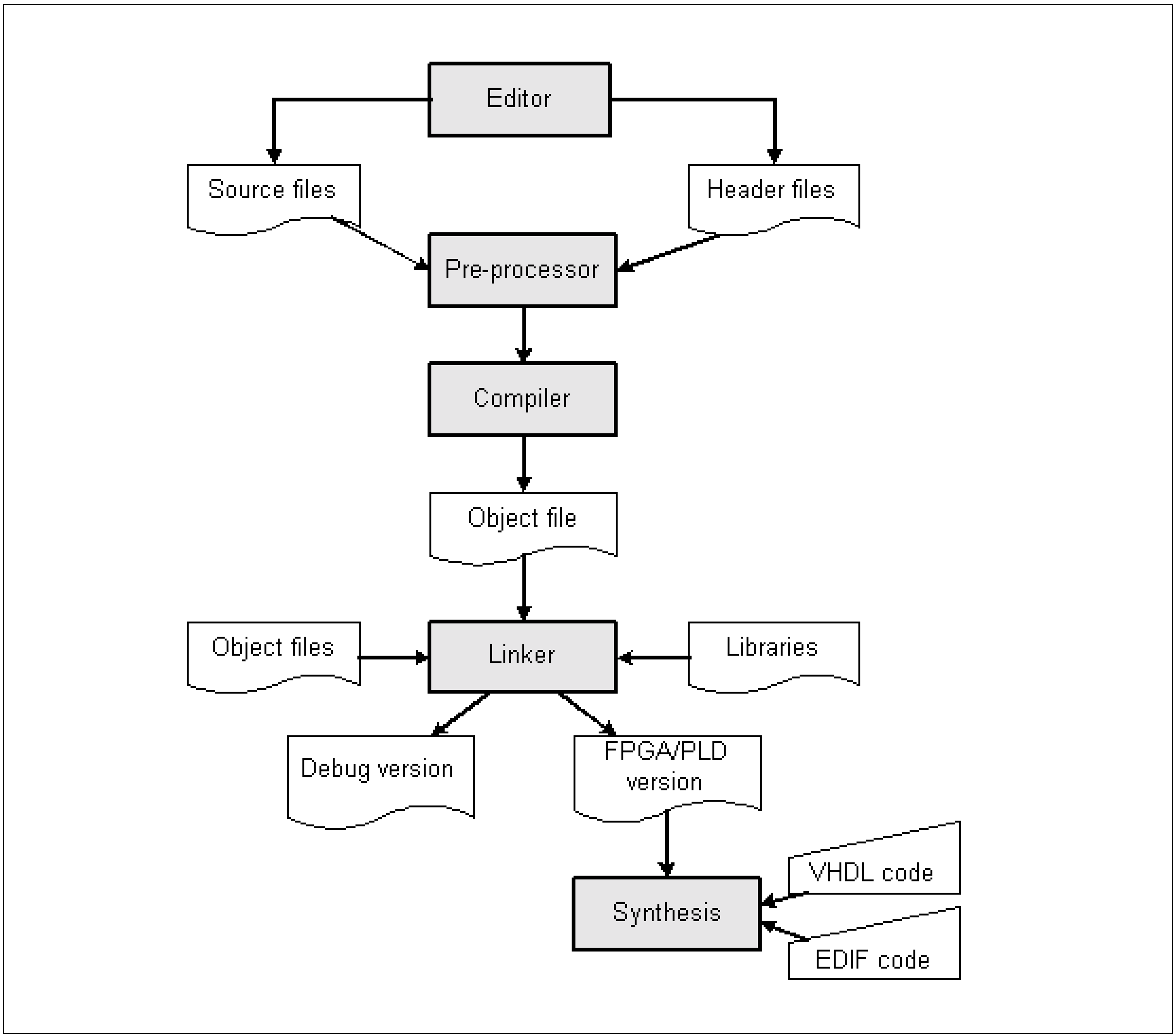
You can combine multiple files in a single project. The project can have a single `main()` function or several. If there are multiple `main` functions within a single project, they can be loaded onto the same chip. Each `main()` function can be associated with a different clock by putting it in a separate source file. If you have more than one `main()` function in the same source file, they must all use the same clock.

The project can include libraries (pre-compiled Handel-C code). EDIF, Verilog and VHDL linking is done by Place and Route tools.

4.2.4 Linking multiple files to a single output module

The Handel-C compiler has a linker, allowing you to have multiple input files and links to library files.

Multiple files can be linked into a single output module. These files can be pre-compiled core modules, libraries or header files. The `extern` keyword allows you to reference a function or variable in another file.



LINKING MULTIPLE FILES TO A SINGLE OUTPUT MODULE

Linking is carried out during a build. You define the files to link by adding files to a project within the GUI.

4.2.5 Removing files or folders from a project

You can remove a file or folder from a project by selecting it in the Workspace window and pressing the Delete key or selecting **Edit**>**Delete**.

Note that the folders within a project do not exist within your directory structure. If you delete a folder from a project, its contents will also be deleted. Files are not deleted the file from the hard disk, so no confirmation will be asked for.

4.2.6 Search paths for project files

Code files that you have added to the project workspace will be compiled and built. Header files will only be found by the preprocessor if they exist on a known path.

The directories searched are in the following order:

1. Directory containing the Handel-C file that has the `#include` directive (if within quotes)
2. Directories listed in **Project**>**Settings**>**Preprocessor**> **Additional include directories** (in the order specified)
3. Directories listed in the **Directories** pane of the **Tools**>**Options** dialog (in the order specified)
4. Directories in the `HANDELC_CPPFLAGS` environment variable (in the order specified)

4.3 Workspace and project directories

When you create a workspace, a directory is created for that workspace. Projects within the workspace may be in the same directory or a sub-directory.

When you build a project, a directory is created for the build results. The default directory name is the name of the configuration type (`Debug`, `Generic`, `Release`, `Verilog` `VHDL` or `EDIF`). You can change this by setting the **Output Directory** values in the **General** tab of the **Project Settings** dialog.

Adding an existing project to a workspace

To add an existing project to the current workspace:

1. Select Insert Project into Workspace from the **Project** menu.

   An **Open** dialog appears.
2. Browse for the project (`.hp`) file that you wish to add to the workspace.

4.4 Configuring a project

Once you have created a project, you should configure its settings. These settings define what type of chip is targeted, and how the compiler, preprocessor and optimizer work.

The default settings are correct for a new project that you wish to debug.

4.4.1 Defining project configurations

A collection of project settings is referred to as a configuration. DK provides six default configurations: Debug, Release, VHDL, Verilog, EDIF and Generic. VHDL, Verilog and EDIF are not available in Nexus PDK.

You can define your own configurations by copying an existing one and making changes to it.

1. Select **Build**>**Configurations...**
2. Click the **Add** button in the dialog that appears.
3. Enter a name for your new configuration, and select the configuration type that you wish to use as a base in the **Copy settings from** box. Click **OK**.
4. Click the **Close** box.
5. Open the **Project settings** dialog, select the new configuration and edit the settings as required.

User-defined configurations are only available within the project they were created in. The maximum number of configurations in a single project is 1024.

Making changes to a project configuration

To change a project configuration, open the **Project Settings** dialog, and select it in the **Settings For..** box. Any changes that you make are saved with this configuration.

4.4.2 Complex projects

If you know that you are going to have multiple projects (perhaps you need to have two independent circuits on the same chip), it is better to create a workspace first and then add the projects to it.

If you have an existing workspace set up, open it. Otherwise:

1. Select **New** from the **File** menu. Create a new workspace to store your project(s).
2. You are asked to enter the workspace name and the path for the directory where it is to be stored. Workspace files have `.hw` extensions

   Type the path in the Location box,

   OR

   Use the ... button to browse for a directory.

Creating a complex project

If your project is a board or system, it will contain subprojects. If you merely add files to a complex project, you can compile them but not link them. For them to be linked successfully, they must be in a sub-project (which may be a chip, core or library).

To ensure that the subprojects are built when you build the complex project, you can set up the subprojects as dependencies of the board or system project. Select **Project>Dependencies**... You will be offered a list of the projects in the workspace. Check the ones that you wish to be rebuilt when you build the complex project.

When you create a new complex project type (by writing a new `.cf` file) a dialog box appears when you click **OK**. The **New Project Components** dialog box asks what projects you wish to use for the components of your project. You can either create a new project or select one within the workspace from the drop-down list. If your project exists but is not in the workspace, you can add it using the **Insert Project** button.

4.5 Project and file dependencies

Dependencies ensure that files that are not part of the project are updated during a build. They also specify the order that files must be compiled and built.

There are three types of dependencies used in DK:

- Project dependencies
- File dependencies
- External dependencies

The only one you can change directly is Project Dependencies. The others show information calculated by the compiler.

4.5.1 File dependencies

File dependencies are listed in the file properties. They specify:

- The user `include` files that are not included in the project but are needed to compile and build a selected file
- Other files in the project that must be compiled before this file

These dependencies are generated when you compile the file. You can specify dependencies for a file that is compiled using custom build steps.

To examine dependencies for a file

- Select the file in the **File View** pane of the Workspace window and press Alt + Enter

  OR
- Right-click the file name and select **Properties** from the shortcut menu

4.5.2 Project dependencies

The **Project**>**Dependencies...** dialog allows you to select other projects within the workspace that this project is dependent on. Projects listed here will be rebuilt as necessary when the project is rebuilt.

If you are building a complex project, such as a board or system that has several chips on it, you can create a separate project for each chip, and make the system project dependent upon them.

4.5.3 External dependencies

The **External Dependencies** folder appears in the Workspace window after a project has been built. It contains a list of the header files required by the project that are not included in the project.

4.6 Properties dialog

To view the properties of a file, folder, project or workspace:

1. Select a file or other item in the Workspace window.
2. Select **View**>**Properties**.

Alternatively you can right-click after selecting the item and choose **Properties**.

The properties are displayed on the following tabs:

- General
- Inputs
- Outputs
- Dependencies

4.6.1 General tab

The information displayed on the **General** tab depends on whether you are viewing the properties of a file, folder, project or workspace.

Selection	Item	Description
File	**Filename**	Displays name and full path of current file
	Language	Allows you to select file type: Handel-C or ANSI C/C++
Folder	**Folder Name**	Displays name of folder and allows you to change it
	Extensions	Displays the extensions associated with the folder and allows you to change them
Project	**Project File**	Displays name of project
Workspace	**Workspace Name**	Displays name of current workspace
	Workspace Path	Displays full path to workspace file (.hw)

The language option allows you to choose whether to compile a file for Handel-C, ANSI C or C++. If you want to build ANSI C or C++ files, you need to specify custom build commands.

4.6.2 Inputs tab

The information on the Inputs tab is set up by the Project settings. If **Always use custom build step** has been selected for the file or project, inputs are specified by the build commands. Otherwise, they are determined by the compiler.

Selection	Item	Description
File	**Tools**	Displays tools associated with current file
	Files	Displays the name and full path of current file
Project	**Tools**	Displays tools associated with current project
	Files	Displays the name and relative path of input files for each tool

4.6.3 Outputs tab

The information on the Outputs tab is set up by the Project settings. If the **Always use custom build steps** option has been selected for the file or project, outputs are specified by the outputs defined on the **Build commands** tab in Project Settings. Otherwise, outputs are determined by the compiler.

Selection	Item	Description
File	**Tools**	Displays tools associated with current file
	Files	Displays the name and relative path of the output file for the current build configuration
Project	**Tools**	Displays tools associated with current project
	Files	Displays the name and relative path of the output files for the current build configuration

4.6.4 Dependencies tab

The **Dependencies** tab is only visible on the **Properties** dialog if you have a file selected. The information on it is set up by the **Project settings**. If **Always use custom build steps** has been selected for the file, the dependencies are specified by the build commands. Otherwise, they are determined by the compiler.

Item	Description
Tools	Displays tools associated with current file
Files	Displays the files that must be compiled before the selected file: • user `include` files that are not included in the project but are needed to compile and build the selected file • other files in the project that must be compiled before this file The list is generated when you compile the file. If you have used a custom build step, the list is generated from the information that you give in the **Build commands** tab.

4.7 Project and file settings

Project settings define how your files and projects are compiled and built. Select **Project>Settings** to see the **Project Settings** dialog box. The different settings are available via tabs. The tabs available will depend on the project type. For example, the **Library** tab is only available for a library-type project.

The tabs available are:

- General
- Chip
- Preprocessor
- Compiler
- Optimizations
- Linker
- Debugger
- Build commands
- Library

If you can't see the tab you want, then scroll the tabs by clicking on the arrows at the end of the tabs. Some tabs may only be visible if you have selected a Handel-C file in the left window.

Independent settings for files

You can create independent settings for a file. You might wish to do this if you wanted to change the optimization level or specify custom build commands for a particular file. Project settings for a file override the general project settings.

To create settings for a file:

1. Open the **Project Settings** dialog (either right-click the file in the File View and select **Settings**, or select **Project>Settings**).
2. Select the name of the file that you wish to affect in the file pane of the Project Settings dialog.
3. Make the appropriate changes.

4.7.1 General tab

Different settings are available for projects and for individual files.

Item	Meaning	Value	Default
Always Use Custom Build Steps	Allows you to use custom build steps for a Handel-C file instead of a normal build.	Check to use custom build steps	Clear
Exclude From Build	Excludes file from build	Check to exclude file from current build.	Clear
Intermediate files	The sub-directory where intermediate files are stored	Directory path name relative to the project directory	***configuration name***
Output files	The sub-directory where the final output is stored (`.dll`, netlist etc.)	Directory path name relative to the project directory	***configuration name***

If you specify custom build steps, they will always be executed for a project or a non-Handel-C file. If you specify them for a Handel-C file, they will only be executed if you tick **Always Use Custom Build Steps**.

4.7.2 Preprocessor tab

Item	Meaning	Value	Default
Preprocessor definitions	Equivalent to the **#define** directive	Set as required	`DEBUG`, `SIMULATE` or `NDEBUG`
Additional include directories	Add directories to the search path for `include` directories	Set as required; separate multiple paths by a comma	None
Ignore standard include directories	Allows you to omit default include search path, (to ignore standard include files).	Check to omit default include search path.	Clear
Additional preprocessor options	Add any cpp commands	Set as required	None

4.7.3 Compiler tab

Item	Meaning	Value	Default
Generate debug information	Compile for debug-enabled simulation. Only available in Debug, Release and Generic modes.	Check for Yes	Checked for Debug. Not checked for Release or Generic.
Generate warning messages	No effect. This option is reserved for future use.	-	-
Detection of simultaneous function calls	Detect simultaneous calls to the same function when debugging. You can only use this option in Debug, using the new (fast) simulator	Check to turn option on	Checked
Detection of simultaneous channel reads/writes	Detect simultaneous accesses to the same channel when debugging. You can only use this option in Debug, using the new (fast) simulator.	Check to turn option on	Checked
Detection of simultaneous memory accesses	Detect simultaneous accesses to memory when debugging. You can only use this option in Debug, using the new (fast) simulator.	Check to turn option on	Checked
Expand netlist for:	Specify whether the netlist should be expanded to minimize area (select Area from drop-down list) or to maximize speed (default). This option only has an effect for EDIF output for Actel devices.	Select Area or Speed	Speed

Detection of simultaneous memory accesses will only detect simultaneous accesses to different addresses within a memory, not simultaneous accesses to the same address.

4.7.4 Optimization tab

Item	Meaning	Value	Default
High-level optimization	Early, high level optimization. Speeds up compilation.	Check if required	Not checked for Debug mode. Checked for all other modes.
Rewriting optimization	Optimize logic where signals are tied high or low etc.	Check if required	Checked. Not available for Debug or Release modes.
Common sub-expression (CSE) optimization	Eliminate duplicate common sub-expressions. Usually leads to smaller designs but may increase routing and hence delay.	Check if required	Checked. Not available for Debug or Release modes.
Partitioning before CSE optimization	Split up complex gates before performing CSE	Check if required	Checked. Not available for Debug or Release modes.
Repeated CSE optimization	Repeats CSE optimization, removing further sub-expressions. Slows down compilation.	Check if required	Checked. Not available for Debug or Release modes.
Conditional rewriting optimization	Assumes certain states and propagates the conclusions through the logic. Optimizes according to results. Will slow down compilation. Best used in conjunction with other optimizations.	Check if required	Checked. Not available for Debug or Release modes.
Repeated conditional rewriting optimization	Repeats the conditional rewrite until nothing more can be achieved. Can substantially increase compilation time.	Check if required	Not checked. Not available for Debug or Release modes.

Item	Meaning	Value	Default
Generate macros above width **(-lpm)**	Causes compiler to generate macros for common operators (e.g. multipliers, adders) instead of expanding them to gates. Place and route tools can use these macros to optimize the logic for a particular device. The logic produced tends to be optimized for speed, but may increase the size of your design.	Set to width required. For example, a value of 8 will mean macros will be created for operators that are more than 8 bits wide.	Empty (option not selected for any widths). This option is only available for EDIF output.
Disable fast carry chain optimizations	Disables the generation of fast carry chains for adders, subtractors, multipliers, dividers, comparators and modulo arithmetic. Fast carry chains tend to speed up a design, but restrict the placement of logic on a device.	Check to disable fast carry chains	Not checked. This option is only available for EDIF output.

4.7.5 Chip tab

Item	Meaning	Value	Default
Family	The family containing the part you are targeting	Select family from dropdown list	Generic
Part	The part number you are targeting	Type in part number	Depends on project

You must specify a chip type for EDIF output. If you do not want to specify a target for VHDL or Verilog output, select **Generic**. This will result in generic VHDL or Verilog without any target-specific constructs such as RAM primitives.

4.7.6 Linker tab

The items that appear on this tab depend on which build configuration you have selected.

Item	Meaning	Value	Default
Output format	Target for the compiler	Determined by target settings	As required
Object/library modules	Extra libraries (.hcl) and object files (.hco) required	Type path and file specifications separated by commas	As required
Additional Library Path	Directory path to search for Handel-C libraries	Type paths separated by commas	None
Additional C/C++ Modules	C or C++ libraries and object files required for the project	Type path and file specifications separated by commas	None
Synthesis / simulation tool	Output style for VHDL or Verilog. (You cannot target VHDL or Verilog from Nexus PDK.)	Generic, LeonardoSpectrum, Precision, ModelSim or Synplify Choose ModelSim for simulation. Choose Generic if you want to target a synthesis tool that is not listed.	Generic
Ignore standard lib path	Don't look for libraries along default library path	Check not to search standard path	Clear
Save browse info	Store information needed to browse symbols	Check to store	Checked
Use netlist simulator	Use previous (slower) version of the simulator. You would probably only want to use this if you have asynchronous signals that run faster than the design clock.	Check to use the old simulator	Clear
Generate estimation info	Get the compiler to generate HTML files giving depth and timing information (only available for EDIF builds)	Check for Yes	Clear

Item	Meaning	Value	Default
Exclude timing constraints	Disable generation of timing constraints (in generated NCF, TCL or ACF file)	Check to disable	Clear (timing constraints are generated)
Use technology mapper	Create EDIF output with look-up table primitives instead of logic gates	Check for Yes	Clear
Simulator compilation command line	Specify options for the backend compiler. Used for building simulations and PC-hosted code.	Define how the C++ compiler is called to compile `simulator.dll`. You may use 4 compiler-supplied parameters. You can also specify commands to generate an .exe file.	Link options defined in the `cl.cf` file (Debug, Generic and Release only).

Handel-C library files with the extension `.lib` and Handel-C object files with the extension `.obj` are no longer supported.

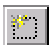
4.7.7 Debugger tab

Item	Meaning	Value	Default
Working directory	Directory that the simulator uses as the current working directory	Directory path name relative to the project directory	Current project directory (.)

4.7.8 Build commands tab

You can specify build commands for a project or an individual file. The commands will only be executed in the build configuration in which they were specified.

Build commands are always available for a project, and for ANSI-C and C++ files. If you want to specify commands for a Handel-C file, tick the **Always Use Custom Build Steps** box on the **General** tab of Project Settings. You will then be able to access the Build Commands tab.

Description	Specify a description to be displayed when the custom build step is executed. The description can include file and directory macros.
View	Choose **Commands**, **Outputs** or **Dependencies**. You can only specify dependencies for files.
Commands / **Outputs** / **Dependencies**	The use of the pane depends on what you have selected in the **View** box.
	Create a new command, output or dependency. Press return when you have finished writing.
	Delete the command, output or dependency selected.
	Move selected command, output or dependency up.
	Move selected command, output or dependency down.

4.7.9 Library tab

In a library project the **Library** tab provides settings which are provided by the **Chip** tab and **Linker** tab in other projects.

Item	Meaning	Value	Default
Family	The family containing the part you are targeting	Select family from dropdown list	**Generic** option is equivalent to omitting the `-f` option from the command line
Part	The part number you are targeting	Type the part number	Depends on project
Object/library modules	Extra libraries (`.hcl`) and object files (`.hco`) required	Type path and file specifications separated by commas	Clear
Additional library path	Extra library directories required	Type paths separated by commas	Default path is `DK\Lib` directory
Save browse info	Store information needed to browse symbols	Check for Yes	Checked

Handel-C library files with the extension `.lib` and Handel-C object files with the extension `.obj` are no longer supported.

5. Building a project

5.1 Build process

A build happens when:

- You click on the Build button
- You have uncompiled files and you select one of the **Start Debug** commands in the **Build** menu
- You select **Build** or **Rebuild All** from the **Build** menu

This should:

1. Pre-process header files and compile dependent header files
2. Compile any files that have been added or changed since your last compilation and also compile any files dependent upon them. (Changed files are saved.)
3. Compile all dependent projects
4. Link the compiled files together
5. Calculate the number of gates used
6. Build a symbol table
7. Generate a simulator `.dll` or a netlist

If you change the configuration for a project, you will need to compile all the files. Select the **Build** >**Rebuild All** command to ensure that all the files are recompiled.

The results of the compilation and build are displayed in the Build window. Double-clicking an error takes you to the appropriate line in the source file.

5.1.1 Running the compiler

The Handel-C compiler compiles and optimizes Handel-C source code into a file suitable for simulation, a VHDL or Verilog file ready for synthesis or a netlist file which can be placed and routed on a real device. (VHDL, Verilog and EDIF outputs are not available in Nexus PDK.)

DK includes a modified version of the GNU preprocessor. Flags can be passed to the preprocessor using the **Preprocessor** tab of the **Project**>**Settings** dialog box.

You can run the compiler in either of two ways:

- The compiler is normally invoked automatically when you select an option from the **Build** menu.
- To run the compiler from a command line, use the command `handelc`.

Once the compile has completed, the output window shows an estimate of the number of NAND gates required to implement the design.

5.1.2 Setting up code for debug

There are several methods of coding Handel-C to help you debug a project.

They fall into two kinds:

- Code that will automatically be discarded by the compiler if you do not compile a project for debug, e.g. the `with {infile = "`***`file`***`"}` directive.
- Code where you supply alternatives to be compiled for debug and release or target compilations. In these cases, you can use the `#ifdef DEBUG`, `#ifdef NDEBUG` and `#ifdef SIMULATE` directives.

By default, `DEBUG` and `SIMULATE` will be defined if you compile for debug, and `NDEBUG` will be defined for all other compilations.

Example

```
#ifdef SIMULATE
    sim_chan ? var; // Read from simulator
else
    HardwareMacroRead(var); // Real HW interface
endif
```

Summary of coding techniques used for debug

- Substitute simulator channels for hardware interface channels.
- Use the `assert` directive to stop a compilation if a condition is untrue.
- Substitute file input for external channel input.
- Export the contents of variables into files.

5.1.3 Building and compiling for debug

Debug is the default compilation configuration.

Open the Project Settings dialog (Alt F7). Check that **Debug** appears in the **Settings For** dropdown menu. The compiler will create a file which is in turn compiled into native machine code using Microsoft Visual C++, GCC or GNU C++. This creates the chip simulation.

To build and compile your project, select **Build** from the **Build** menu. Messages from the compiler appear in the **Build** tab of the output window.

5. Building a project

5.1.4 Building with library and object files

Creating a library file

To create a library file, create a project of type **library** and build it as normal. It will generate a `.hcl` file. Library projects have a **Library** tab instead of a **Linker** tab in the Project settings dialog.

Using a library file

You can use a Handel-C library file in any project.

1. Select the **Linker** or the **Library** tab in the Project settings dialog.
2. Add the library file name to the Object/library modules box.
3. Default library paths for DK are set up in the **Directories** tab of the **Tools>Options** dialog. If the new library's directory path is not set up for DK, set it up for your project by adding the directory path to the **Additional Library Path** box. Multiple file names must be separated by commas. Wildcards are not supported.

Using an existing object file

You can use an existing compiled object (.hco) file in another project

1. Select the **Linker** or the **Library** tab in the Project settings dialog.
2. Add the object file name to the Object/library modules box. You may use an absolute or relative path name. Multiple file names must be separated by commas. Wildcards are not supported.

5.1.5 Preparing to build for hardware

Once your program has been simulated correctly you must add the necessary hardware interfaces. It is worth testing all interface outputs and inputs using a simulator such as the Waveform Analyzer before you build for hardware.

- Convert any file reading and writing procedures into interface or bus procedures.
- Ensure that you have converted all C/C++ functions to Handel-C.
- Convert any interfaces to plugins into interfaces to black box code or remove them entirely.
- Define and declare any external RAMs, off-chip interfaces etc.
- Change the project settings to EDIF, Verilog or VHDL.

You can only target hardware from DK Design Suite. Nexus PDK will only let you simulate your project.

5.1.6 Compiling for release or target

When you are satisfied with your project, select **Build**>**Set Active Configuration** and choose the type of build you require from the available configurations.

VHDL

VHDL files may be simulated using a VHDL simulator (such as ModelSim), synthesized using an RTL tool, and then placed and routed. By default, most optimizations will be turned on. This option is disabled in Nexus PDK.

Verilog

Verilog files may be simulated using a Verilog simulator (such as ModelSim), synthesized using an RTL tool, and then placed and routed. By default, most optimizations will be turned on. This option is disabled in Nexus PDK.

EDIF

EDIF files are ready to be placed and routed. By default, most optimizations will be turned on. This option is disabled in Nexus PDK.

Release

Release allows you to simulate your project without the delays inherent in debug. It also allows you to compile simulation-only libraries without debug information, to protect intellectual property.

Generic

Generic build mode only applies to library projects. Generic libraries are Handel-C intellectual property which are not targeted at a particular output format. They consist of compiled code that can be used in another program. Generic mode can be linked for simulation, EDIF 2.0.0, Verilog IEEE Std 1364-1995 or VHDL 1987.

5.2 Build commands in DK

Build commands are specified on the **Linker** tab or **Build commands** tab in Project Settings or in the command-line compiler.

You need to specify custom build commands if you want to build a non-Handel-C file (e.g. C or C++ file). You can also specify custom post-build commands.

.dll files

`.dll` files are created by default when you build a Handel-C simulation.

.exe files

If you want to build an `.exe` file, change the **Simulator compilation command line** on the **Linker** tab in Project Settings. Alternatively, use an appropriate build command in the command-line compiler.

.obj files

If you want to use Handel-C functions in your C or C++ code, you need to build the Handel-C file as an .obj file. Change the Simulator compilation command line on the **Linker** tab in Project Settings. Alternatively, use an appropriate build command in the command-line compiler.

If you want to build C or C++ files for simulation, you need to build these files as .obj files using custom build commands on the **Build commands** tab of Project Settings.

5.2.1 Simulator compilation command lines

The **Simulator compilation command line** is specified on the **Linker** tab in Project Settings. The default command line uses the backend compiler you specified when you installed DK, and the new simulator to build a `.dll` file for simulation. You need to change this if you have changed your backend compiler (Visual C++, Borland or GCC). If you tick the **Use netlist simulator** option on the **Linker** tab, the **Simulator compilation command line** for the old simulator will appear instead.

If you are using the command line compiler instead of the GUI, you need to select the correct command from those listed below.

You also need to change the **Simulator compilation command line** if you want to build an .exe file or an .obj file instead of a `.dll` file.

Commands for different compilers

There is a different default simulation command line for each of the backend compiler supported by DK.

If you are targeting the new simulator, the default command lines are:

- Microsoft Visual C++: `cl /Zm1000 /LD /Oityb1 /GX /I"`***`InstallDir`***`\DK\Sim\Include" /Tp"%1" /Fe"%2" %4`
- Borland C++: `bcc32 -WD -g0 -j0 -P- -O2 -I"`***`InstallDir`***`\DK\Sim\Include" -e"%2" "%1" %4`
- GCC: `g++ -dll -shared -fno-builtin -I"`***`InstallDir`***`\DK\Sim\Include" -O2 -mno-cygwin "%1" -o"%2" %4`

Targeting the old (netlist) simulator

If you are targeting the old simulator, the default command lines are:

- Microsoft: `cl /LD /Oityb1 /nologo /I"`***`InstallDir`***`\DK\Sim\Include" /Tc"%1" /Fe"%2" "`***`InstallDir`***`\DK\Sim\Lib\numlib.lib" /link /nodefaultlib:libc.lib`
- Borland: `bcc32 -WD -g0 -P- -O2 -I"`***`InstallDir`***`\DK\Sim\Include" -e"%2" "%1" "`***`InstallDir`***`\DK\Sim\Lib\numlibbc.lib"`

- GCC: `gcc -dll -shared -O2 -mno-cygwin -I"`***InstallDir***`\DK\Sim\Include" -o"%2" -xc "%1" -xnone "`***InstallDir***`\DK\Sim\Lib\numlibgcc.lib"`

To target GCC and the new simulator, use "G++...". To target GCC and the old simulator, use "GCC...".

5.2.2 Generating a standalone executable

You can generate an .exe file from Handel-C by changing the default simulator compilation command line. The command needs to target the correct backend compiler (your C++ compiler).

If you build a simulation to run as an .exe file it will run faster than a simulation from the DK GUI.

Using Project Settings in the GUI

If you are using the GUI compiler, change the default text in the **Simulator compilation command line** pane in the **Linker** tab of the **Project Settings** dialog.

- Visual C++: `cl /Zm1000 /Oityb1 /GX /I"`***InstallDir***`\DK\Sim\Include" /Tp"%1" /Fe"%3.exe" %4`
- Borland: `bcc32 -WD -g0 -j0 -P- -O2 -I"`***InstallDir***`\DK\Sim\Include" -e"%3.exe" "%1" %4`
- GCC: `g++ -dll -shared -fno-builtin -I"`***InstallDir***`\DK\Sim\Include" -O2 -mno-cygwin "%1" -o"%3.exe" %4`

You could set up a new build configuration to store these project settings.

Using the command line compiler

If you are using the command line compiler, change the `-cl` option. For example, if you are using GCC as your backend compiler:

```
handelc -s -cl"g++ -dll -shared -fno-builtin -I"InstallDir\DK\Sim\Include"
    -O2 -mno-cygwin "%1" -o"%3.exe" HandelCFileName.hcc
```

5.2.3 Generating an .obj file

If you want to use Handel-C functions in C or C++ code, you need to compile your Handel-C file as an .obj file. To generate an .obj file from a Handel-C file you need to change the default simulator compilation command line in DK.

5. Building a project

Using the GUI

To create an `.obj` file from a Handel-C file using the GUI compiler, change the default **Simulator compilation command line** on the **Linker** tab of the Project Settings dialog. Use the relevant command for your backend compiler:

- Microsoft Visual C++: `cl /Zm1000 /c /Oityb1 /GX /I"`***`InstallDir`***`\DK\Sim\Include" /Tp"%1" /Fo"%3.obj" %4`
- Borland C++: `bcc32 -c -g0 -j0 -P- -O2 -I"`***`InstallDir`***`\DK\Sim\Include" -o"%3.obj" "%1" %4`
- GCC: `g++ -c -shared -fno-builtin -I"`***`InstallDir`***`\DK\Sim\Include" -O2 "%1" -o"%3.obj" %4`

You could set up a new build configuration to store these project settings.

Using the command line compiler

If you are using the command line compiler, change the `-cl` option:

- Visual C++: `handelc -s `***`HandelCFileName`***`.hcc -cl "cl -c -O2 -I"`***`InstallDir`***`\DK\Sim\Include" %1 -Fo%3.obj"`
- Borland: `handelc -s `***`HandelCFileName`***`.hcc -cl "bcc32 -c -O2 -I"`***`InstallDir`***`\DK\Sim\Include" -o%3.obj %1"`
- GCC: `handelc -s `***`HandelCFileName`***`.hcc -cl "g++ -c -O2 -I"`***`InstallDir`***`\DK\Sim\Include" %1 -o%3.obj"`

If you want to create an `.obj` file from a C or C++ file, you have to specify a custom build command on the **Build Commands** tab in Project Settings

5.2.4 Post-build commands

You can specify post-build commands for a project on the **Build Commands** tab in Project Settings. Open Project Settings, then check that your project is selected in the left pane, rather than a file. You can then select the **Build commands** tab. Any build commands that are specified at the project level are executed after all the project files have been compiled.

Example

To copy the `Result.dll` to a different directory after it has been compiled:

1. Select **Commands** in the **View** box and type
   `copy $(TargetDir)\Result.dll $(WkspDir)\bin\Result.dll`
2. Select **Outputs** in the **View** box and specify `$(WkspDir)\bin\Result.dll.` as the output file.

5.3 Custom build commands

Custom build commands are specified on the **Build Commands** tab in Project settings. You can specify custom build commands for

- a project
- an individual file

You need to specify custom build commands if you want to build a file that is not a Handel-C file (e.g. a C++ file).

If you specify custom build commands, they will always be executed for a project or a non-Handel-C file. If you specify them for a Handel-C file, they will only be executed if you tick **Always Use Custom Build Steps** box on the **General** tab.

The build commands will be executed at the appropriate point in the build process if the output file is out of date with respect to the input file. Custom build commands applying to the whole project will always be executed after the normal build process has completed.

If you specify commands involving a `.bat` file, you need to precede the command with "`call`".

The commands will only run in the configuration in which you specified them.

5.3.1 Specifying a custom build

To specify a custom build:

1. Open the Project Settings dialog (**Project**>**Settings**). Select a file or a project in the left pane.
2. Click on the **Build commands** tab.
   If you have selected a Handel-C file you will need to tick the **Always Use Custom Build Steps** box on the **General** tab to access the Build commands tab.
3. Type a description in the **Description** box.
4. Select **Commands** in the **View** box.
5. Type your commands in the pane below.
6. Select **Outputs** in the **View** box and write the names of your output files in the pane below. You must specify at least one output file.
7. If you are specifying build rules for a file, you can also specify dependencies (select **Dependencies** in the **View** box).
8. Click **OK** then build your project. You will see the text you specified in the **Description** box as the custom build steps are executed.

If you are using GCC as your backend compiler, this should be specified in the build command using g++ if you have a C++ file, or gcc if you have a C file.

5.3.2 Build commands, outputs and dependencies

Custom build commands, outputs and dependencies are specified on the **Build commands** tab.

Use quotes around strings if they have spaces in them.

Commands

You can specify build commands when in the Commands view. Commands can include file and directory macros. If you write more than one command, they will be run in order from top to bottom. If you create a command involving a `.bat` file, you need to precede the command with "call".

Outputs

You can specify the names of output files when in the Outputs view. The files are time stamped and the commands are only executed when the files are out of date with respect to the associated source files or project. Specify different output files for each set of commands. If you specify the same output file for more than one file with build commands, or for a file and the project it belongs to, only the first set of commands will be executed.

You must specify at least one output file (even if you specify custom build steps at project level).

Dependencies

You can specify files that need to be built before the custom build step when you are in the Dependencies view. This will affect the order of the build process.

5.3.3 File and directory macros

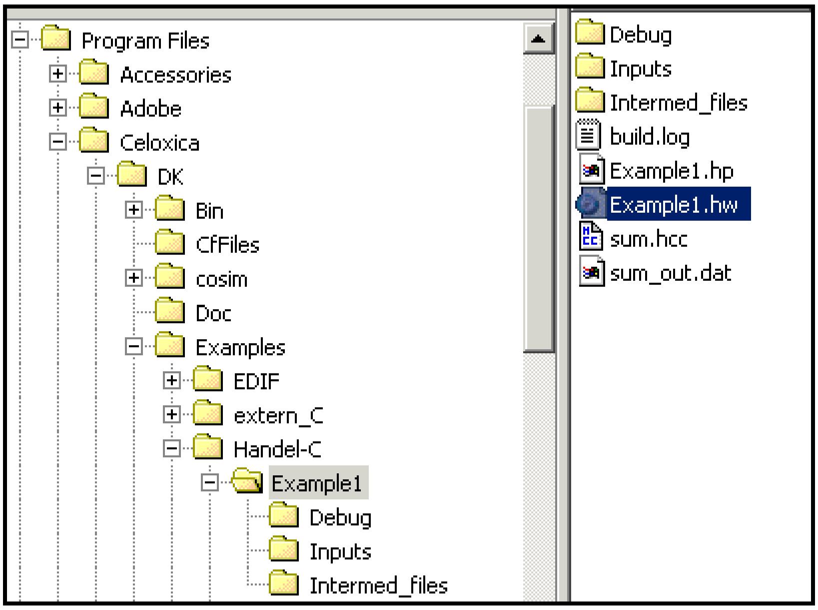
File and directory macros are supported for use in custom build commands. You can also use them in the custom build description. Write the macros in the form `$(`***`Macro`***`)` where ***`Macro`*** refers to one of the file or directory expressions listed below, such as `$(IntDir)`.

Macros are expanded prior to display or execution. If the expanded macro contains spaces, you will need to enclose the macro name in quotes. The directory or file referenced must already exist or be created by DK or another tool before the macro runs.

File and directory macros make it easier to move your project to a different directory or computer, and reduce the chance of typographical errors in file pathways.

Macro	Description
$(IntDir)	Path to directory specified for intermediate files, relative to project directory
$(OutDir)	Path to directory specified for output files, relative to project directory
$(TargetDir)	Full path to the directory specified for output files
$(InputDir)	Path to input directory relative to project directory
$(ProjDir)	Full path to project directory
$(WkspDir)	Full path to workspace directory
$(TargetPath)	Name and full path for project output file
$(TargetName)	Name of output file
$(InputPath)	Name and full path for input file
$(InputName)	Name of input file
$(WkspName)	Name of project workspace

Examples



Assuming the directory structure above on drive C:

- $(OutDir) would expand to \Debug
- $(TargetDir) would expand to C:\Program Files\Celoxica\DK\Examples\Handel-C\Example1\Debug
- $(ProjDir) would expand to C:\Program Files\Celoxica\DK\Examples\Handel-C\Example1
- $(WkspName) would expand to Example1.hw

6. Command line compiler

The Handel-C compiler can be invoked from the command-line as well as from the GUI. If you wish to use it from the command-line, you must pass options to it directly instead of via the **Project settings** dialog.

To run the compiler from the command line, use the command `handelc`, for example:

```
handelc -verilog -syn Leonardo MyFile.hcc
```

6.1 Summary of command line options

The table below summarizes the options available when using the command line compiler.

Option	Meaning
`-c`	Compile only. Do not generate netlist. Output .hco or .hcl file
`-f` ***Family***	Specify target family
`-p` ***Part***	Specify target part
`-fc`	Disable generation of fast carry chains
`-b`	Generate browse info database file
`-notcon`	Disable generation of timing constraints in generated NCF, TCL or ACF file
`-r` ***"Filename"***	Specify browse info database file name
`-o` ***"Path_and_Name"***	Specify output file name and path
`-xc` "***Filename***"	Treat file as Handel-C source file
`-xl` ***"Filename"***	Treat file as Handel-C library file
`-xo` "***Filename***"	Treat file as Handel-C object file
`-L` "***Pathname"***	Add ***pathname*** to library path
`-help`	Print help screen (summary of command line options)
Simulation and debugging options	
`-s`	Target simulator (use -s -ns to target old simulator for unsupported plugin features)
`-cl` ***"CommandLine***"	Specify command line for compiling simulator output
`-be` "***Options***"	Pass options to backend compiler
`-g`	Compile with debug information
`-S"`***string***`"`	Detect simultaneous function calls, channel and memory accesses
`-W`	Reserved for future use.

Hardware output options (not available in Nexus PDK)

`-edif`	Target EDIF output
`-vhdl`	Target VHDL output
`-verilog`	Target Verilog output
`-e`	Estimate logic depth and area when generating EDIF output. (Generate HTML files)
`-lutpack`	Use technology mapper when generating EDIF output
`-syn` ***`SynthesisTool`***	Specify VHDL or Verilog output style
`-N+speed`	In EDIF output for Actel devices, expands netlist to maximize speed
`-N+area`	In EDIF output for Actel devices, expands netlist to minimize area

Preprocessor options

`-cpp` ***`"Option"`***	Pass *Option* to preprocessor. This enables you to pass options in addition to those listed below.
`-D` ***`Symbol`***	Define preprocessor symbol
`-E`	Pre-process source only.
`-I` ***`"Pathname"`***	Add ***`pathname`*** to preprocessor include path
`-U` ***`Symbol`***	Undefine preprocessor symbol

Optimizer options

`-O`	Turn on maximum optimizations
`-O-`	Turn off all optimizations
`-O+` ***`optimize`***	Turn on ***`optimize`*** optimization
`-O-` ***`optimize`***	Turn off ***`optimize`*** optimization
`-lpm` ***`Width`***	Use macros for data paths wider than ***`Width`***

6.2 Compiler target options

The Handel-C compiler can target the simulator or hardware. Hardware targeting is not available in Nexus PDK. If you are using the command line compiler, you can specify one of the target options:

`-s`	Target the fast simulator
`-s -ns`	Target the backward compatible (old) simulator
`-vhdl`	Target VHDL 1987 output
`-edif`	Target EDIF 2.0.0 output
`-verilog`	Target Verilog IEEE Std 1364-1995

Target modifications

You can modify the HDL code generated by using further options:

-syn ***SynthesisTool***	Specify the style of VHDL or Verilog output. ***SynthesisTool*** must be one of: `Generic`: generates generic code. Use this option if you want to target a synthesis tool that is not listed in one of the other options. `Leonardo`: generates Mentor Graphics LeonardoSpectrum-style code `Synplify`: generates Synplicity Synplify-style code `Precision`: generates Mentor Graphics Precision-style code `ModelSim`: generates Model Technology ModelSim-style code. Use this option if you want to simulate your code. This option is ignored if not used in conjunction with the `-vhdl` or `-verilog` options. E.g. `handelc -verilog -syn Leonardo MyFile.hcc`
`-lutpack`	Use technology mapper when generating EDIF output

If you are generating VHDL or Verilog code for simulation with ModelSim, you can only use multi-port memories if the ports have the same width and the same depth.

6.3 Pass options to preprocessor

If you are using the command line compiler, you can use the -cpp option to pass options to the Handel-C preprocessor.

The following options are available:

Option	Description
`-D` ***Symbol***	Define preprocessor symbol
`-U` ***Symbol***	Undefine preprocessor symbol
`-E`	Pre-process source only (stop after pre-processing and don't compile code).
`-I` ***Pathname***	Adds ***Pathname*** to preprocessor include path

`-I`, `-D` and `-U` can be used directly and do not have to be passed to the preprocessor with -cpp.

Example

```
handelc -s -cpp -Iinclude prog.hcc
```

This adds the directory include to the search path.

6.4 Optimizer options

If you are using the command line compiler, you can use the `-O` option to control the optimization levels. (If you are using the DK GUI, use the **Optimization** tab in Project Settings.)

Option	Description
`-O`	Turns on all optimizations
`-O-`	Turns off all optimizations
`-O+`***`optimize`***	Turns on ***`optimize`*** optimization
`-O-`***`optimize`***	Turns off ***`optimize`*** optimization

The possible values for ***`optimize`*** are:

`cr`	Conditional rewriting optimizations
`cse`	Common sub-expression elimination optimizations
`fcc`	Disable fast carry chain optimizations
`high`	High-level optimizations
`lpm` ***`N`***	Generate macros (instead of gates) for common operators above width ***`N`*** For example, `lpm 8` means that macros will be created for operators that are more than 8 bits wide.
`pcse`	Partitioning before CSE optimizations
`rcse`	Repeated CSE optimizations
`rcr`	Repeated conditional rewriting optimizations
`retime`	Retiming optimizations. (Moves flip-flops from gate inputs to the outputs to reduce the number of FFs. This is only applied locally and does not take into account any potential negative effects on timing.)
`rewrite`	Rewriting optimizations

(Further information about these options is available in the description of the **Optimization** tab (Project Settings).)

If no optimizer command line options are specified:

- In EDIF, VHDL, Verilog and Generic modes all optimizations are enabled except for `fcc`, `lpm` and `rcr`. Enabling `rcr` can substantially increase compilation time.
- In Debug mode, no optimizations are enabled. You can only specify high-level optimization (high) in Debug mode.
- In Release mode, only high-level optimization is enabled. You cannot enable any other optimizations in this mode.

Examples

`handelc -O prog.hcc -edif`

Compiles the program `prog.hcc` with all default optimizations.

`handelc -O+rcr prog.hcc -edif`

Compiles the program `prog.hcc` with all default optimizations plus repeated conditional rewriting.

`handelc -O-cse prog.hcc -edif`

Compiles the program `prog.hcc` all default optimizations except for common sub-expression elimination.

6.5 Compiler debugging options

If you are using the command line compiler, you can use these options to help you debug Handel-C programs:

-s	Target the simulator
-s -ns	Target the old simulator (if you have asynchronous signals that run faster than the design clock)
-g	Compile with debug information
-e	Estimate logic area and depth
-S	Detect simultaneous accesses to functions, memory and channels
-W	No effect. Reserved for future use.

6.5.1 Targeting the fast simulator

If you are using the command line compiler, use the -s option to target the new (fast) simulator.

handelc -s *file*.hcc

To use the old simulator, use the -s -ns option.

The fast simulator is targeted by default from the GUI.

If you are using GCC as your backend compiler, you need to use the command line option `G++` if you are targeting the new simulator and `GCC` if you are targeting the old simulator (see default simulation command lines).

6.5.2 Targeting the old (netlist) simulator

You would probably only want to use the old simulator (netlist simulator) if you have an asynchronous path from an input to an output where the input is fed from a plugin which changes the value faster than the clock and the output has a plugin which reads data faster than the clock. The fast simulator will give incorrect results, since it is a time cycle accurate simulator.

Using the command line compiler

To target the previous version of the simulator, use the -ns option on the compiler command line after the simulator option (-s).

```
handelc -s -ns file.hcc
```

The default simulation command lines are different for the old and new simulator.

Using the DK GUI

To target the old simulator from the GUI, select the **Use netlist simulator** option on the **Linker** tab in Project Settings. The **Simulator compilation command line** on the **Linker** tab will change to the default command for the old simulator.

6.5.3 Detecting simultaneous access to functions, memory and channels

When you are debugging your code, you can choose whether you want the simulator to detect simultaneous calls to functions, simultaneous memory accesses and simultaneous channel accesses.

You can only use these options with the fast simulator (not the old netlist simulator). By default, all of these options are switched on. The detection of simultaneous memory access may slow down the debugger significantly if you have a lot of rams in your code.

If you are using the GUI, the options are set on the **Compiler** tab in Project Settings. If you are using the command line compiler, use the -S option:

`-S+parfunc`	Detection of simultaneous function calls is on.
`-S-parfunc`	Detection of simultaneous function calls is off.
`-S+parmem`	Detection of simultaneous memory accesses is on.
`-S-parmem`	Detection of simultaneous memory accesses is off.
`-S+parchan`	Detection of simultaneous channel accesses is on.
`-S-parchan`	Detection of simultaneous channel accesses is off.

`-S+parmem` will only detect simultaneous accesses to different addresses within a memory, not simultaneous accesses to the same address.

6.6 Simulation compilation control options

To control the way that a simulation is compiled, you can pass options to the backend compiler. (The backend compiler is specified when you install DK.)

`-cl`	Specify the backend compiler command line
-be	Pass options to backend compiler

6.6.1 Pass options to command line

If you are using the command line compiler, the `-cl"`***`CommandLine`***`"` option can be used to pass options for compiling the code for simulation. Handel-C code is converted into a temporary C++ file, and this is then compiled by the backend compiler, so that it can run on a host machine.

If you are not using the command-line compiler, the ***`CommandLine`*** option can be passed to the back-end compiler via the string in the **Simulator compilation command line** box in the **Linker** tab of the **Project Settings** dialog

The ***`CommandLine`*** option is a quoted string consisting of the command to be executed by the compiler. There are various parameters the compiler can provide:

%1 : Name of the temporary C++ source file generated from Handel-C

%2 : `.dll` output file name

%3 : output file root

%4 : string passed to -be option

Examples

```
handelc -s file.hcc -cl"g++ -c -O2 %1 -o%3.obj"
```

generates a .cpp file for the simulator (for example, called xyz.cpp) and then runs the command:

```
g++ -c -02 xyz.cpp -ofile.obj
```

```
handelc -s -cl"g++ %1 -o%3.exe %4" -be"vga.lib" fred.hcc
```

generates a .cpp file and then runs the following command:

```
g++ temp.cpp -ofred.exe vga.lib
```

The options to target GCC are different, depending on whether you are targeting the new simulator (`G++`) or the old simulator (`GCC`).

6.6.2 Pass options to backend compiler

The -be"***String***" option can be used to pass extra options to the backend compiler.

Handel-C code is converted into a temporary C++ file, and this is then compiled by the backend compiler, so that it can run or be simulated on a host machine.

The ***String*** option is a quoted string that replaces the %4 variable in the command line used to invoke the backend compiler. This command line may be that defined in the HANDELC_SIM_COMPILE environment variable or that defined in the -cl build option. If the %4 variable is not present in the command line, the -be"***String***" option will not be used. No checks are performed on the string value.

Examples

`handelc -s aloha.hcc -cl"bill %1 %2 %4" -be"gibbons and apes"`

generates a temporary .cpp file for the simulator (for example, ***xyz***.cpp) and then runs the command:

`bill xyz.cpp aloha.dll gibbons and apes`

If the HANDELC_SIM_COMPILE environment variable has been set to `cl /LD %1 %3.obj %4 -Fec.dll`

`handelc -s driver.hcc -be"vga1.lib"`

generates a temporary .cpp file for the simulator (for example, ***xyz***.cpp) and then runs the command:

`cl /LD xyz.cpp driver.obj vga1.lib -Fec.dll`

6.7 Environment variables

The Handel-C compiler has three environment variables associated with it.

- HANDELC_SIM_COMPILE is an alternative to the -cl command line option. It is used to create the simulation file when compiling using the command line.
- CELOXICA_DK_HOME is the DK install directory. For example, if you install in the default location, CELOXICA_DK_HOME is `C:\Program Files\Celoxica\DK`
- The value of HANDELC_CPPFLAGS is passed as command line options to the preprocessor each time the compiler is executed.

The DK installation sets the HANDELC_CPPFLAGS variable to contain the –C option. The -C option passes source code comments through to the compiler.

To change the environment variables use the facilities described in the installation instructions.

Temporarily changing environment variables

You can temporarily alter the value of the variable by typing the following at the command prompt:

set HANDELC_CPPFLAGS=***Command Line Options***

For example:

```
set HANDELC_CPPFLAGS=-C -DDEBUG
```

7. Simulation and debugging

7.1 Using the simulator

The simulator enables you to test your program without using real hardware. It allows you to see the state of variables in your program at every clock cycle.

You can view information about the simulation in various windows:

- See the clocks in use and the threads currently running in the **Clocks/Threads** window
- See the current function, and what functions were called to reach it, in the **Call Stack** window
- Select variables to be displayed in the **Watch** and **Variables** windows

You can run code in the simulator in several ways:

- Run until the end
- Run until you reach the current cursor position
- Run until you reach a user-defined breakpoint
- Step through the code one clock cycle at a time
- Advance through code one execution point at a time
- Pause the simulation

7.1.1 Swapping between simulators

DK has a new faster simulator. You are recommended to use the new simulator unless you have an asynchronous path from an input to an output where the input is fed from a plugin which changes the value faster than the clock and the output has a plugin which reads data faster than the clock. The fast simulator will give incorrect results, since it is a time cycle accurate simulator.

To swap between simulators:

- Open the **Linker** tab in Project settings. Tick the **Use netlist simulator** box to use the old simulator, or untick it to use the new simulator. The default **Simulator compilation command line** changes when you tick and untick the box.

OR

- If you are using the command line compiler, use the -s option to target the new simulator and -s -ns option to target the old simulator.

7. Simulation and debugging

7.1.2 Starting debug and simulation

From the **Build** menu select **Start debug**. The **Debug** menu appears in place of the **Build** menu.

- Where the code includes multiple threads using separate clocks you need to select a clock. The first thread associated with that clock becomes the current thread.
- You can step through the code from one clock cycle to the next. Statements that are completed at the end of the current clock cycle are marked with an arrow.

  Alternatively you can advance through code from execution point to execution point, or use breakpoints to halt the debugger at any selected line in the code.
- You can use the Waveform Analyzer to inspect signals on outputs and generate signals for inputs

7.1.3 Debug symbols in the editor window

Statements associated with the current clock cycle are marked with arrows. All these statements execute together. If you single-step or advance through the code, you will see the arrows move.

In the current thread

The yellow arrow marks the current execution point. When you are stepping through code, it marks the point in the code that will consume a clock cycle on that thread.

White arrows mark all other code executed in the current clock cycle in the current thread. They mark "control logic"; control statements that lead to the execution point marked by the yellow arrow.

Green arrows mark current function calls. This gives a stack trace for the current thread.

In other threads

The equivalent of yellow, white and green arrows are all marked grey in other threads. To see them, you must switch to that thread.

Other symbols

Active breakpoint

Disabled breakpoint

Enabled and disabled breakpoints on same line

Pointer to error and browse results

7.1.4 Selecting a clock

If you are simulating a project with multiple clocks, a **Select Clock** dialog will pop up asking you to select which clock to use when you start the simulation.

During simulation the Clocks/Threads window shows all clocks in use. The selected clock is the one associated with the current thread. To select a different clock, follow a different thread.

7.1.5 Selecting a thread to follow

In debug the **Clocks/Threads** window shows all the running threads. The thread currently followed by the simulator is in bold. You can change the followed thread in three ways.

Selecting a thread in the code editor

1. Click a code line marked with a grey arrow within the thread you want to follow. (Grey arrows mark execution points in other threads).
2. Right-click the mouse and select **Follow Thread** from the shortcut menu.
3. If a single thread is active at the code line, the menu option identifies it. If several threads are active, you can select the thread you want from a dropdown list. Thread identifiers match those shown in the **Clocks/Threads** window.

Setting a breakpoint in the code editor

Set a breakpoint in the thread you want to follow. When the breakpoint is reached, that thread becomes the current thread.

Selecting a thread in the Clocks/Threads window

Open the **Clocks/Threads** window, select a thread, right-click and select **Follow Thread**.

7.1.6 Following function calls in the Call Stack window

The way a function has been called is displayed in the **Call Stack** window. This shows the current function at the top of the window, and the uncompleted functions that called it beneath.

Debug symbols in the Call Stack window

Yellow arrow marks the current function in the current thread.

Green arrows mark function calls on the stack (showing the path of calls to reach the current function).

Breakpoint marker indicates that there is a breakpoint on the line. The breakpoint marker may be red (enabled), white (disabled) or grey (enabled and disabled on same line).

7.1.7 Examining variables

During debug you can examine variable values in two windows:

- Watch window (**View**>**Debug windows**>**Watch**)
- Variables window (**View**>**Debug windows**>**Variables**)

By default variables are displayed in decimal. To change the base, right-click in the selected window and select a new base from the pop-up menu.

You can change the display base of an individual variable using the Handel-C specification `with {base=`***n***`}`.

You can turn off the display of a variable by using the Handel-C specification `with {show = 0}`. For example:

```
int 32 pike with {show = 0};
```

Arrays and structures are displayed with a + button next to the name. Click on this button to display individual array elements or structure members.

7.2 Using the debugger

You can use the debug commands to:

- Step through every clock cycle in your code
- Advance through every execution point in your code
- Set and remove breakpoints to segment the simulation
- Follow a selected processing thread or clock
- View the clock cycle count
- See how a function has been called
- Examine variables

You can also use the `extern "`***Language***`"` construct to link to standard C and C++ libraries to use the `printf`/`cout` functions and other standard file I/O functions.

7.2.1 Generating debug information

When you compile your project in Debug mode, you can choose to generate debug information. This allows you to step through your code and examine the code called on each clock cycle, or at each execution point.

- If you are using the command line compiler, use the `-g` option to generate debug information.
- If you are using the GUI compiler, select **Generate Debug information** on the **Compiler** tab in Project Settings.

In previous versions of DK you could select the **Generate Debug information** option for Verilog, VHDL and EDIF modes to produce a more structured output. In DK2, all HDL output is structured, so the option is no longer available.

7.2.2 Debug project configuration

The default settings for the Debug project configuration are those to enable you to debug a project.

The Project Settings specific to debug are:

Preprocessor	defines the variables DEBUG and SIMULATE. This allows you to set up the code according to whether you are using the simulator, e.g. use simulator channels instead of real interfaces.
Compiler	**Generate Debug** and **Generate warning** boxes checked.
Linker	**Output format** set to **Simulator.**
	Save browse info box checked.
	Generate estimation information option (create HTML files) switched off.
	Exclude timing constraints (-notcon) unchecked
Debugger	**Working directory** for debugger set to current (.).
Optimizations	High-level optimization switched off.

7.2.3 Stepping through code

In a sequential language such as ANSI-C, you can step through code one line at a time, and you stop at an execution point. In Handel-C, you step through code one clock cycle at a time. You can use Advance to move through code one line at a time.

Because Handel-C is a parallel language, there can be multiple execution points. Where a par statement is found in your code the execution splits into separate threads, one for each branch of the par statement. The threads will wait until they have all completed before the main thread of the code can continue after the par block.

When you are debugging you can only follow one thread at a time. The simulator steps through the thread you are following one clock cycle at a time. If other threads within a parallel block require more clock cycles, these clock cycles will not be stepped through. The clock cycle count in the **Clocks/Threads** window increases when you leave the par block to show the number of clock cycles required by the longest thread.

Single stepping

To step through your code, select **Build**>**Start Debug**>**Step Into**. You can continue stepping by pressing F11.

The step that is currently executing is shown by a yellow arrow. If other code in the same thread is executed in the same clock cycle this is shown by white arrows. You can advance to this code, but not step to it, as it doesn't take any clock cycles.

In addition to statements that take clock cycles, you can also step to breakpoints or to function or macros calls. You can choose to **Step Into**, **Step Out** of or **Step Over** functions and macros.

Stepping through code: example

This example illustrates the behaviour of debugger arrows when you are stepping through your code.

To run the example, open the `Debug_arrows.hw` workspace in DK by double-clicking it. The example is in ***`InstallDir`***`/DK/Examples/Handel-C/ExampleDebug/`.

Stepping through the example

1. Build the project in Debug mode by selecting **Build debug_arrows** from the **Build** menu.
2. Step into the code by selecting **Build**>**Start Debug**>**Step Into** or pressing F11. Press F11 again
   The yellow arrow (current execution point) should be at the function call to `blob()` and the green arrow should be at the start of the `main()` function. The white arrows show other code executed on the same clock cycle.
3. Open the Clocks/Threads window (**View**>**Debug Windows**>**Clocks/Threads**).
4. Step over the `blob()` function by selecting **Debug**>**Step Over** or pressing F10.
   The yellow arrow should be at the `y = 3` statement after the call to the `blob()` function. If you had stepped into the function (F11) instead, the yellow arrow would be within the `blob()` function. Notice that the number of clock cycles reported in the Clocks/Threads window is 1.
5. Press F10 again.
   The yellow arrow should be at the first `delay` statement within the `par` block, and the number of clock cycles reported will have increased to 2.
6. Press F10 again.
   The yellow arrow will be at the `y = 0` statement.
   Note that the number of clock cycles has increased to 4. This is because the other thread in the `par` statement takes two clock cycles (two `delay` statements), and the current thread cannot continue executing until the parallel thread has finished.
7. To exit the simulation, select **Debug**>**Stop Debugging**.

7.2.4 Advancing through code

If you step through your code you will move forward one clock cycle at a time. To move forward a single execution point rather than a complete clock cycle, use the **Advance** command or select **Debug**>**Advance**. You must Step Into your code before you can use the Advance, by pressing F11.

7.2.5 Arrow behaviour during step and advance

When you are stepping or advancing through your code, yellow and white arrows mark the execution points.

A subset of execution points, which include assignments and other statements that take clock cycles and function calls, may be stepped over or into. All execution points may be advanced to.

Step statements

Step statements are statements that take a clock cycle, function calls and any statement that has a breakpoint set.

When you are stepping through code, a yellow arrow marks the current step statement and white arrows show the other execution points associated with that step statement. You can also Advance to any of these steps.

Any assignment (=, ++ , -- , += , -= , *= , %= , <<= , >>= , &= , |= , ^=)

`return(` ***Expression*** `);` (If Expression is assigned on return)

`Channel ?` ***Variable*** ;

`Channel !` ***Expression*** ;

`releasesema();`

`delay;`

`Function()`

`prialt (...);` (Where no default clause coded)

Advance statements

Advance statements are executable statements that do not take any clock cycles. (Functions and statements that have breakpoints set are special cases. These are treated as step statements rather than advance statements.)

You cannot step to Advance statements. When you are advancing through code, the current advance statement is marked by a yellow arrow. When you are stepping through code, advance statements are shown as white arrows, when associated with the current step statement.

`{...}` (netlist (old) simulator)

`return;`

`return(`***Expression***`);` (If ***Expression*** is not assigned on return)

`while (`***Expression***`){…}`

`if (`***Expression***`){…}` else {…}

`do {...} while (`***`Expression`***`);` (The "do" part is considered the active point rather than the "while" part)

`switch (`***`Expression`***`){...}`

`break;`

`goto `***`Label`***`;`

`continue;`

`prialt (...);` (Where a default clause exists)

for loops

```
for        // white arrow for Step, yellow arrow for Advance
(Init;     // yellow arrow for assignment on first pass if you are stepping
Test;      // no arrow
Iter)      //yellow arrow for assignment
{...}
```

No execution point. Lines ignored by Advance and Step: no arrows displayed

The following lines of code do not have any execution point. You cannot step or advance to these lines and you cannot set breakpoints on them.

`{...}` (new simulator)

`par`

`seq`

`par | seq (`***`index_Base`***` ; `***`index_Limit`***` ; `***`index_Count`***`)` (No true assignments involved)

All declarations

`ifselect`

`assert`

7.2.6 Using breakpoints

Breakpoints give you an alternative to stepping through code.

You can set breakpoints on any line of code that contains an execution point.

When the debugger reaches a breakpoint it pauses until you request it to continue. You can restart the simulation by selecting **Debug**>**Restart**.

If you set breakpoints on statements in a par block, all breakpoints will be hit as you run through the code, but the order in which they are hit is undefined.

7. Simulation and debugging

You can carry out more complex actions using the Breakpoints dialog (**Edit**>**Breakpoints**).

Setting breakpoints

1. Select the line of code where you wish the simulator to pause. To search for known names, use **Edit**>**Find**.
2. Click the **Insert/Remove Breakpoint** button

   OR

   Right-click the mouse and select **Insert Breakpoint**.

Multiple breakpoints on the same line

A breakpoint can be active or inactive. You might wish to have two breakpoints on the same line, set to break according to different conditions, and have one of them active and one inactive, depending which thread you were following.

You can have multiple breakpoints on the same line by entering the same line twice in the **Edit**>**Breakpoints** dialog box. You can disable a breakpoint by unchecking its box in this dialog and enable it by checking the box.

Disabling breakpoints

A breakpoint can be active or inactive.

If you wish to keep a breakpoint but not to stop at it:

1. Move the cursor to the line of code where the breakpoint is set.
2. Right-click the mouse.
3. Select **Disable Breakpoint**.

All breakpoints are listed in the **Edit**>**Breakpoints** dialog box. You can also disable a breakpoint by unchecking its box in this dialog.

Removing breakpoints

- Find the line of code where the breakpoint is set.
- Click the breakpoint button

  OR
  Right-click the mouse and select **Remove Breakpoint**

  OR
  Open the breakpoints dialog (**Edit**>**Breakpoints**), select the breakpoint(s) to be removed and click **Remove**.

Setting breakpoints in macros and inline functions

If you set a breakpoint in an inline function or a macro procedure, the breakpoint will occur every time that the code is used.

You cannot set a breakpoint in a macro expression.

Breakpoints in replicated code

If you set a breakpoint in replicated code, a breakpoint is set in every copy of the code. When you step through the code all of these breakpoints are stepped over simultaneously.

The clock cycle counter in the **Clocks/Threads** window is not incremented until you have passed through all the breakpoints set in a single clock cycle.

8. Optimizing code

8.1 Logic estimator

The Handel-C compiler can give information on logic usage and depth to help you optimize your designs. (Note that this information is based on estimates, since full place and route is needed to get exact logic and area information.)

Logic estimation information is only available for EDIF builds. You cannot use the Logic estimator from Nexus PDK.

Using the logic estimator

To generate information about the logic area and depth of your code:

- check the **Generate estimation info** box on the **Linker** tab of the **Project Settings** dialog.

  OR

- use the -e option on the command line compiler. For example:
  `handelc -e -fs -g test.hcc`

The information generated is most detailed for builds targeting devices supported by the Technology Mapper (with the technology mapper enabled).

When you compile your code a set of HTML files will be produced, containing

- line by line information on use of resources (e.g. NAND gates, or look-up tables for mapped-EDIF)
- description of the longest combinational paths in your code:

You can access the information by opening the overview page `summary.html` in an Internet browser. `summary.html` will be placed in the build directory for your project.

8.1.1 Logic area and depth summary

You can view logic area and depth information about your code by opening the file `summary.html` in an Internet browser. The file is created in your build directory by the logic estimator if you have selected the **Generate estimation info** option.

Area estimation information

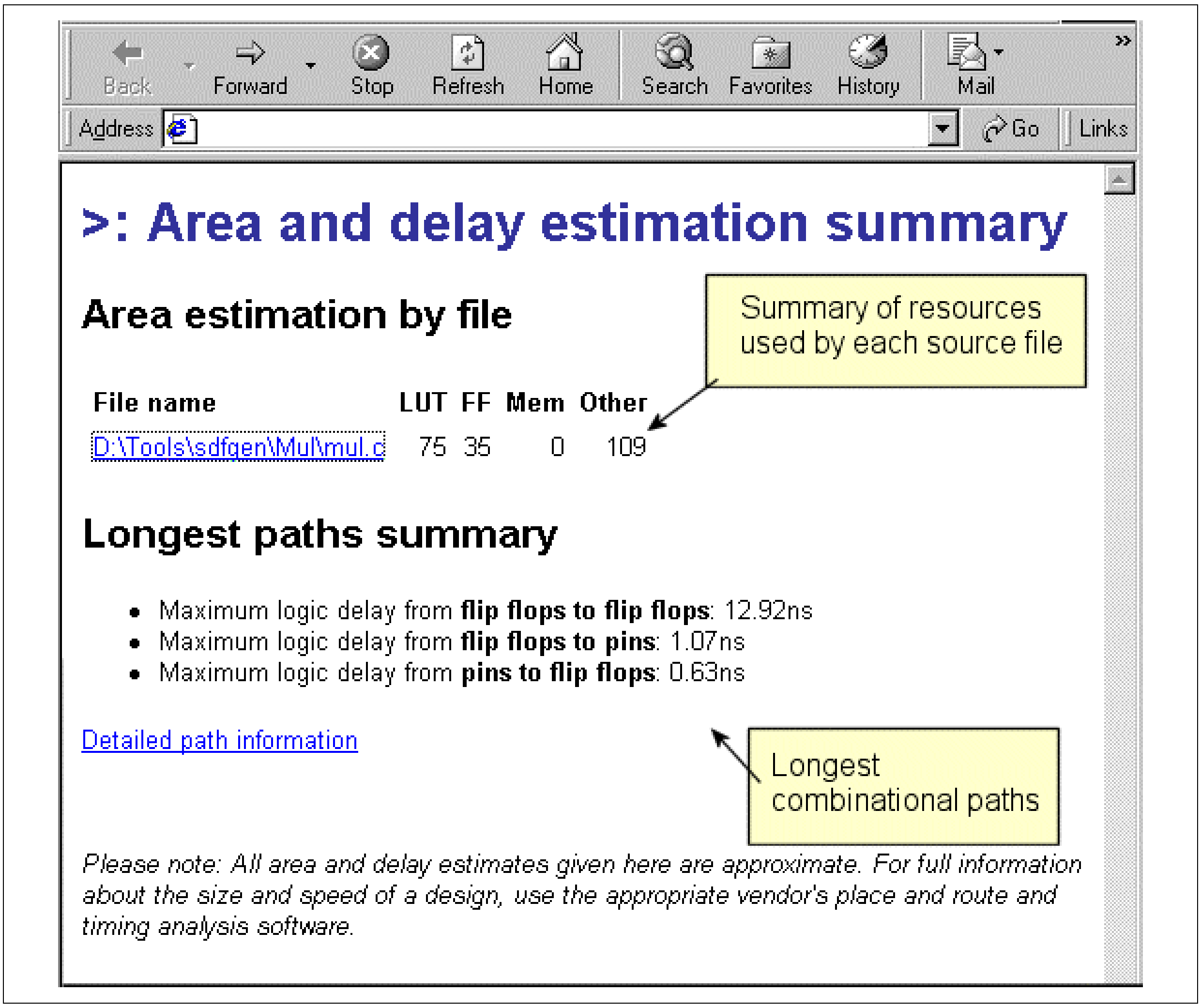
- For technology mapped-EDIF: consists of the number of look-up tables, flip-flops, memory resources and other device-specific logic resources (listed under “Other”).
- For non-mapped EDIF: consists of the number of NAND gates, flip-flops and memory resources.

Each source file listed is linked to more detailed logic area information.

Longest paths summary

`summary.html` displays estimates of the longest path for each combination of flip-flops to/from pins, RAMs and flip-flops, pins to/from RAMs and pins, RAMs to RAMs. It also links to more detailed combinational path information.

8.1.2 Area and delay estimation example



8.1.3 Information on logic area

The detailed information about area provided by the logic estimator consists of the number of resources created for each line of your source code. Totals are summarized in the overview page, `summary.html`.

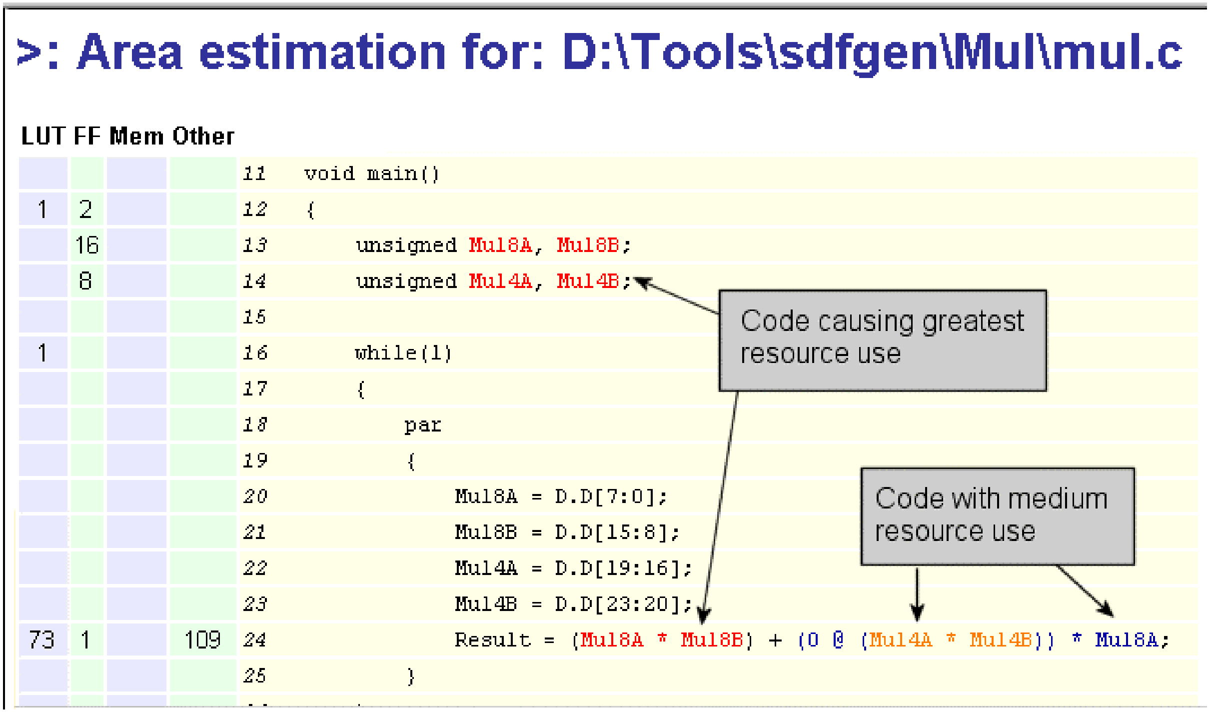
8. Optimizing code

For each line of code, the areas that use the greatest resource in that line are highlighted in colour. Red code provides 75% or more of the maximum, orange code 50 -75% of the maximum, and blue 25 - 50%. Black code contributes up to 25% of the maximum.

The number of resources used is listed next to each line of code. Resources listed are:

- LUT: look-up tables (mapped-EDIF output only)
- NAND: NAND gates (non-mapped EDIF only)
- FF: Flip flops
- Mem: Memory bits
- Other: device-specific logic elements (mapped-EDIF output only):
  - Altera: CARRY_SUM, CARRY
  - Xilinx: MuxF5, MuxF6, MuxF7, MuxF8, MuxCY, XORCY, MultAND

Logic area estimation example



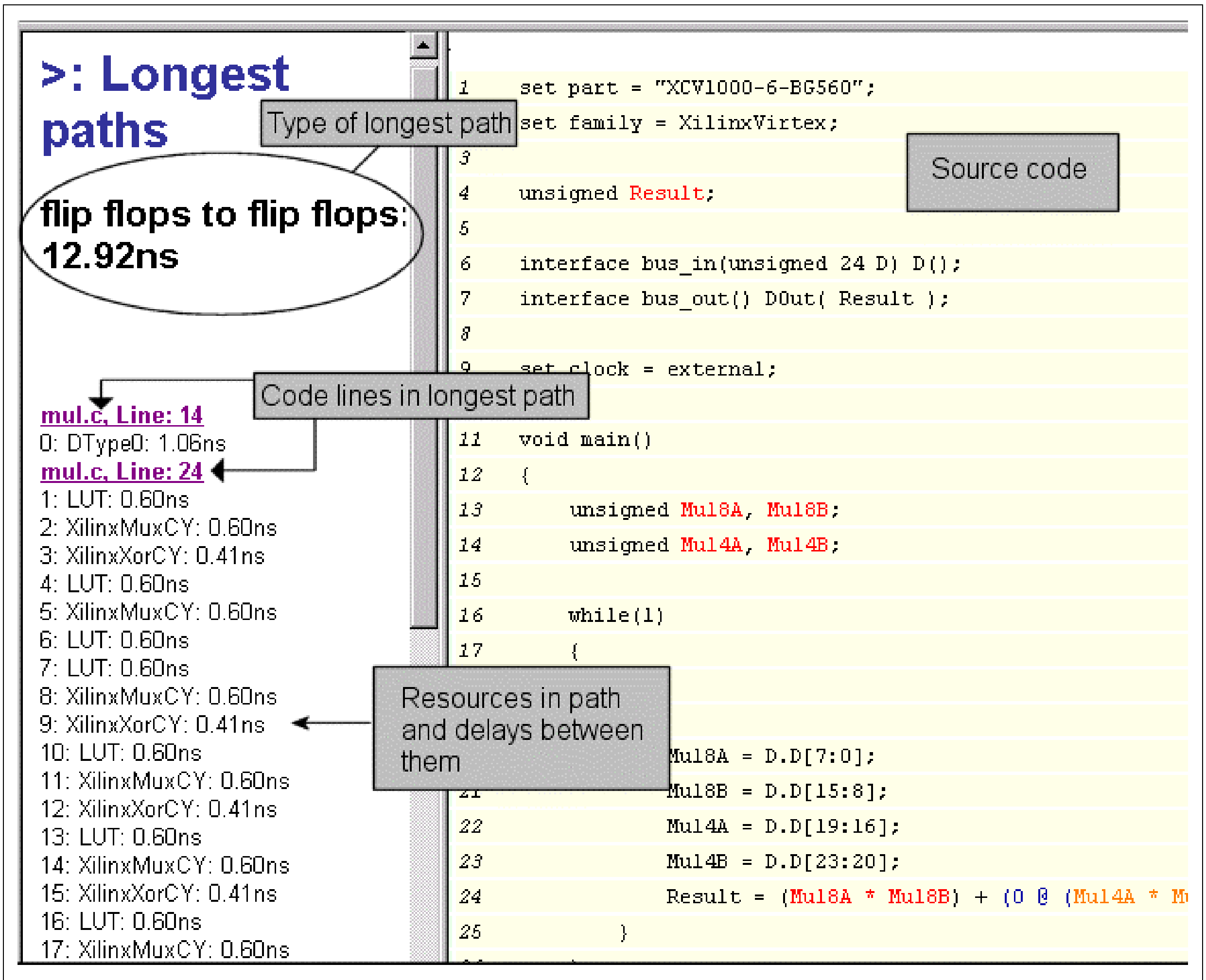
8.1.4 Information on combinational paths and delay

Information on logic delay generated by the logic estimator is summarized on the overview page, summary.html. This contains a link to more detailed information, where the longest combinational path is given for the following 9 path types. If that path does not exist, it is not included:

	From flip-flops	**From pins**	**From RAM**
To flip-flops	•	•	•
To pins	•	•	•
To RAM	•	•	•

For each of the longest paths, there is a list of the lines of source code that contribute to the path, and a list of resources used by each of these lines.

Logic depth estimation example



8.2 Optimizing code example

The optimizing code example is based on a windowing program. Windowing is a technique which can be used to improve the results of the discrete Fourier transform. The program reads in 15 samples at a time, and multiplies them by a symmetrical window.

The original program is optimized to run in software. The example shows how the program can be optimized to run in hardware in two stages. The logic estimator allows you to view the effects of each of the optimization stages.

Each version of the program is contained in a different project (Opt1, Opt2 and Opt3) within the same workspace: `DK\Examples\Handel-C\ExampleOpt\optexample.hw`.

Double-click on the workspace file to open the example in DK. You need to set the example to build in EDIF. You can only target EDIF from the full version of DK, not from Nexus PDK.

8.2.1 Optimizing code example: original program

The original program is in `optexample1.hcc`. The program applies a series of multiplications to input data (this is the windowing technique). The multiplications are in a `while` loop which runs as long as data is fed into it.

The code is written with the windowing loop unrolled (each multiplication step is in sequence) as this can be efficient for software implementation. However, in hardware, each of the 15 calls to the `MULT` macro will instantiate a separate multiplier, which is not area efficient.

Note that, apart from `WindowParameters[7]`, each window parameter is used twice. The Handel-C compiler identifies this and only builds the logic for each different multiplier once. This is then shared for each of the two multiplications.

Build the original program (Opt1 project) and view logic estimation information. Then look at the next version of program in Opt2.

8.2.2 Building the optimizing code example

If you have the full version of DK, you can build each of the versions of the optimizing code example, and view the results generated by the logic estimator by following the steps below. You cannot target EDIF or use the logic estimator if you have Nexus PDK.

Opening the example and checking project settings

1. Open the workspace file (***`InstallDir`***`\DK\Examples\Handel-C\ExampleOpt\optexample.hw`) in DK by double-clicking on it.
2. Select the project you want to build: **Project**>**Set Active Project**.

3. Set the build configuration to EDIF: select **Build**>**Set Active Configuration**, then click on EDIF below the project you want to build and press **OK**.
4. Note that
   - the **Generate estimation info** and **Use Technology mapper** options are selected on the **Linker** tab in Project Settings.
   - most of the compiler optimizations are selected on the **Optimizations** tab in Project Settings.
5. To view the files in the workspace, check that you are in file view and click on the + sign to the left of the chip icon.
6. To examine the code, double-click on the relevant Handel-C files in the workspace pane.

Building the example

7. Build the example by pressing F7.
   You will see description of the compiler optimization steps in the bottom left-hand corner of the DK window. This will say **Ready** when the build has completed.

Examining the information produced by the Logic estimator

8. Browse to the EDIF directory for the project you have built. For example, if you have built Opt1, browse to `*InstallDir*\DK\Examples\Handel-C\OptExample\Opt1\EDIF`
9. The EDIF directory will contain a number of HTML files. Open `summary.html` in an Internet browser.
   This will show you an area and delay estimation summary for the project, with links to more detailed information.

8.2.3 Optimizing code example: stage 1

The second stage in optimizing the example is in `optexample2.hcc`.

Optimizations

In the Opt2 project, the code has been optimized by using a shared function for the multiplier:

```
unsigned 32 Mult( unsigned 24 A, unsigned 8 B )
{
   return (0 @ A) * (0 @ B);
}
```

Results of optimization

The shared multiplier results in considerably smaller hardware. However, there is considerable logic associated with the function calls, as the data from each of the 15 calls has to be multiplexed to the single multiplier. This also has an associated speed penalty as a multiplexor has some delay associated with it.

Viewing the results

Build optexample2.hcc (Opt2 project).

Open the summary.html page for this project and for the previous project (Opt1) to compare the delay and estimation information.

You should see the following changes:

- the number of look-up tables (LUT) has decreased
- the maximum logic delay from flip-flop to flip-flop has increased slightly

Then look at the next version of program in Opt3.

8.2.4 Optimizing code example: stage 2

The second stage in optimizing the example is in optexample3.hcc.

Optimizations

In the Opt3 project the main while loop is rewritten so that the multiply operation is only called from a single point in the code (this is called 'loop rolling'):

```
while(1)
   {
      par
      {
         ...
         DataWindowed = (0 @ DataInReg) * (0 @ WindowCoefficient);
         ...
      }
   }
```

The multiply operation takes data from DataInReg and WindowCoefficient and places it in DataWindowed.

The remaining code in the par statement makes sure that WindowCoefficient has the correct coefficients on each step.

```
      par
      {
         WindowCoefficient = WindowParameters[Index];
         DataWindowed = (0 @ DataInReg) * (0 @ WindowCoefficient);
         Index += Direction ? -1 : 1;
         if ( Direction == 0 && Index == 6 ) Direction = 1;
         else if ( Direction == 1 && Index == 1 ) Direction = 0;
      }
```

The window coefficients are stored in a dynamically indexed ROM:

```
static rom unsigned 8 WindowParameters[8] =
{
   0, 13, 48, 99, 156, 207, 242, 255,
};
```

This is an efficient storage mechanism for relatively small numbers of values on Xilinx devices.

Results

The hardware for the final version of the windowing program is smaller and faster than either of the previous versions.

Viewing the results

Build `optexample3.hcc` (Opt3 project).

Open the `summary.html` page for this project and for the previous projects (Opt2 and Opt1) to compare the delay and estimation information.

You should see the following changes:

- the number of look-up tables (LUT) has decreased (less than for Opt2 and Opt1)
- there are now some memory bits (Mem), due to putting the window coefficients in ROM
- the maximum logic delay from flip-flop to flip-flop has decreased and is less than that for Opt2 and Opt1

9. Tutorial examples

9.1 Example 1: accumulator

The workspace for Handel-C tutorial example 1 is in ***InstallDir***\DK\Examples\Handel-C\Example1.

The program takes a number of values from a file and calculates the sum of those values. It illustrates the basics of producing a Handel-C program and demonstrates the use of the simulator.

The data is read from a sample file sum_in.dat (provided in the Example1 directory) and results are written to a file sum_out.dat (in the Example1 directory).

9.1.1 Compiling and simulating Example 1

1. Open the workspace file (DK\Examples\Handel-C\Example1\Example1.hw) by double-clicking on it. DK starts with the Example1 workspace open.
2. Check that you are in **File View** in the Workspace window and click on the + sign to the left of the chip icon to see what files are within the project.
3. To examine the code, double-click the file sum.hcc in the Workspace window. If you cannot see it, you can make the Workspace window larger by dragging its borders.
4. Build the project in Debug mode by selecting **Build Example1** from the **Build** menu. Messages from the compiler appear in the output window. They give an approximation of the number of hardware gates required to implement the program.
5. Start the debugger by pressing F11 to step through the simulation, or F5 to run to the end. The simulator reads the contents of values from the file sum_in.dat, sums them, and writes the result to the file sum_out.dat.

   To watch the accumulation progressing in the variable sum, open a Watch window (select **View**>**Debug Windows**>**Watch** or type Alt+3) and type sum in the window.
6. The simulator terminates at the end of the program.
7. Examine the files to ensure that the output file contains the correct result. If you wish to change the values in sum_in, ensure that each value is placed on a separate line.

9.2 Example 2: pipelined multiplier

The workspace for Handel-C tutorial example 2 is in ***InstallDir***`\DK\Examples\Handel-C\Example2`.

The program performs multiplication using a replicated parallel structure to create a pipeline.

The operands used are the initialization values to the arrays of `leftOps` and `rightOps`, such that the `results[`***n***`] = leftOps[`***n***`] * rightOps[`***n***`]`.

This multiplier calculates the 16 LSBs of the result of a 16-bit by 16-bit multiply using long multiplication. The multiplier produces one result per clock cycle with a latency of 16 clock cycles. This means that although any one result takes 16 clock cycles, you get a throughput of 1 multiply per clock cycle. Since each pipeline stage is very simple, combinational logic is shallow and a much higher clock rate is achieved than would be possible with a complete single cycle multiplier.

At each clock cycle, partial results pass through each stage of the multiplier in the sum array. Each stage adds on 2^n multiplied by the b operand if required. The LSB of the a operand at each stage tells the multiply stage whether to add this value or not.

Operands are fed in on every clock cycle on signals `leftOp` and `rightOp`. Results appear 16 clock cycles later on every clock cycle on signal `result`.

9.2.1 Example 2: Index array test code details

```
/*
 * Index at end of array macro
 */
#define IndexAtArrayEnd(Index, ArrayLimit) \
    select(exp2(width(Index)) == (ArrayLimit), !(Index), ((Index) == \
                                                            (ArrayLimit)))
```

The `IndexAtArrayEnd` macro tests if the index of size `ArrayLimit` is at the end of an array, whatever width the index counter has been assigned by the compiler. In most cases, this is a normal comparison, but if the index overflows, the test will compare the overflow value. An example is an index of size 4. The compiler will assign the index a width of 2 bits (to store the values 0 – 3). When it is compared against 4, the index will hold the value 0 (as the most significant bit has been lost) and the compiler will generate an error. In this case, the `IndexAtArrayEnd` macro compares against 0 instead of against 4.

This implies that such a comparison cannot be made at the start of the cycle, when element zero is being processed, but only at the end of the cycle after the index has been incremented.

9.2.2 Compiling and simulating Example 2

To compile and simulate the pipelined multiplier, open the workspace in the `Examples\Handel-C\Example2` directory and select **Build Example2** from the **Build** menu. You can then start the debugger.

1. Open the workspace file (`DK\Examples\Handel-C\Example2\Example2.hw`) by double-clicking on it. DK starts with the Example2 workspace open.
2. Check that you are in **File View** in the Workspace window and click on the + sign to the left of the chip icon to see what files are within the project.
3. To examine the code, double-click the file `parmult.hcc` in the Workspace window. If you cannot see it, you can make the Workspace window larger by dragging its borders.
4. Build the project in Debug mode, by selecting **Build Example2** from the **Build** menu. Messages from the compiler appear in the output window. They give an approximation of the number of hardware gates required to implement the program.
5. Start the debugger by pressing F11 to step through the simulation. Press F11 again to proceed to the next step.
6. Check that you are following a thread in the `parMult()` function. The yellow arrow (current thread) should be next to a line within the `parMult()` function in the lower part of the code in `parmult.hcc`. If it is not, right-click on one of the grey arrows next to the `parMult()` function and select **Follow Thread**.
7. Open the **Variables** window (**View**>**Debug Windows**>**Variables**) and select the **Locals** tab.
8. View the values propagating through `xx` (intermediate value of left operand) by clicking on the + sign next to `xx` and the pressing F10 several times. You can also view the values in `yy` (intermediate value of right operand) and `rr` (intermediate result).
   Each time you press F10, one stage of the pipeline will be completed. After 18 clock cycles, the first result is available, and subsequent results are provided on successive clock cycles.
9. To stop simulation, select **Debug**>**Stop Debugging** (Shift F5). The simulation will not stop by itself.

9.3 Example 3: queue

The workspace for Handel-C tutorial example 3 is in ***InstallDir***`\DK\Examples\Handel-C\Example3`.

The example shows how to create parallel tasks and how to communicate between those tasks. It also illustrates arrays of variables and arrays of channels. The example shows a project containing independent main functions which are implemented independently in hardware.

There are two source files: queue.hcc handles the queue function, while main.hcc provides I/O facilities. Definitions common to both files are given in queue.hch. They both have a clock set (in this case, the signal on pin 1 is used for both functions).

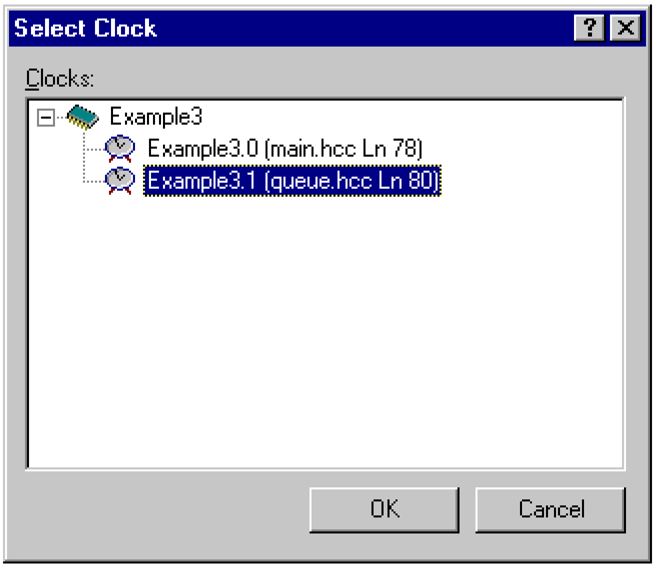
The queue function code illustrates the use of parallel tasks and channel communications by implementing a simple four-place queue. Each task holds one piece of data and has an input channel connected to the previous queue location and an output channel connected to the next queue location.

At each iteration, the data moves one place up the queue. The program executes an infinite loop, and you must use **Stop Debugger** to terminate the simulation.

The queue presented here is parameterized on the width of the input and output channels because the width of all internal variables are undefined and inferred by the compiler.

9.3.1 Compiling and simulating example 3

1. In the `Examples\Handel-C\Example3` directory, double-click on the workspace file `Example3.hw`.
2. Build the example in Debug mode by selecting **Build**>**Build Example3**.
3. Press F11 to step into the program in the debugger.
4. When you start debugging, you will be asked to select a clock to follow. Select the queue function clock (tagged with the file name queue.hcc).



5. To view local variables, select **View**>**Debug Windows**>**Variables** or press Alt+4. Then select the **Locals** tab at the bottom of the window.
   The Variables window shows the variables local to the function. Press F11 repeatedly to step through the code, and watch the values change.

6. To watch the queue in the debugger, open the Watch window (**View**>**Debug Windows**>**Watch** or press Alt+3). Click on the + next to the `state` variable to display a list of the array elements. Press F11 repeatedly to step through the code, and watch the values change.
7. To stop simulation, select **Debug**>**Stop Debugging**.

9.3.2 Example 3: detailed explanation

This example uses four parallel tasks, each containing one word of data. At each iteration, one word is passed from one task to another in a chain like this:

State [0] --> State [1] --> State [2] --> State [3] -->

The links between the processes are entries in the `links` array of channels. Input and output to and from the system is handled by the `main` function.

Communication between the two functions is handled by an array of channels.

The queue only reads data and writes data on every other clock cycle.

A replicated pipeline is used to implement the queue. The first and last entries in the pipeline are treated differently by using a `select` expression or an `ifselect` statement to differentiate them at compile time.

9.4 Example 4: clients / server communication

The workspace for Handel-C tutorial example 4 is in ***`InstallDir`***`\DK\Examples\Handel-C\Example4`.

The clients and server are implemented as independent pieces of hardware, communicating via channels. The server reads data from an array of channels from the clients and puts the results in a queue as they arrive. They are read from the queue by a dummy service routine. This is where the client requests could be processed by a real server routine.

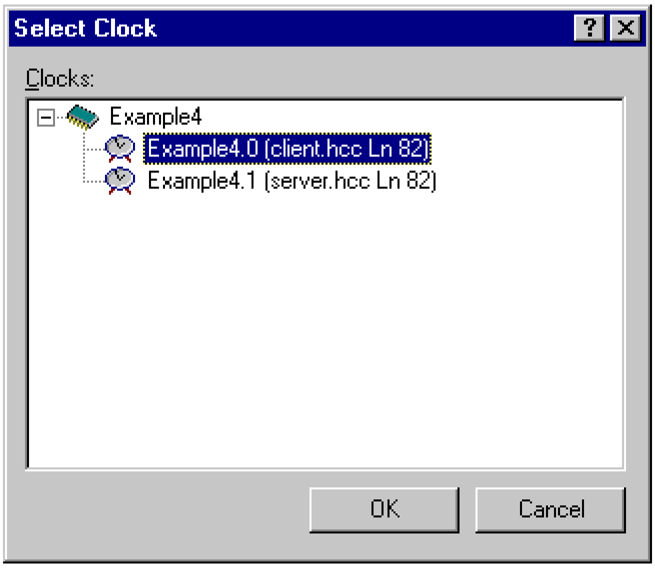
The server clock runs at half the speed of the client clock to allow time for complex assignments during request processing.

There is a pair of identical client functions. These functions merely select valid requests from an array and send them to the server.

9.4.1 Compiling and simulating example 4

1. Double-click on the workspace file `Example4.hw` in the `Examples\Handel-C\Example4` directory.
2. Build the example in Debug mode by selecting **Build**>**Build Example4**.
3. Step into the program within the debugger by pressing F11.

4. When you start debugging, you will be asked to select a clock to follow (the client clock or the server clock). Choose the client clock by selecting it.



5. Step through the code by pressing F11.
6. If you open the **Clocks/Threads** window (press Alt + 5), you will see that the client clock advances more quickly than the server clock.
7. The program executes an infinite loop, and you must stop the debugger (press Shift + F5) to terminate the simulation.

9.4.2 Example 4: code details

The internal queue is implemented in a structure consisting of two counters (queueIn and queueOut) which are used to test how full the queue is, and an mpram containing the queued data. Use of an mpram allows the queue to be written to and read from in the same clock cycle.

```
typedef struct
{
    unsigned int queueIn;
    unsigned int queueOut;
    mpram
    {
        wom int DataWidth dataIn[MaxQueue];
        rom int DataWidth dataOut[MaxQueue];
    } values;

} Queue;
```

A `prialt` in a `do while` loop checks whether each client is ready to send data, and reads the data if it is ready. The use of `prialt` with a default case ensures that the server doesn't have to wait for each client to have data. The use of a loop ensures that each client is polled. If a single `prialt` statement were used with cases for each client, clients further down the `prialt` statement might never be polled, because higher-priority clients could always grab the resource.

```
while (1)
{
    i = 0;
    do
    {
        prialt
        {
            case clientReq[i] ? value:
                QueueInsert(reqQueue, value);
                break;
            default:
                break;
        }
        i++;
    } while (!IndexAtArrayEnd(i, MaxClients));
}
```

9.5 Example 5: microprocessor

The workspace for Handel-C tutorial example 5 is in ***InstallDir***`\DK\Examples\Handel-C\Example5`.

In this example, Handel-C implements a simple microprocessor. This microprocessor executes a program stored in ROM to calculate members of the Fibonacci number sequence.

It is equally possible to produce processors which contain specialized instructions for any application. Thus, you could use Handel-C to develop processors capable of executing programs for specialized applications with the minimum of effort.

9.5.1 Example 5: Microprocessor description

The system described in this example consists of a ROM containing the program to execute, a RAM containing some scratch variables and a processor that understands 10 opcodes. Each instruction is made up of a 4-bit opcode and a 4-bit operand. The _asm_ preprocessor macro is the assembler for this language and is used to fill in the entries in the program ROM declaration.

The processor has three registers:

- a program counter, pc, that points to the next instruction to be fetched from the ROM
- an instruction register, `ir`, containing the instruction being executed
- an accumulator register, x, used as one input to the 'ALU'

The instructions that the processor can execute are:

Opcode	Description
`HALT`	Stop processing
`LOAD`	Load a value from RAM into x
`LOADI`	Load a constant into x
`STORE`	Store x to RAM
`ADD`	Add a value from RAM to x
`SUB`	Subtract a value from RAM from x
`JUMP`	Unconditional jump to a ROM location
`JUMPNZ`	Jump to a ROM location if x is not 0
`INPUT`	Read a value into x
`OUTPUT`	Write x to user

Using these instructions, a ROM is built containing a program to generate the Fibonacci numbers.

The execution unit of the processor simply fetches instructions from the program ROM and executes them using a switch statement.

9.5.2 Compiling and simulating example 5

1. Double-click on the workspace file `Example5.hw` in the `Examples\Handel-C\Example5` directory.
2. Build the example in Debug mode by selecting **Build**>**Build Example5**.
3. Step into the program within the debugger by pressing F11.

9.6 Example 6: clock manager

The workspace for Handel-C tutorial example 6 is in ***InstallDir***`\DK\Examples\Handel-C\Example6`.

The program creates a clock manager for a Handel-C program that interfaces to an external EDIF program.

The program takes in a clock signal from an external program, and then selects whether to use the same clock or a clock divided by 10 for the part of the project to be written in Handel-C.

The program demonstrates how to instantiate primitives and parameterize them using the `properties` specification.

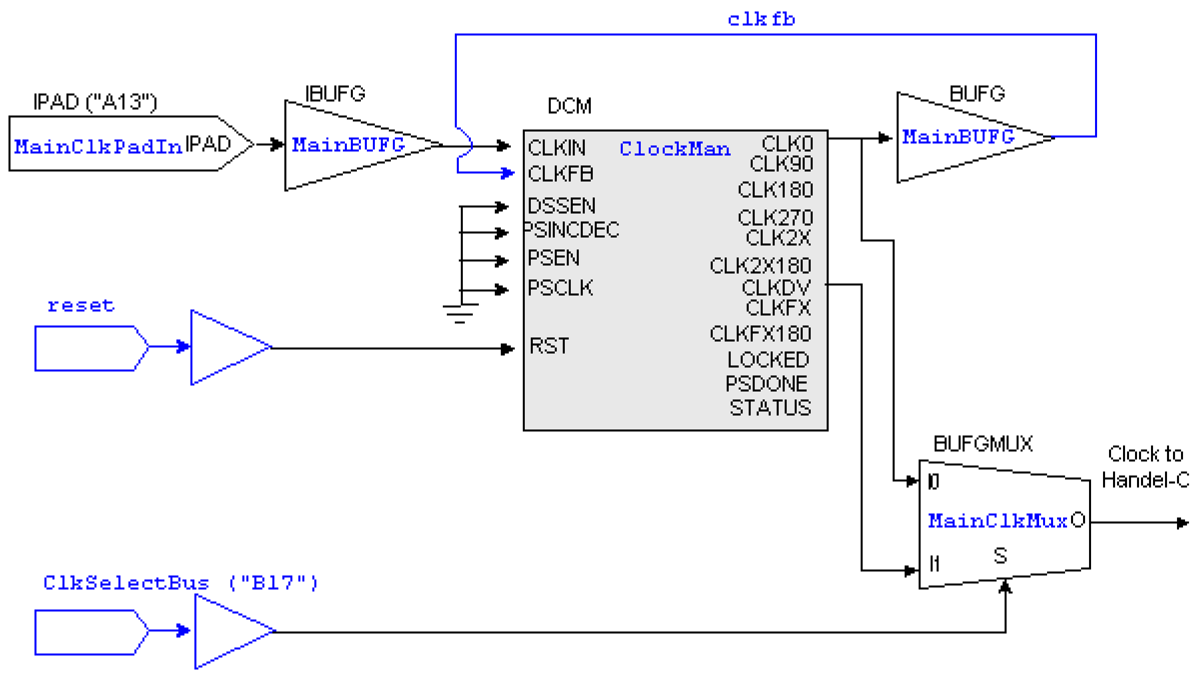
This example cannot be simulated as the clock is fed from an instantiated primitive and there is no clock statement in the Handel-C code. The example should be built for EDIF output. This option is not available in Nexus PDK.

9.6.1 Example 6: description of program

Example 6 demonstrates how to instantiate primitives, and how to parameterize primitives using the `properties` specification.

The Handel-C code connects to an external EDIF block which implements a clock manager. It allows you select between two clocks for your Handel-C program.

The primitives are instantiated using Handel-C `interface` definitions.



CIRCUIT PRODUCED BY EXAMPLE 6 CODE

The main clock signal `MainClkPadIn`, is fed into the Clock Manager, `ClockMan`.

The ClkSelectBus signal determines whether the clock used by the Handel-C program, MainClkMux, will be the same as the main clock (CLK0 signal), or divided by 10 (CLKDV signal).

Instantiating primitives

The primitives (IPAD, IBUFG, DCM, BUFG and BUFGMUX) are instantiated by the interface definitions in Handel-C.

For example,

```
interface IPAD(unsigned 1 IPAD) MainClkPadIn() with {properties = {{"LOC",
"A13"}}};
```

instantiates a primitive called IPAD, with a 1-bit output port called IPAD.

MainClkPadIn is the instance of the IPAD (there is only one instance of the primitive in this case). The properties specification sets the location property (LOC) and constrains it to pin A13.

If you build the example and examine the EDIF netlist, you can see the primitives represented by the blocks starting (cell...

For example:

```
(cell IPAD
   (cellType GENERIC)
   (view view_1
      (viewType NETLIST)
      (interface
         (port IPAD (direction OUTPUT))
   )
)
```

9.6.2 Compiling example 6 and examining the netlist

1. Double-click on the workspace file CustomClock.hw in the Examples\Handel-C\Example6 directory.
2. To examine the code, check that you are in **File View** in the Workspace window and click on the + sign to the left of the chip icon. Then double-click the file CustomClock.hcc to open it in code editor window.
3. Change the build configuration to EDIF: select **Build**>**Set Active Configuration**, then select EDIF and press **OK**.
4. Compile and build it by selecting **Build**>**Build CustomClock**.
5. Browse to the build folder: Example6\EDIF.
   You should see the following files: CustomClock.edf and CustomClock.hco.
6. To view the netlist, open CustomClock.edf in Notepad.

The primitives are visible in the blocks starting (Cell...

10. Porting C to Handel-C

10.1 Stages in porting C to Handel-C

There are a number of stages in porting and mapping a conventional C program to hardware. These are:

1. Decide how the software system maps onto the target hardware platform.
2. Convert the conventional C program to Handel-C and use the simulator to check correctness.
3. Modify code to take advantage of extra operators available in Handel-C.
4. Add fine grain parallelism.
5. Add the necessary hardware interfaces and map the simulator channels onto them.
6. Use the device place and route tools to generate the device image(s).

These steps are guidelines only. Some of the stages may not be relevant to your design or you may require extra stages if your design does not fit this example flow.

Decide how the software maps to the hardware

For example, external RAM connected to the device can be used to hold buffers used in the conventional C program. This mapping may also include partitioning the algorithm between multiple devices and, hence, splitting the conventional C into multiple Handel-C programs. Convert the conventional C program to Handel-C and use the simulator to check correctness. You can convert the program a piece at a time, leaving functions in C code and linking them into the Handel-C.

Convert the program from C to Handel-C

Remember that there may be optimizations that can be made to the algorithm given that a Handel-C program can use parallelism. For example, you can sort numbers more quickly in parallel by using a sorting network. This form of coarse grain parallelism can provide massive performance gains so time should be spent on this step.

Modify code to take advantage of extra operators available in Handel-C

For example, concatenation and bit selection can be used where conventional C programs may use shifts and masks. Simulate again to ensure program is still correct.

Add fine grain parallelism

For example, make parallel assignments or execute individual instructions in parallel to fine-tune performance. Again, simulate to ensure that the program still functions correctly.

Add the hardware interfaces

Add the hardware interfaces necessary for the target architecture and map the simulator channel communications onto these interfaces. If possible, simulate to ensure mapping has been performed correctly.

10.2 Porting C to Handel-C: Edge detector example

The edge detector example is provided in the `Examples\Handel-C\ExampleC` directory. It consists of a number of versions of the same application that detail the process of porting a conventional C application to a Handel-C application. All but the final stage (targeting real hardware) are presented as complete examples that may be simulated with the Handel-C simulator. They are stored as separate projects within a single workspace. You can execute this code, and simulate the different versions of the ported code.

The examples use specific hard-coded file names for the image data. The image data file names must be exactly the same as those given in the examples, or the source code must be edited and recompiled.

Edge detector example: program description

The edge detector program reads data from a raw data file into a buffer. The function `edge_detect` then performs a simple edge detection and stores the results in a second buffer which is stored in a second file.

The edge detection is performed by subtracting the pixel values for adjacent horizontal and vertical pixels, taking the absolute values and thresholding the result. The source and destination images are both 8-bit per pixel greyscale images.

Edge detector development stages

The edge detector example workspace is in ***InstallDir***`\DK\Examples\Handel-C\ExampleC\`.

The example is a series of programs showing how to port conventional C routines to Handel-C. Each of the programs is in a separate project within a single workspace.

10.2.1 Edge detector example: original program

The edge detector program gives a simple example of porting C code to Handel-C.

The original ANSI-C program is in ***InstallDir***`\DK\Examples\Handel-C\ExampleC\Edge_C\edge.c`.

To examine the file in DK:

1. Select **File**>**Open.**
2. Choose ANSI C/C++ in the **Files of type** box.
3. Browse to the location of the file.
4. Click **Open**.

Edge detector example: running the original C program

The conventional C source file and a compiled version are provided along with an example image (`source.bmp`).

You can run the program now to see the results. This is done by changing to the `Examples\Handel-C\ExampleC\Data` directory, opening a Command Prompt window and typing the following commands:

1. Convert the example BMP file to raw data with the `bmp2raw` utility.

```
bmp2raw -b source.bmp source.raw 8bppdest.rgb
```

2. Execute the conventional C edge detector.

```
..\Edge_c\edge_c
```

3. Convert the output from the edge detector back to a BMP file using the `raw2bmp` utility:

```
raw2bmp -b 256 dest.raw dest_c.bmp 8bppsrc.rgb
```

To compare results, you can use the standard Windows Paint utility to display the source and destination BMP files.

10.2.2 Stage 1: First pass conversion to Handel-C

The first step is to port the conventional C to Handel-C with as few changes as possible to ensure that the resulting program works correctly. The file handling sections of the original program are modified to read data from a file and write data back to a file using the Handel-C simulator.

The first pass conversion is in `edge_v1.hcc`. The following points should be noted about the port:

1. The `Source` and `Dest` buffers have been replaced with two RAMs.
2. An `abs()` macro expression has been used to replace the standard C function.
3. The `x` and `y` variables have been given widths equal to the number of address lines required for the RAMs to simplify the index of the RAM. Without this, each variable would have to be padded with zeros in its MSBs to avoid a width conflict when accessing the RAM.
4. Temporary variables have been used for the three pixels read from RAM to avoid the restriction on only one access per RAM per clock cycle. Without these variables, the condition for the `if` statement would require multiple accesses to the `Source` RAM.

5. The pixel values must be extended by one bit to ensure the subtract does not underflow.
6. The chanin (Input) and chanout (Output) simulator channels are used to transfer data in and out of the Handel-C simulator. They replace the fread and fwrite file operations in the original C source. chanin is used to read data from the source image file into the Handel-C simulator. chanout is used to write data from the Handel-C simulator to the destination image file. The file name is given using the with specification, e.g.
   `chanin unsigned Input with {infile = "..\Data\source.dat"};`

Running the first attempt Handel-C code

To execute the Handel-C code:

1. Convert the example BMP file to text data with the bmp2raw utility by opening a Command prompt or MS-DOS window, changing to the Examples\Handel-C\ExampleC\Data directory and typing:

   `bmp2raw source.bmp source.dat 8bppdest.rgb`

2. Open the DK edge detector workspace (Examples\Handel-C\ExampleC\ExampleC.hw) by double-clicking on it.
3. Build the first version of the Handel-C code in Debug mode by selecting **Build**>**Build_Edge_v1**. If Edge_v1 is not the active project, set this by selecting **Project**>**Set Active Project**>**Edge_v1**.
4. Run the project by selecting **Build**>**Start Debug**>**Go** or pressing F5
5. Convert the output from the edge detector back to a BMP file using the raw2bmp utility by opening a Command prompt or MSDOS window, changing to the Data directory and typing::

   `raw2bmp 256 dest.dat dest_v1.bmp 8bppsrc.rgb`

Data files are read from and written to the \Data directory, since this is set as the working directory on the Debugger tab in the Project Settings dialog.

10.2.3 Stage 2: First optimization of the Handel-C program

The second stage in developing the edge detector program is to change some of the operators familiar in C to operators more suitable for Handel-C.

In the stage 1 code, every time the Source or Dest RAM is accessed, a multiplication is made by the constant WIDTH. The Handel-C optimizer simplifies this to a shift left by 8 bits but we could easily do this by hand to reflect the hardware more accurately and reduce compilation times. We can also introduce new macros to access the RAMs given x and y coordinates:

```
macro expr ReadRAM(a, b) =
            ((unsigned 1)0) @
                Source[(0@a) + ((0@b) << 8)];
macro proc WriteRAM(a, b, c)
            Dest[(0@a) + ((0@b)<<8)] = c;
```

Notice how the macros pad both the result and the coordinate expressions with zeros. This allows us to reduce the width of the x and y counters to 8 bits each and reduces clutter in the rest of the program. This width reduction does mean that the loop conditions must be altered because x and y are no longer wide enough to hold the constant 256. Instead, we test against zero since the counters will wrap round to zero after 255.

Running the code version 2

To execute this version of the edge detector example Handel-C code:

1. If you have not done so, convert the example BMP file to text data.

   Open a Command prompt or MS-DOS window, change to the `ExampleC\Data` directory and type

   `bmp2raw source.bmp source.dat 8bppdest.rgb`
2. Open the DK edge detector workspace (`Examples\Handel-C\ExampleC\ExampleC.hw`) by double-clicking on it.
3. Make the version 2 project current within the ExampleC workspace by selecting **Project**>**Set Active Project**>**Edge_v2**:
4. Build and run the project by selecting **Build**>**Build Edge_v2** followed by F5.
5. Convert the output from the edge detector back to a BMP file using the `raw2bmp` utility by opening a Command Prompt or MS-DOS window. Change to the Data directory and type:

   `raw2bmp 256 dest.dat dest_v2.bmp 8bppsrc.rgb`

Data files are read from and written to the `\Data` directory, since this is set as the working directory on the Debugger tab in the Project Settings dialog.

10.2.4 Stage 3: Adding fine grain parallelism

To improve performance in the edge detector program we can make two modifications:

1. Replace `for` loops with `while` loops
2. Add multiple parallel accesses to external RAMs in single clock cycles

The version 3 edge detector project contains the Handel-C code with these modifications.

To execute this version of the code:

1. If you have not done so, convert the example BMP file to text data.

   Open a Command prompt or MS-DOS window, change to the `ExampleC\Data` directory and type

   `bmp2raw source.bmp source.dat 8bppdest.rgb`
2. Open the DK edge detector workspace (`Examples\Handel-C\ExampleC\ExampleC.hw`) by double-clicking on it.
3. Make the version 3 project current within the ExampleC workspace by selecting **Project>Set Active Project>Edge_v3**
4. Build and run the project by selecting **Build>Build Edge_v3** followed by F5.
5. Convert the output from the edge detector back to a BMP file using the `raw2bmp` utility by opening a Command Prompt or MS-DOS window. Change to the `Data` directory and type:

   `raw2bmp 256 dest.dat dest_v3.bmp 8bppsrc.rgb`

Data files are read from and written to the `\Data` directory, since this is set as the working directory on the Debugger tab in the Project Settings dialog.

Replacing for loops with while loops

The `for` loop expands into a `while` loop inside the compiler in the following way:

```
for (Init; Test; Inc)
    Body;
```

becomes:

```
{
    Init;
    while (Test)
    {
        Body;
        Inc;
    }
}
```

This is normally not efficient for hardware implementation because the *Inc* statement is executed sequentially after the loop body when in most cases it could be executed in parallel. The solution is to expand the `for` loops by hand and use the `par` statement to execute the increment in parallel with one of the statements in the loop body.

Multiple parallel access to external RAM

An area in which the edge detector program's performance can be improved concerns the three statements required to read the three pixels from external RAM. Without the restriction on multiple accesses to RAMs the loop body of the edge detector could be executed in a single cycle whereas the current program requires four cycles, three of which access the RAM. As many of these RAM accesses need to be eliminated as possible.

Since it is not possible to access the external RAM more than once in one clock cycle, the only way to improve this is to access multiple RAMs in parallel. Version 2 accesses most locations in the external RAM three times.

For example, when x is 34 and y is 56:

- The three pixels read are at coordinates (34,55), (33,56) and (34,56).
- The first pixel is also read when x is 34 and y is 55, and when x is 35 and y is 55.
- The second pixel is also read when x is 33 and y is 56, and when x is 33 and y is 57.

If the pixels are stored in two extra RAMs when they are read from the main external RAM for the first time then you could access these additional RAMs to get pixel values in the main loop.

The first step is to store the previous line of the image in an internal RAM on the device. This allows the pixel above the current location to be read at the same time as the external RAM is accessed. The second step is to store the pixel to the left of the current location in a register. The loop body then looks something like this:

```
Pixel1 = ReadRAM(x, y);
Pixel2 = PixelLeft;
Pixel3 = LineAbove[x];

LineAbove[x] = Pixel1;
PixelLeft = Pixel1;
```

At first glance, it looks worse, as the number of clock cycles has increased, but you can now add parallelism to make it look like this:

```
par
{
    Pixel1 = (int)ReadRAM(x, y);
    Pixel2 = PixelLeft;
    Pixel3 = (int)LineAbove[x];
}

par
{
    LineAbove[x] = Pixel1;
    PixelLeft = Pixel1;
}
```

Note the LineAbove RAM must be initialized at the start of the image to contain the first line of the image and the `PixelLeft` variable must be initialized at the start of each line with the left hand pixel on that line.

Since the second of these `par` statements and the `if` statement are not dependent on each other they can be executed in parallel.

Putting all these modifications together gives an `edge_detect` procedure. Examine edge_v3.hcc in DK or Notepad. Notice that the increment of `y` has been moved from the end of the loop to the start, and the start and end values have been adjusted accordingly. This allows the increment to be executed without additional clock cycles, which would be required if it were placed at the end of the loop.

10.2.5 Stage 4: Further fine grain parallelism

We have now reduced the core loop body from five clock cycles (including the loop increment) to 2 clock cycles. Can we do any better? The answer is yes because we should be able to access the two off-chip banks of RAM in parallel. Thus, the two parallel statements in the loop body could be executed simultaneously if we could organize the data flow correctly.

We have to modify the program again because the `LineAbove` internal RAM is accessed in both clock cycles. Parallelizing the two statements is not permitted because it would involve two accesses to the same internal RAM in a single clock cycle.

The solution is to increase the number of internal RAMs. The current line can be copied into one internal RAM while the previous line is read from a second internal RAM. The two internal RAM banks can then be swapped for the next line. Note that with Handel-C declaring two banks with 256 elements each (`ram unsigned char LineAbove[2][WIDTH]`) is much more efficient than 256 banks with two elements each, whereas in conventional C there would be no practical difference.

By also removing the `Pixel1`, `Pixel2` and `Pixel3` intermediate variables, the two statements in the loop body can be rolled into one. We use the LSB of the y coordinate to determine which line buffer to read from and which line buffer to write to. The external RAM read is done using a shared expression (`RAMPixel`) since we need the value from the RAM in multiple places but only want to perform the actual read once.

In the new version of the edge detector the core loop is now only one clock cycle long and is executed 255 times per line. One extra clock cycle is required per line for the initialization of variables and 255 lines are processed. In addition, 255 cycles are required to initialize the on-chip RAM and one extra clock cycle per frame is required for variable initialization. This gives a grand total of 65536 clock cycles per frame or an average of exactly one pixel per clock cycle. Since there is no way of getting the image into or the results out from the device any faster than this without changing the hardware interface, we conclude that we have reached the fastest possible solution to our problem.

Running the code version 4

To execute version 4 of the edge detector example Handel-C code:

1. If you have not done so, convert the example BMP file to text data.

   Open a Command prompt or MS-DOS window, change to the `ExampleC\Data` directory and type

   `bmp2raw source.bmp source.dat 8bppdest.rgb`
2. Open the DK edge detector workspace (`Examples\Handel-C\ExampleC\ExampleC.hw`) by double-clicking on it.
3. Make the version 4 project current within the ExampleC workspace by selecting **Project**>**Set Active Project**>**Edge_v4**.
4. Build and run the project by selecting **Build**>**Build Edge_v4** followed by F5.
5. Convert the output from the edge detector back to a BMP file using the `raw2bmp` utility by opening a Command Prompt or MS-DOS window. Change to the `Data` directory and type:

   `raw2bmp 256 dest.dat dest_v4.bmp 8bppsrc.rgb`

Data files are read from and written to the `\Data` directory, since this is set as the working directory on the Debugger tab in the Project Settings dialog.

10.2.6 Stage 5: Adding hardware interfaces

Once the edge detector program has been simulated correctly you must add the necessary hardware interfaces.

1. Add read and write procedures.
2. Declare external pins and synchronize the frame grabber.
3. Change the project settings for EDIF, Verilog or VHDL.

Edge detector example: adding macro procedures

There must be two new macro procedures - one to read a word from the host and one to write a word to the host. These could also be implemented as functions.

The suitably modified code looks like this:

```
// Read word from host
macro proc ReadWord(Reg)
{
    while (ReadReady == 0)
        delay;
    Read = 1;       // Set the read strobe
    par
    {
        Reg = dataB.in; // Read the bus
        Read = 0;   // Clear the read strobe
    }
}

// Write one word back to host
macro proc WriteWord(Expr)
{
    par
    {
        while (WriteReady == 0)
            delay;
        dataBOut = Expr;
    }
    par
    {
        En = 1;     // Drive the bus
        Write = 1; // Set the write strobe
    }
    Write = 0;      // Clear the write strobe
    En = 0;         // Stop driving the bus
}
```

Edge detector example: pins and frame grabber

We need to define the pins for the external RAMs and remove the RAM declarations we added to simulate the RAMs.

The main program also needs to be modified to include the code to synchronize the frame grabber with the edge detector.

10. Porting C to Handel-C

10.2.7 Edge detector example: compiling for hardware

To compile the edge detector program for EDIF, VHDL or Verilog output, you need to change the build configuration settings in the DK GUI (**Build**>**Set Active Configuration**). You cannot compile for hardware if you are using Nexus PDK.

The code is not designed for a specific device. You would need to know the appropriate pins for the device you are targeting. The pin definitions given are examples only and do not reflect the actual pins available on any particular device.

The code excluding the edge detection and host interface macros is given below.

```
#define LOG2_WIDTH 8
#define WIDTH 256
#define LOG2_HEIGHT 8
#define HEIGHT 256

set clock = external "P1";
unsigned 8 Threshold;

// External RAM definitions/declarations
ram unsigned 8 Source[65536] with {
      offchip = 1,
      data = {"P1", "P2", "P3", "P4","P5", "P6", "P7", "P8"},
      addr = {"P9", "P10", "P11", "P12",
              "P13", "P14", "P15", "P16",
              "P17", "P18", "P19", "P20",
              "P21", "P22", "P23", "P24"},
      we = {"P25"}, oe = {"P26"}, cs = {"P27"}};
ram unsigned 8 Dest[65536] with {
      offchip = 1,
      data = {"P28", "P29", "P30", "P31",
                    "P32", "P33", "P34", "P35"},
      addr = {"P36", "P37", "P38", "P39",
              "P40", "P41", "P41", "P43",
              "P44", "P45", "P46", "P47",
              "P48", "P49", "P50", "P51"},
      we = {"P52"}, oe = {"P53"}, cs = {"54"}};
macro expr ReadRAM(a, b) =
            ((unsigned 1)0) @ Source[(0@a) + ((0@b) << 8)];
macro proc WriteRAM(a, b, c) Dest[(0@a) + ((0@b)<<8)] = c;
```

```
#ifndef SIMULATE
// Host bus definitions/declarations
unsigned 8 dataBOut;

int 1 En = 0;
interface bus_ts_clock_in(int 4) dataB(dataBOut, En==1) with
              {data = {"P55", "P56", "P57", "P58"}};

int 1 Write = 0;
interface bus_out() writeB(Write) with
              {data = {"P59"}};

int 1 Read = 0;
interface bus_out() readB(Read) with
              {data = {"P60"}};

interface bus_clock_in(int 1) WriteReady() with
              {data = {"P61"}};

interface bus_clock_in(int 1) ReadReady() with
              {data = {"P62"}};
#endif
```

Insert `edge_detect`, `ReadWord` ***and*** `WriteWord` ***function and macro definitions here***

```
void main(void)
{
    ReadWord(Threshold);

    while(1)
    {
        unsigned Dummy;

        ReadWord(Dummy);
        edge_detect();
        WriteWord(Dummy);
    }
}
```

11. Integrating C/C++ files in your project

You can integrate C or C++ files in a Handel-C project built for Debug or Release:

1. Add the C/C++ files to your project (use **Project**>**Add**>**Files**).
2. Specify the file language
   If you are adding a file with a non-standard extension, you may need to right-click the file to specify its type in File properties.
3. Edit your Handel-C files to call the C/C++ functions if required.
4. Set up custom build steps to compile the C/C++ files.
5. Link the C/C++ files and libraries into your Handel-C project.
6. Build and simulate your project. You can only use the new (fast) simulator.

You cannot debug C/C++ code in DK, or set breakpoint in it. If you want to build an .exe file instead of a simulator `.dll` file, change the **Simulator compilation command line** to specify this.

If you step into a C/C++ simulation in DK and use the **Break** or **Stop Debugging** commands, a dialog will appear after a few seconds saying: '**Simulator is not responding. Terminate simulation process?**'. Select **Yes**.

11.1 Calling C/C++ functions from Handel-C

You can call C/C++ library functions from Handel-C code in code built for Debug or Release:

- by using the extern "***language***" construct to declare an individual function
- by #including a specific C/C++ header file.

For example:

```
extern "C"
{
   int printf(const char *f, ...);
}
```

OR

```
extern "C"
{
//strip Microsoft specific extensions from the header file
   #define __cdecl

   #include "j:\vc98\include\stdio.h"
}
```

11. Integrating C/C++ files in your project

You can only link to C/C++ code if you are building for Debug or Release.

If you call a function that requires a user action before the program continues, your DK simulation may appear to hang. For example, if you make a call to `getchar()`, you need to press Enter in the DOS program before it will continue executing. Once you have done this, you can continue using DK GUI commands.

11.2 Compiling and linking in a C/C++ file

If you are integrating a C or C++ file into a Handel-C project, you need to specify custom build commands and link in the file and any libraries it uses.

Specifying the custom build commands

1. Select the file in the Workspace window and then select **Project**>**Settings**>**Build commands**.
2. Set the **Description** to display appropriate text (e.g., Compiling C++ file...)
3. Set the **Commands** to compile the file. Use quotes around strings if they have spaces in them.
4. Set the **Outputs** to be the output file name (e.g. `MyProject.obj`).
5. Specify any files that need to be built before the current file in the **Dependencies**.

Linking the C/C++file and library

1. Select the project containing the file.
2. Select the **Linker** tab on the Project Settings dialog. Add the output file name (e.g. `MyProject.obj`) to the **Additional C/C++ Modules** box.
3. Add the names and paths of any library files used by the C/C++ file to the **Additional C/C++ Modules** box. Separate entries by commas.
4. Save the Project settings by pressing the **OK** button

11.2.1 Build commands to compile C/C++ files

If you are using C or C++ files in a DK project, you need to specify custom build commands to compile them for simulation. Custom build commands are specified on the **Build commands** tab in Project Settings.

Example commands for building a C/C++ object file to be linked into a simulation .dll

Visual C++ example: `cl -c "$(InputPath)" -Fo MyProject.obj`

Borland example: `bcc32 -O2 -c "$(InputPath)" -o MyProject.obj`

GCC example: `g++ -O2 -c "$(InputPath)" -o MyProject.obj` (for a C++ file; if you were building a C file the command would be `gcc -O2 -c $(InputPath)" -o MyProject.obj`).

You would then specify `MyProject.obj` as the output file name.

File path strings need to have quotes around them if they contain spaces. The file and directory macros must also be quoted if the string they represent contains spaces.

Using the Wide Number library

If you are using the Wide Number library, you need to have `DK\Sim\Include` on the Include path, using the `-I` command. For example:

```
cl -c -I"C:\Program Files\Celoxica\DK\Sim\Include" Fred.cpp -Fo Fred.obj
```

11.3 Calling Handel-C functions from C/C++

You can call Handel-C functions from C or C++ code using the `extern "`***language***`"` construct. To do so:

- the C/C++ code must reside in a different file to the Handel-C code.
- the widths of parameters must match. If necessary, use the wide number library to provide type definitions for wide Handel-C variables.
- you must specify a clock in any Handel-C source files containing functions that are called by the C/C++ code.

1. Build the Handel-C file as an .obj file in DK.
2. Run a build command in your C/C++ compiler to link in the .obj file when you compile your C/C++ project

C/C++ compiler build commands

- Visual C++: `cl -O2 -I"`***InstallDir***`\DK\Sim\Include"` ***C++FileName***`.cpp` ***HandelCFileName***`.obj`
- Borland: `bcc32 -O2 -I"`***InstallDir***`\DK\Sim\Include"` ***C++FileName***`.cpp` ***HandelCFileName***`.obj -o`***C++FileName***`.exe`
- GCC: `g++ -w -O2 -I"`***InstallDir***`\DK\Sim\Include" C++FileName.cpp` ***HandelCFileName***`.obj -o`***C++FileName***`.exe`

These commands are for C++ code, for C code use ***CFileName***`.c` instead of ***C++FileName***`.cpp`, and for GCC, use `gcc` instead of `g++`.

11. Integrating C/C++ files in your project

11.3.1 Calling Handel-C functions from C/C++: example

This example shows how to use the extern construct to use a Handel-C function in your C++ code.

Handel-C:

```
extern "C++" short wideSum(char a, char b)
{
   int 16 result;
   result = (int 16)(0 @ a) + (int 16)(0 @ b);
   return(result);
}
```

C++:

```
extern short wideSum(char a, char b);

void main (void)
{
     char x = 10,y = 5;
     short result;
     result = wideSum(x,y);
}
```

11.3.2 Calling Handel-C functions from C++: tutorial

This example demonstrates how to use Handel-C functions in your C++ code. The example files are in ***InstallDir***`\DK\Examples\extern_C\Handel-C in C++`.

The example code creates an HDLC protocol. The HDLC (High level Data Link Control) protocol is a general-purpose protocol that operates at the data link layer (layer 2) of the OSI reference model. Data is packaged into frames which end with a 16-bit Cyclic Redundancy Check (CRC) value.

The HDLC code is written in C++ (`hdlc.cpp`). This code calls a CRC function written in Handel-C (`crc.hcc`). `HDLCTest.txt` contains data to test the HDLC model.

Running the example

1. Open `CRC.hw` in DK
2. Select **Project**>**Settings**, and open the **Linker** tab.
3. Change the default **Simulator compilation command line** to compile an `.obj` file.
4. Check that the project is in Debug mode and then build it (**Build**>**Build crc**). This should create a file called `crc.obj` in the Project directory.

11. Integrating C/C++ files in your project

5. Open a command prompt and browse to the directory containing the example files. Use one of the following commands, depending on which C++ compiler you are using:
   Visual C++: `cl -O2 -I"`***`InstallDir`***`\DK\Sim\Include" hdlc.cpp crc.obj`
   Borland: `bcc32 -O2 -I"`***`InstallDir`***`\DK\Sim\Include" hdlc.cpp crc.obj -ohdlc.exe`
   GCC: `g++ -w -O2 -I"`***`InstallDir`***`\DK\Sim\Include" hdlc.cpp crc.obj -ohdlc.exe`
   This should create a file called `hdlc.exe` in the Project directory.
6. Double-click the icon of the `.exe` file to run it. It should use the data in the test file, `HDLCTest.txt`, and display notification of the data transmitted:

   ```
   Data received: 0x03

   Data received: 0x07

   ...

   Data received: 0x0e
   ```

11.4 Using extern C: bitonic sort example

This program runs a bitonic sort algorithm (a sort algorithm designed for parallel processing).

It consists of two files:

- `ctestbench.c` (ANSI-C file): contains functions for filling a buffer with data and for checking that data is sorted.
- `mainhc.hcc` (Handel-C file): declares the functions in `ctestbench.c` using the `extern "C"` construct, and the standard C library function `printf`. This file contains the sorting algorithm. It calls the C function to load the data, applies the bitonic sort algorithm (using the `printf` function to display debug information) and then calls the C function to check the data.

11.4.1 Compiling and simulating the bitonic sort example

1. Open the workspace file (***`InstallDir`***`\DK\Examples\extern_C\bitonic_sort\CTestBench.hw`) by double-clicking on it. DK starts with the CTestBench workspace open.
2. Check that you are in **File View** in the Workspace window and click on the + sign to the left of the chip icon to see what files are within the project.
3. To examine the code, double-click `ctestbench.c` or `mainhc.hcc`.

4. If you are using Borland or GCC (GNU) as your backend compiler, you will need to alter the custom build commands for `ctestbench.c`:
   Open Project Settings (**Project**>**Settings**)
   Click on the + next to the CTestBench project in the left hand pane to display the project files, and select `ctestbench.c`.
   Select the **Build commands** tab.
   Change the line shown in the **Commands** window:
   Borland: `bcc32 -n"$(TargetDir)" -c "$(InputPath)" -o"CTestBench.obj"`
   GCC: `gcc -c "$(InputPath)" -o "$(TargetDir)\ctestbench.obj"`
   Select **Outputs** in the View box. Ensure that the **Outputs** box shows the correct location of the output file (e.g. `$(TargetDir)\CTestBench.obj`). Do not put the output file location in quotes.
   The output file and location must also be specified in the **Additional C\C++ Modules** field on the **Linker** tab in Project Settings.
5. Build the example in Debug mode by selecting **Build CTestBench** from the **Build** menu, or pressing F7.
6. Start the debugger by pressing F5. Alternatively, press F11 to step through the simulation, and advance to the end (Ctrl+F11).

   If you run to the end you should see a command window with the following messages:

   ```
   Getting data from external C routine...

   Sorting data...

   Checking data...

   Data correct!
   ```
7. Stop the debugger by pressing Shift F5.

11.5 Porting C++ to Handel-C: HDLC example

The High-level Data Link Control (HDLC) protocol is a general-purpose protocol that operates at the data link layer (layer 2) of the OSI reference model. Data is packaged into frames which end with a 16-bit Cyclic Redundancy Check (CRC) value.

The program consist of two files:

- `hdlc.cpp` (C++file)
- `CRC.hch` (Handel-C file)

The program takes data as a bitstream from an input file, `HDLCTest.txt`, packs it into frames, and performs error checking using a CRC function. When you simulate the program, the results are displayed in a command window.

It demonstrates a stage in porting a HDLC program from software to hardware, where the `main` function and the function that calculates the CRC value is moved from C++ to Handel-C.

11. Integrating C/C++ files in your project

You can compile and simulate the program entirely in C++ (using a C++ compiler), or link to some of the C++ functions in Handel-C, and build and simulate the program using DK.

11.5.1 Description of the HDLC example

To examine the code:

1. Open the workspace file (`DK\Examples\extern_C\HDLC\HDLC.hw`) by double-clicking on it. DK starts with the HDLC workspace open.
2. Check that you are in file view and click on the + sign to the left of the chip icon to see what files are within the project.
3. To examine the code, double-click `hdlc.cpp` or `CRC.hcc`.

The program consists of four functions

- `main`: calls the receiver function
- `GetBit` opens a file and reads a bit from it
- `CRCGen`: generates the CRC value
- Receiver calls GetBit, packs the bits into a frame, and calls the CRC checking function

Both files contain two `#define` statements at the start. One of these will be commented out, for example:

```
//#define SOFTWARE
#define HARDWARE
```

If the `#define SOFTWARE` remains (`#define HARDWARE` is commented out), the code in `CRC.hcc` will not be built due to the `#ifdef HARDWARE` statements and the `CRCGen` function and the `main` function in `hdlc.cpp` will be used.

If the `#define HARDWARE` remains (`#define SOFTWARE` is commented out), the `main` function runs in Handel-C and the program can be simulated using DK. The `CRCGen` function is also shifted to Handel-C.

The Handel-C file declares the external C++ function `receiver`, and makes the `CRCGen` function available to C linkage using the `extern "C++"` construct. The C++ file declares the extern function `CRCGen`.

The program runs `main` in Handel-C, calls `receiver` in C++, which then calls `CRCGen` in Handel-C.

In order to use the DK debugger, `main()` must be in the Handel-C program.

11.5.2 Compiling and simulating the HDLC example

Compiling and simulating for software (entirely in C++)

To build the program for software only:

1. Open `hdlc.cpp`. The file is located in ***InstallDir***`\DK\Examples\extern_C\HDLC`.
2. Comment out the `#define HARDWARE` statement at the top of the file (and make sure that `#define SOFTWARE` is not commented out).
3. In your C++ compiler, set paths to the DK simulation library and include files:***InstallDir***`\DK\Sim\Lib` and ***InstallDir***`\DK\Sim\Include`.
4. Build and simulate the file using your C++ compiler.

Porting to hardware (split between C++ and Handel-C)

1. Open the HDLC workspace file in DK (***InstallDir***`\DK\Examples\extern_C\Hdlc\HDLC.hw`) by double-clicking on it.
2. Open `crc.hcc` in the code editor window by double-clicking one it.
3. Comment out the `#define SOFTWARE` statement at the top of the file (and make sure that `#define HARDWARE` is not commented out).
4. Open `hdlc.cpp` and comment out the `#define SOFTWARE` statement (and make sure `#define HARDWARE` is not commented out).
5. Check the custom build commands for `hdlc.cpp`:
   Open Project Settings (**Project**>**Settings**)
   Click on the + next to the HDLC project in the left hand pane to display the project files, and select `hdlc.cpp`.
   Select the **Build commands** tab.
   Edit the path shown to the `DK\Sim\Include` directory in the **Commands** window, if necessary.
   If you use Borland or GCC (GNU) as your backend compiler, change the line shown in the **Commands** window:
   Borland: `bcc32 -n$(TargetDir) -I "..\..\..\Sim\Include" -c hdlc.cpp -o"hdlc.obj"`
   GCC: `g++ -I "..\..\..\Sim\Include" -c hdlc.cpp -o"$(TargetDir)\hdlc.obj"`
   Select **Outputs** in the View box. Ensure that the **Outputs** box shows the correct location of the output file (e.g. `$(TargetDir)\hdlc.obj`). Do not put the output file location in quotes.
   The output file and location must also be specified in the **Additional C\C++ Modules** field on the **Linker** tab in Project Settings.
6. Build the project in Debug mode, by selecting **Build Hdlc** from the Build menu, or pressing F7.
7. Start the debugger by pressing F5. Alternatively, press F11 to step through the simulation, and advance to the end (Ctrl+F11). To end the simulation, press the Stop Debugging button or select **Debug**>**Stop Debugging**. After a few seconds a dialog will appear asking if you want to close the simulation.

11. Integrating C/C++ files in your project

Results

You should get the same results for both versions of the program. A command window will display notification of data transmitted:

```
Data received: 0x03
Data received: 0x07
...
Data received: 0x0e
```

If you have built the program for software (using the C++ compiler), the command window will be produced by `hdlc.exe`. If you have built the program for hardware (using the DK Handel-C compiler) the command window will be produced by `hdlc.dll` inside the DK debugger.

12. Targeting hardware

12.1 Targeting a particular synthesis tool

You need to specify an output style for VHDL or Verilog output. This enables the compiler to generate code that uses the features of the selected synthesis/simulation tool.

1. In the **Project Settings** dialog, ensure that the mode is **VHDL** or **Verilog**.
2. Select the **Linker** tab.
3. In the **Synthesis tool** pull-down list, select the appropriate tool:
   Generic
   Mentor Graphics LeonardoSpectrum
   Mentor Graphics Precision
   Model Technology ModelSim (used for simulation)
   Synplicity Synplify

Choose Generic if you want to use a synthesis tool which is not listed. Choose ModelSim if you want to simulate your code.

If you are using the command line compiler, use the `-syn` ***`SynthesisTool`*** option.

If you are generating VHDL or Verilog code for simulation with ModelSim, you can only use multi-port memories if the ports have the same width and the same depth.

12.2 Technology mapping

The DK Technology Mapper performs technology mapping of general logic into device-specific logic blocks.

Technology mapping is available for EDIF output for the following devices:

- Actel: ProASIC and ProASIC+
- Altera: Apex 20K, 20KE and 20KC, Apex II, Excalibur, Cyclone, Stratix, Stratix GX
- Xilinx: Virtex, VirtexE, Virtex-II, Virtex-II Pro, Spartan-II, Spartan-IIE, Spartan-3

Creating technology mapped EDIF

To create mapped-EDIF:

- Tick the **Use technology mapper** option on the **Linker tab** in Project Settings.

OR

- Use the `-lutpack` option in the command-line compiler

If you have created a project for an Actel device by selecting **File**>**New**, and then basing your project on one of the Actel chips listed, technology mapping is on by default. In all other circumstances, it is off by default.

The Handel-C compiler can generate an estimate of the number of look-up tables and other resources that will be used by the mapped-EDIF, using the logic estimator.

12.3 Optimizing arithmetic hardware in Actel devices

If you are targeting Actel ProASIC or ProASIC+ devices, you can optimize arithmetic hardware for area or for speed in EDIF output from your Handel-C code.

In the DK GUI, open the **Compiler** tab in Project Settings (you need to select a file in the left pane to see this tab). In the **Expand netlist for:** box, choose **Area** (to minimize area) or **Speed** (to maximize speed). The default setting is **Speed**.

If you are using the command-line compiler, use the `-N` option:

-N+area	minimizes area
-N+speed	maximizes speed

The area and speed settings affect adders, subtractors, multipliers and dividers in Actel devices. They have no effect for Altera and Xilinx devices.

12.4 Targeting hardware via VHDL

To target hardware via VHDL, set the **Build**>**Set Active Configuration** option to VHDL. This compiles directly to a `.vhd` file which can be passed to your synthesis or simulation tools. You cannot compile Handel-C to VHDL from Nexus PDK.

You must specify the synthesis tool that you are using the **Linker** tab in **Project Settings**. If you wish to simulate your VHDL, select ModelSim as the tool used.

The code generated is structured and relates the Handel-C function names to the VHDL entity names.

If you want to simulate VHDL produced from Handel-C code or you want to target Xilinx devices, you need to link to a ROC file.

In previous versions of DK, you could generate less readable VHDL by un-checking the **Generate Debug information** box on the **Compiler** tab of **Project Settings** or using the command line `-g` option. This option is no longer available. All VHDL output now has names generated from Handel-C variable names.

If you want to co-simulate Handel-C with VHDL code you can use the Co-simulation Bridge for ModelSim provided in Celoxica's Platform Developer's Kit.

12.4.1 VHDL file structure

Each Handel-C source file is mapped to a VHDL file. Each Handel-C function is mapped to an entity and architecture. There is also a top-level VHDL file which links the design entities together and contains global design ports. Macros are converted to inlined VHDL. Source files consisting only of macro expressions or macro procedures will be converted to an empty file and then deleted.

In previous versions of DK you could compile your VHDL without debug information to produce less readable output. This option is no longer available; all VHDL output now has names generated from Handel-C variable names. The -g option now has no effect for VHDL output.

File names

VHDL file names depend on whether you build your files using the DK GUI, or from the command line compiler.

If you use the command line compiler, you specify the name of your top-level output file using the -o option. For example:

```
handelc -vhdl -o OutputFile
```

will produce a top level file called `OutputFile`.vhd

If you are using the GUI, the top-level file is called ***ProjectName***_top.vhd.

Other files are named after the Handel-C files: ***FileName***.***suffix*** source files became ***FileName_suffix***.vhd. For example, `UsefulFile.hcc` becomes `UsefulFile_hcc.vhd`.

If there is more than one source file with the same name, further files are called ***FileName_suffix_N***.vhd, where ***N*** increments from 1.

Entities

The top-level VHDL file contains an entity with the same name as this file (without the .vhd extension).

Each VHDL file corresponding to a Handel-C source file starts with an entity containing global logic defined within that file called ***FileName_suffix***. For example, the global variables in `UsefulFile.hcc` are stored in the entity `UsefulFile_hcc`.

Functions are mapped onto entities called ***FileName_suffix_FunctionName***. For example, a function called `MyFunction` defined in `MySource.hcc` becomes an entity called `MySource_hcc_MyFunction`.

Shared functions have a set of inputs for each use of the function.

`inline` functions have separate entities for each use of the function. The first instance of the function generates an entity as above. Later instances have numbers appended to the name, starting at 1. For example, the fourth instance of the inline function `FastFunction` in `UsefulFile.hcc` becomes `UsefulFile_hcc_FastFunction_3`.

Global reset

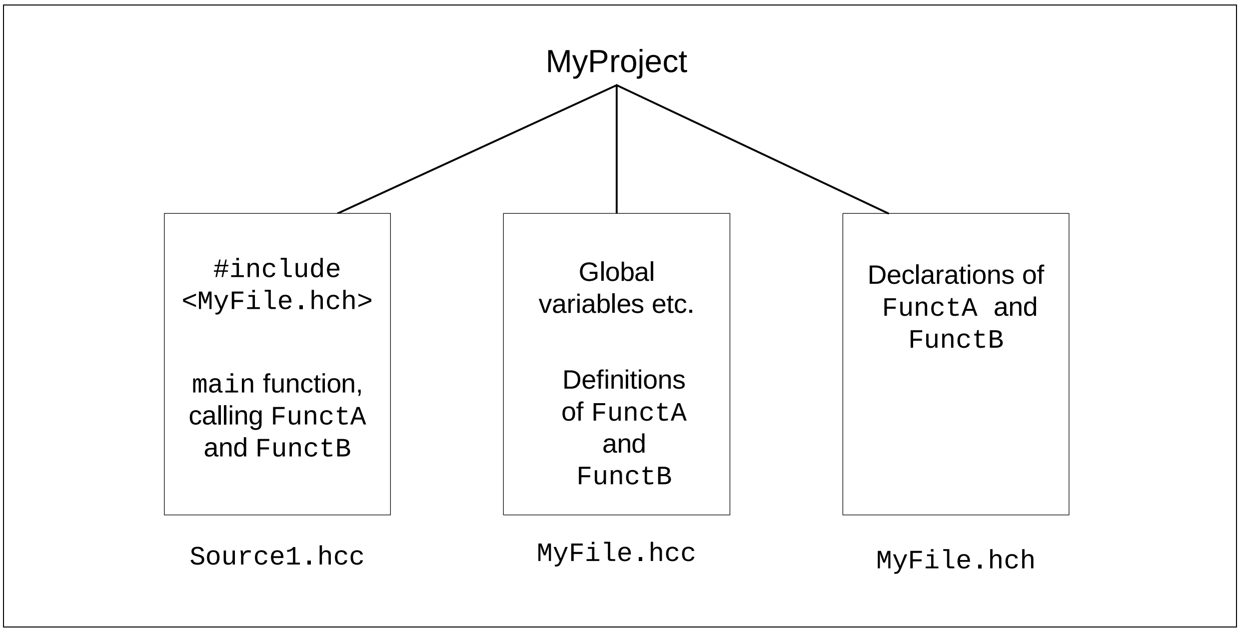
A global reset line is connected to all flip-flops/registers in the design. You can specify a reset pin using the set reset construct.

You must specify a reset pin for Actel devices. If no reset is specified for Altera or Xilinx devices, the registers in the design are reset on configuration. Altera devices have the reset wire connected to ground, Xilinx devices use a ROC block.

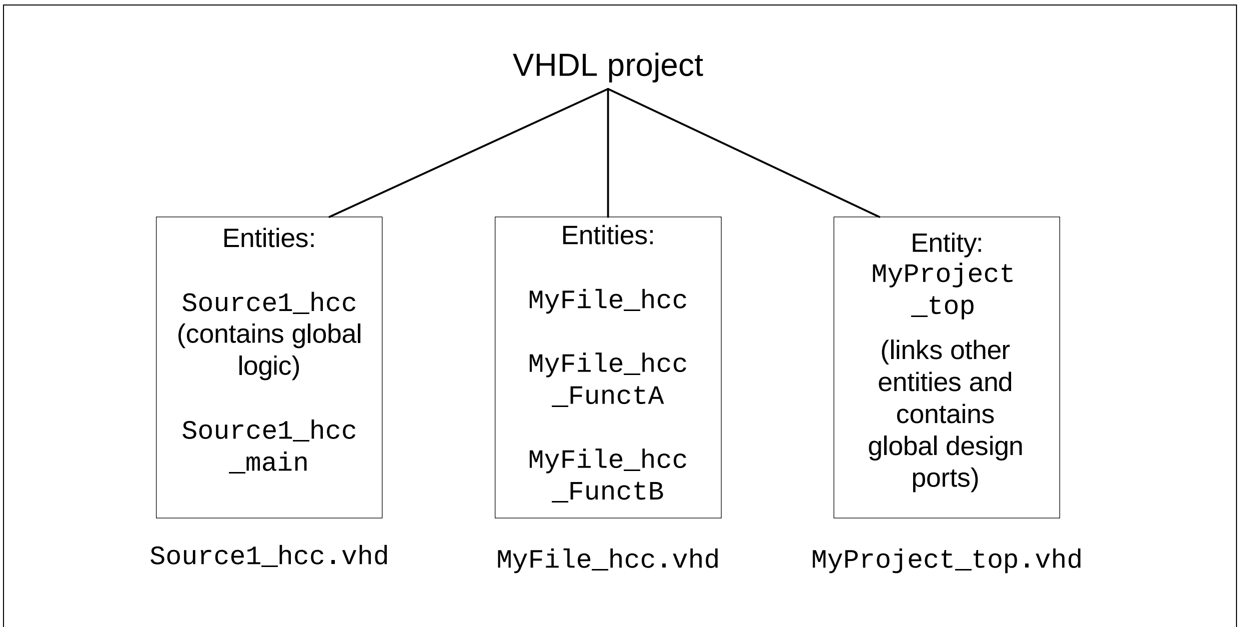
If you specify a reset using the data specification, a reset pin (called ***Data***) is added to the top-level entity of the design. If you specify a reset without a pin, the reset pin will be called ResetPin.

12.4.2 Naming of VHDL files and entities

If you compile the Handel-C project shown below to VHDL using the GUI:



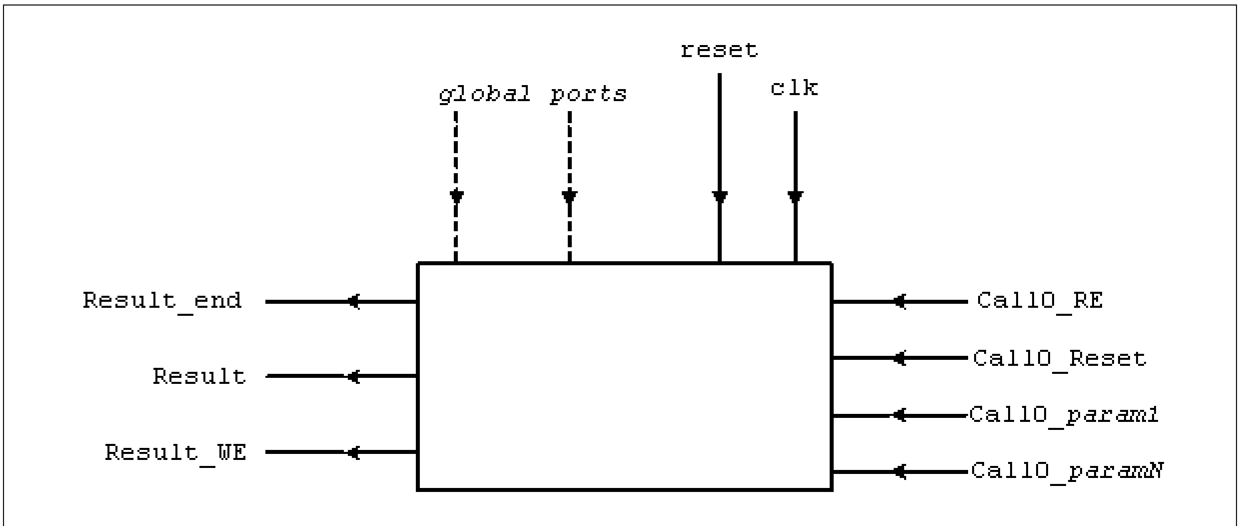
the following VHDL files and entities are produced:



If you compile the files from the command line, the top-level file and its entity are named after the output file name you specify using the -o option. The other file and entity names are the same as those shown above.

12.4.3 Mapping Handel-C functions to VHDL entities

An entity generated from a Handel-C function will have the following inputs and outputs:



There is an input port for each parameter to the function. It is given the name `Call`***`N_parameterName`*** and is of the width of that parameter. For example, the function `add(int 8 a, unsigned 16 b)` in the file `maths.hcc` would be converted to an entity `maths_hcc_add`. The first use of the function would generate an 8-bit port called `Call0_a` and a 16-bit port called `Call0_b`.

Each call to a shared function will duplicate the numbered ports (e.g. `Call0_RE`) with an incremented number. The result lines are the same for each call to the function.

When the function is called, the `Call`***N***`_RE` port is set high. One clock cycle before the function has completed, the `Result_WE` port is set high. When the function completes, the result appears on the `Result` port, and the `Result_End` line is set high.

The `Call`***N***`_Reset` port is used by the `try... reset` construct. If this construct was not used, the port will be connected to ground.

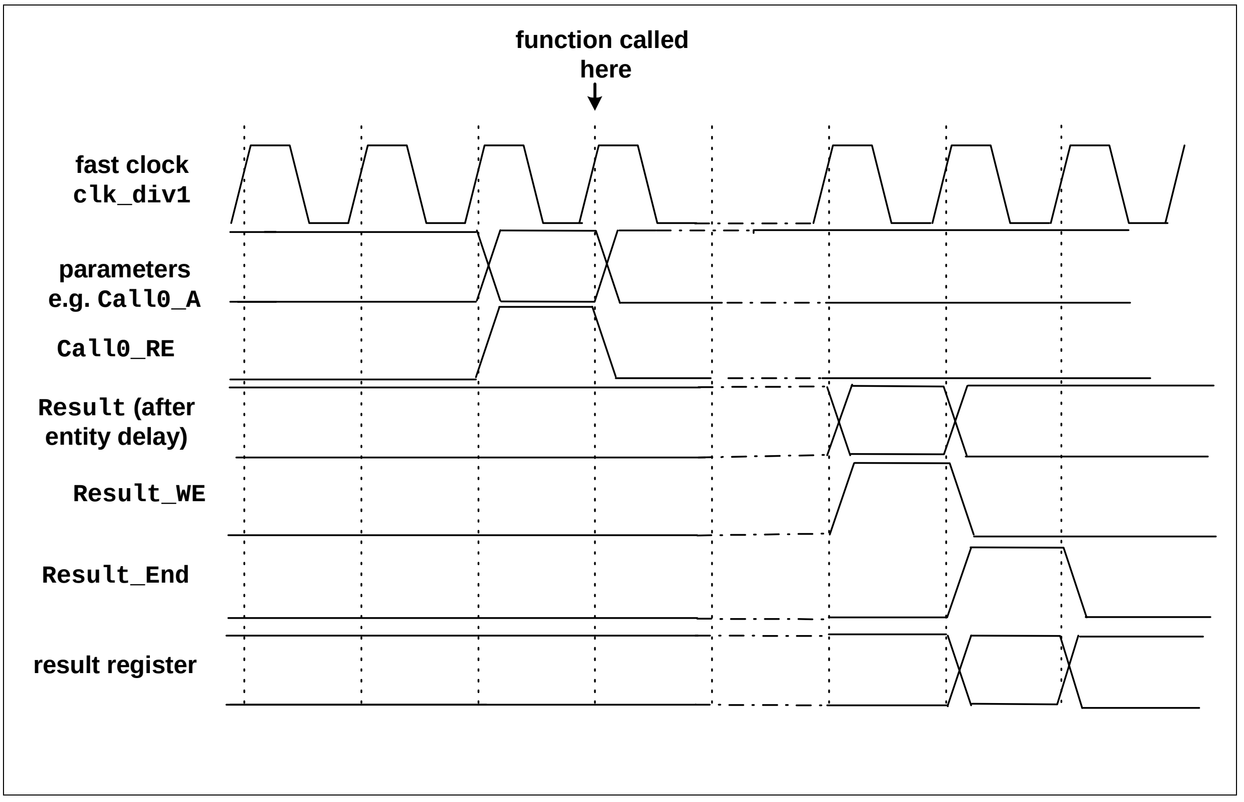
Global ports are produced from signals that cross function boundaries, such as global variables, ground and power. The names of global ports are prefixed with `globals_`.

If you have used the `set reset` construct in your code, the `reset` port will be replaced by a global port.

The name of the clock port depends on whether you specify a clock divide. If there is no clock divide, or the clock is divided by 1, the port will be called `clk`. If the clock divide is more than 1, the name of the port will be `clk_div`***N***, where ***N*** is the value of the divide.

Timing

The timing for the entity signals is shown below.



When the `Result_WE` signal is asserted, the result can be written to the result register on the next rising clock edge.

The `Result_End` signal is asserted in the clock cycle before the entity logic is released.

12.5 Targeting hardware via EDIF

To target hardware via EDIF, you set up your project to target EDIF using the **Build**>**Set Active Configuration** command. This compiles directly to an .edf file which can be passed to your place and route tools. You cannot compile Handel-C to EDIF from Nexus PDK.

12.5.1 EDIF block and net names

Net and block names take this format:

{B | W} [G][T]/***d_filename_lineNumber*** [_***functionName***] [_(***blockType***] [_***pinLoc***] [_***extraInfo***]

where:

B	Indicates that the current name is for a block (as opposed to a net).
W	Indicates that the current name is for a wire/net (as opposed to a block).
G	Optional. Indicates that a net is global and crosses file or function boundaries.
T	Optional. Indicates that the block or net is at the top level of the design.
Id	The unique Id for the block/net, within its name scope.
filename	The name of the file containing the source code from which the block/net was generated. It forms part of the name scope for the block/net.
lineNumber	The number of the line of source code from which the block or net was generated.
functionName	The name of the function containing the source code from which the block/net was generated. It forms part of the name scope for the block/net. This may be missing as blocks and nets can result from code not belonging to any functions.
blockType	Only applies to blocks. A string identifying the type of block in question (i.e. whether it is a register, an AND gate, a pad, etc).
pinLoc	Optional (it only applies to blocks or ports which are pad connections). Identifies the pin number assigned to the pad.
extraInfo	Optional. Contains additional useful debug information. For instance it might specify the variable name from which a register has resulted, whether a particular input pin is a clock connection, whether a wire is a clock net etc.

Note that multiple name scopes can only be guaranteed if the design has been compiled from scratch in DK version 1.1 or greater. Mixing old and new versions of code (by linking in libraries or object files) may mean that everything has a single name scope.

Examples

Actel/Altera external port interfaces, Xilinx pad blocks (external pins):

```
BT1_s4c_4_PADIN_101_ClockInPin_0
B6_s4c_19_PADIN_158_in_1
B7_s4c_19_PADIN_159_in_0
B10_s4c_20_PADOUT_163_Param0_1
B11_s4c_20_PADOUT_164_Param0_0
```

Actel/Altera/Xilinx other blocks:

```
BT2_s4c_4_CLKBUF
B1_s4c_17_DTYPE0
B5_s4c_17_OR
B8_s4c_19_IBUF
B22_s4c_22_BRAM
B1_s4c_25_main_DTYPE1
B5_s4c_29_main_NOT
```

Netnames

```
WGT1_s4c_4_ClockInput
WGT9_s4c_26_CforkIn
WGT6_s4c_28_SeqChain
WGT8_s4c_29_UnaryOpOut_I_0
WGT7_s4c_29_UnaryOpOut_I_1
W1_s4c_25_main
WT1_s4c_4
W11_s4c_20_x_Out_I_0
W10_s4c_20_x_Out_I_1
```

12.5.2 Specifying wire name format in EDIF

You can specify the format of floating wire names in EDIF using the Handel-C `busformat` specification.

This allows you to use the formats:

```
BI        B_I      B[I]        B(I)       B<I>
```

where `B` represents the bus name, and `I` the wire number.

To specify the format of bus wire names use

```
"B"   B[N:0]
```

Example

```
interface port_in(int 4 signals_to_HC with
                      {busformat="B[I]"}) read();
```

This code would produce wires:

```
signals_to_HC[0]
signals_to_HC[1]
signals_to_HC[2]
signals_to_HC[3]
```

12.5.3 Setting up place and route tools

The Altera EDIF compiler requires a library-mapping file. This is supplied as `handelc.lmf`.

If you are targeting Actel devices, you need to import the timing constraints file generated by DK into Actel Designer.

12.5.4 Preparing MaxPlus II to compile Handel-C generated EDIF

1. Start MaxPlus II.
2. Open **MaxPlus II**>**Compiler**.
3. Open the Handel-C-generated EDIF netlist, and any other design files.
4. With the compiler selected, select **Interfaces**>**EDIF Netlist Reader Settings**.
5. In the dialog box, specify **Vendor** as **Custom**.
6. Click the **Customize>>** button.
7. Select the **LMF #1** radio button. Set up the path name for the `handelc.lmf` file (installed in ***InstallDir***`\DK\Lmf`).

12.5.5 Preparing Quartus to compile Handel-C generated EDIF

You need to set up Quartus in different ways depending on which version you are using, and whether you have compiled your EDIF using DK, or used a synthesis tool to convert DK VHDL or Verilog to EDIF.

DK EDIF, Quartus version 2.2 (or newer)

1. Start Quartus.
2. Create or open the project in which you want to compile the netlist generated by Handel-C.
3. Add the Handel-C-generated EDIF netlist, and any other design files, to the project.
4. Select the **Assignments>EDA Tool Settings** menu command.

5. In the **EDA Tool Setting** pane, select **Design entry/synthesis** as the **Tool Type**.
6. Select **Custom** as the **Tool name** from the drop-down list.
7. Click **Settings...**
8. Set the **Library Mapping File** to specify the `handelc.lmf` file installed in ***InstallDir***`\DK\Lmf`.
9. Apply the TCL script that was generated by DK when compiling the Handel-C code to EDIF. The script file has the same file name as the compiled file.
10. To apply the script:

    Enter the following command in the Quartus console window:
    source ***hcedif***`.tcl`
    where ***hcedif*** is the name of the file compiled to EDIF.

    OR

    Select the **Tools**>**Tcl scripts**. Expand the **Projects** folder, select the TCL file to run and click **Run**.
    (The TCL files in the Projects folder will be those in the same directory as your EDIF files for the project).

You can now do the placing and routing.

DK EDIF, Quartus version 2.1 (or older)

1. Start Quartus.
2. Create or open the project in which you want to compile the netlist generated by Handel-C.
3. Add the Handel-C-generated EDIF netlist, and any other design files, to the project.
4. Select the **Project>EDA Tool Settings** menu command.
5. In the dialog box, use the pull-down list to set **Custom** as the **Design entry/synthesis tool**.
6. Click **Settings...**
7. Set the **Library Mapping File** to specify the `handelc.lmf` file installed in ***InstallDir***`\DK\Lmf`.
8. Apply the TCL script that was generated by DK when compiling the Handel-C code to EDIF. The script file has the same file name as the compiled file.
9. To apply the script:

    Enter the following command in the Quartus console window:
    source ***hcedif***`.tcl`
    where ***hcedif*** is the name of the file compiled to EDIF.

    OR

    Select the **Tools**>**Run script** option and specify the TCL file.

You can now do the placing and routing.

DK HDL converted to EDIF using a synthesis tool

If you use DK to generate VHDL or Verilog output, and then use a synthesis tool such as LeonardoSpectrum to compile this to EDIF, you need to select the 'Power-Up Don't Care' option in Quartus (v2.1 or v2.2):

If you are using the command line:

- In the Tcl console window type:
  `project add_assignment "" "" "" "" ALLOW_POWER_UP_DONT_CARE Off`
  Then press **Return**.

If you are using the GUI:

- For Quartus II v2.1:
  Select **Project**>**Option & Parameter Settings**
  Then choose the 'Power-Up Don't Care' from the 'Existing option settings:' list, and set it to 'Off'
- For Quartus II v2.2:
  Select **Assignments**>**Settings**>**Default Logic Options Settings**
  Then choose the 'Power-Up Don't Care' from the 'Existing option settings:' list, and set it to 'Off'

This only needs to be done once for the whole project. You do not need to set this option if you are compiling EDIF generated directly by DK.

12.5.6 Importing timing constraint files into Actel Designer

To import `.gcf` files for Actel ProASIC and ProASIC+:

1. Start Designer.
2. Create or open the design in which you want to compile the netlist generated by Handel-C.
3. Import the Handel-C-generated EDIF netlist, and any other design files, to the project.
4. Use **File**>**Import...** to import the generated `.gcf` file that includes the timing constraints.
5. Compile the design.
6. Ensure that you select **Timing driven** for the layout.
7. Lay out the design.

You can now do the placing and routing.

12.6 Targeting hardware via Verilog

To target hardware via Verilog, set the **Build**>**Set Active Configuration** option to Verilog. This compiles directly to a `.v` file which can be passed to your synthesis or simulation tools. You cannot compile Handel-C to Verilog from Nexus PDK.

You must specify the synthesis tool that you are using the **Linker** tab in **Project Settings**. If you wish to simulate your Verilog, select ModelSim as the tool used.

The code generated is structured and relates the Handel-C function names to the Verilog module names.

If you want to simulate Verilog produced from Handel-C code or you want to target Xilinx devices, you need to link to a ROC file.

In previous versions of DK, you could generate less readable Verilog by un-checking the **Generate Debug information** box on the **Compiler** tab of **Project Settings** or using the command line -g option. This option is no longer available. All Verilog output now has names generated from Handel-C variable names.

If you want to co-simulate Handel-C with Verilog code you can use the Co-simulation Bridge for ModelSim provided in Celoxica's Platform Developer's Kit.

12.6.1 Verilog file structure

Each Handel-C source file is mapped to a Verilog file. Each Handel-C function is mapped to a module. There is also a top-level Verilog file which links the design modules together and contains global design ports. Macros are converted to inlined Verilog. Source files consisting only of macro expressions or macro procedures will be converted to an empty file and then deleted.

In previous versions of DK you could compile your Verilog without debug information to produce less readable code. This option is no longer available; all Verilog output now has names generated from Handel-C variable names. The -g option now has no effect for Verilog output.

File names

Verilog file names depend on whether you build your files using the DK GUI, or from the command line compiler.

If you use the command line compiler, you specify the name of your top-level output file using the -o option. For example:

```
handelc -verilog -o OutputFile
```

will produce a top level file called `OutputFile.v`

If you are using the GUI, the top-level file is called ***ProjectName***_top.v.

Other files are named after the Handel-C files: ***FileName***.***suffix*** source files became ***FileName_suffix***.v. For example, `UsefulFile.hcc` becomes `UsefulFile_hcc.v`.

If there is more than one source file with the same name, further files are called ***FileName_suffix_N***.v, where ***N*** increments from 1.

12. Targeting hardware

Modules

The top-level Verilog file contains a module with the same name as this file (without the `.v` extension).

Each Verilog file corresponding to a Handel-C source file starts with a module containing global logic defined within that file called ***FileName_suffix***. For example, the global variables in `UsefulFile.hcc` are stored in the entity `UsefulFile_hcc`.

Functions are mapped onto entities called ***FileName_suffix_FunctionName***. For example, a function called `MyFunction` defined in `MySource.hcc` becomes an entity called `MySource_hcc_MyFunction`.

Shared functions have a set of inputs for each use of the function.

`inline` functions have separate modules for each use of the function. The first instance of the function generates an module as above. Later instances have numbers appended to the name, starting at 1. For example, the fourth instance of the inline function `FastFunction` in `UsefulFile.hcc` becomes `UsefulFile_hcc_FastFunction_3`.

Global reset

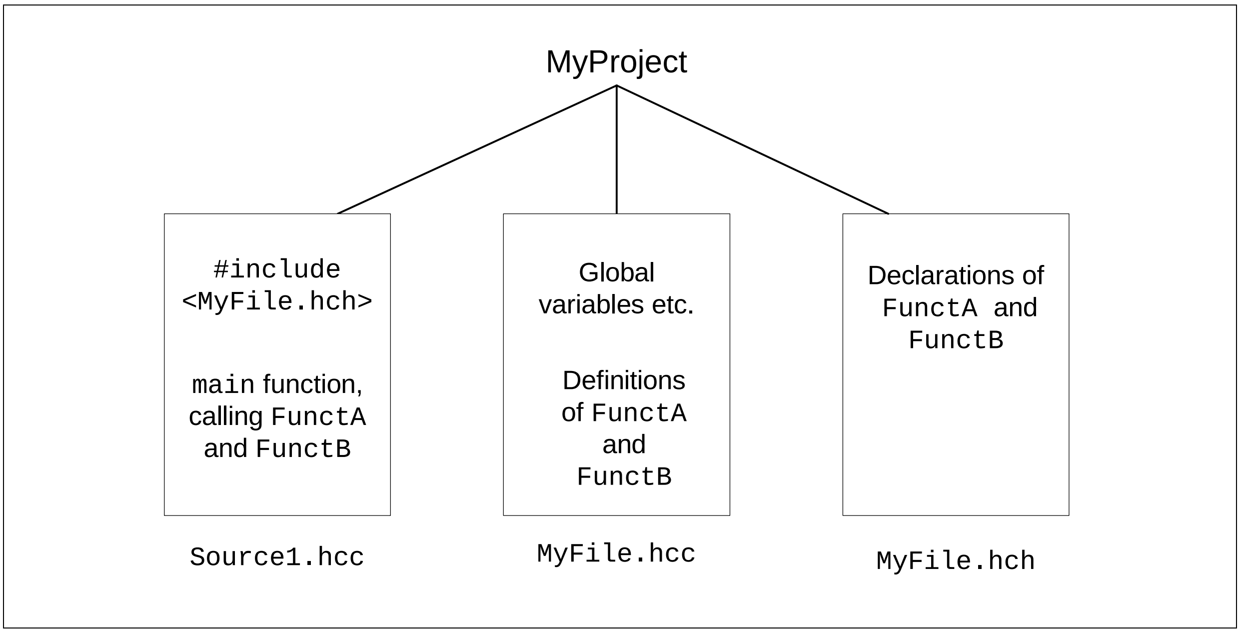
A global reset line is connected to all flip-flops/registers in the design. You can specify a reset pin using the `set reset` construct.

You must specify a reset pin for Actel devices. If no reset is specified for Altera or Xilinx devices, the registers in the design are reset on configuration. Altera devices have the reset wire connected to ground, Xilinx devices use a ROC block.

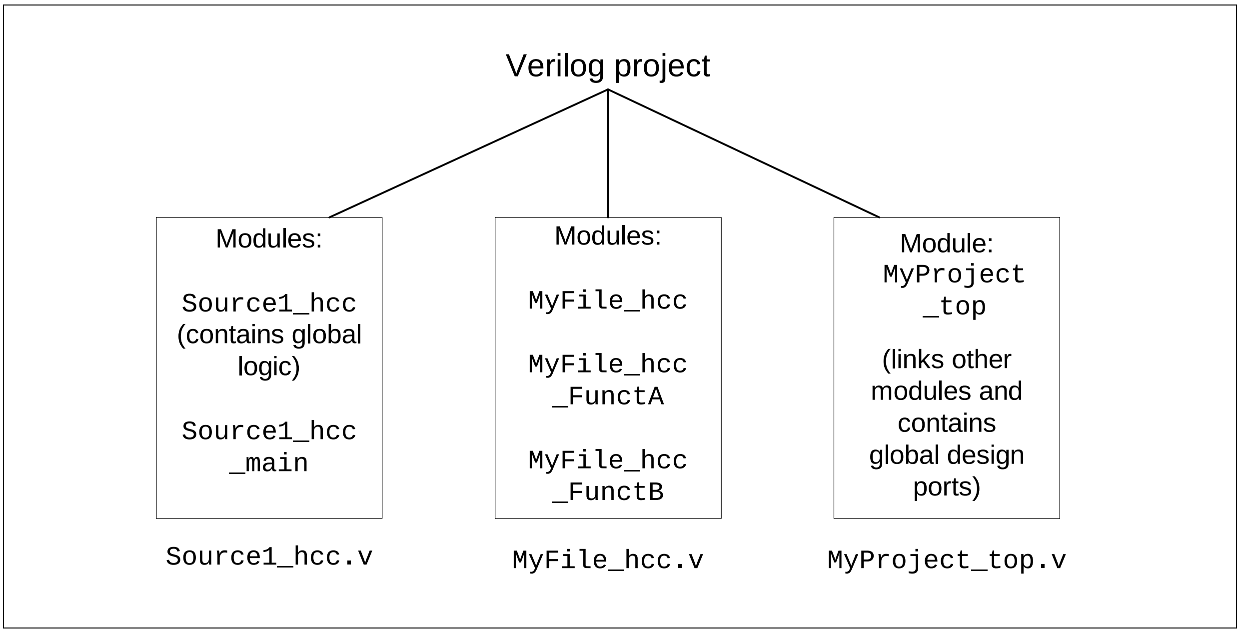
If you specify a reset using the `data` specification, a reset pin (called ***Data***) is added to the top-level module of the design. If you specify a reset without a pin, the reset pin will be called `ResetPin`.

12.6.2 Naming of Verilog files and modules

If you compile the Handel-C project shown below to Verilog using the GUI:



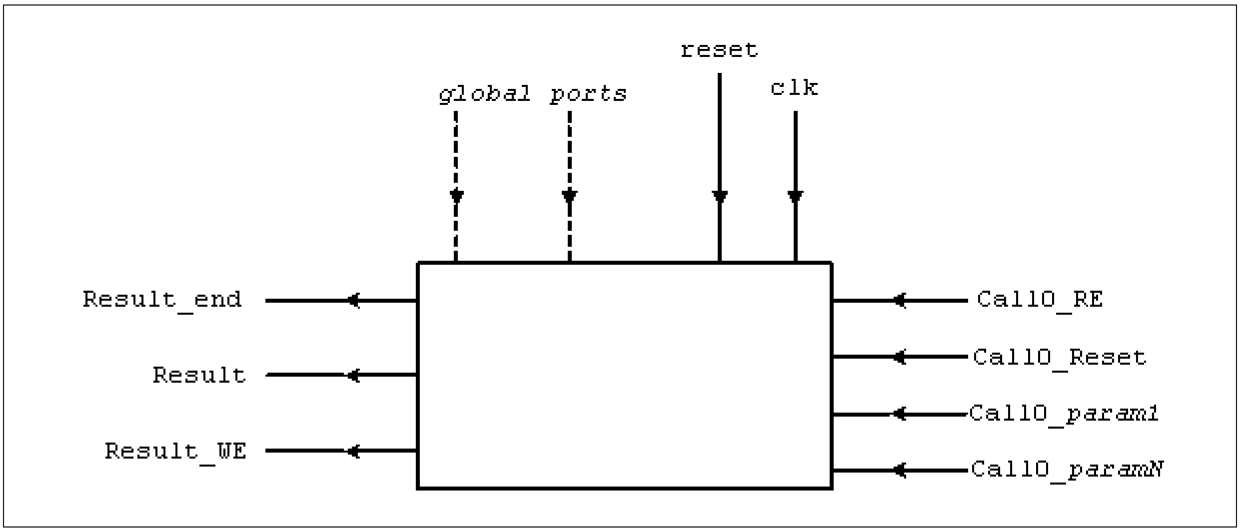
The following Verilog files and modules are produced:



If you compile the files from the command line, the top-level file and its module are named after the output file name you specify using the -o option. The other file and module names are the same as those shown above.

12.6.3 Mapping Handel-C functions to Verilog modules

A module generated from a Handel-C function will have the following inputs and outputs:



There is an input port for each parameter to the function. It is given the name `Call`***`N_parameterName`*** and is of the width of that parameter. For example, the function `add(int 8 a, unsigned 16 b)` in the file `maths.hcc` would be converted to an module `maths_hc_add`. The first use of the function would generate an 8-bit port called `Call0_a` and a 16-bit port called `Call0_b`.

Each call to a shared function will duplicate the numbered ports with an incremented number. The result lines are the same for each call to the function.

When the function is called, the `Call`***`N`***`_RE` port is set high. One clock cycle before the function has completed, the `Result_WE` port is set high. When the function completes, the result appears on the `Result` port, and the `Result_End` line is set high.

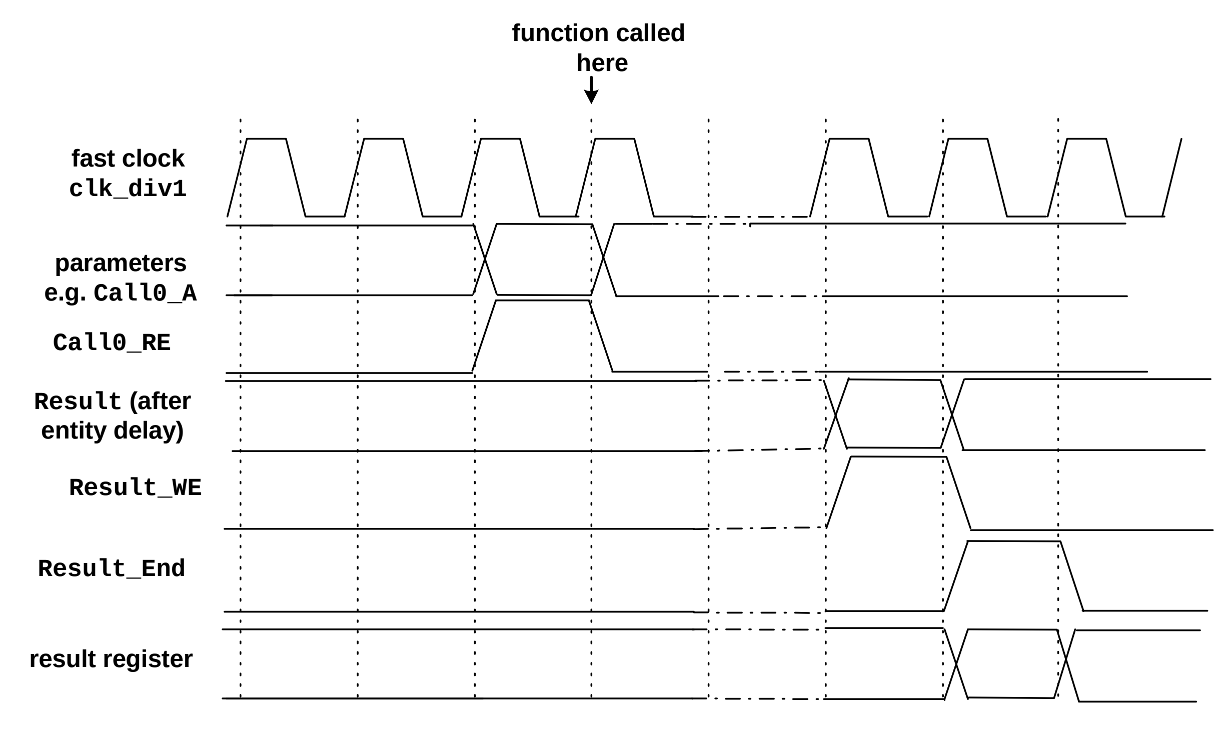
The `Call`***`N`***`_reset` port is used by the `try... reset` construct. If this construct was not used, the port will be connected to ground.

Global ports are produced from signals that cross function boundaries, such as global variables, ground and power. The names of global ports are prefixed with `globals_`.

If you have used the `set reset` construct in your code, the `reset` port will be replaced by a global port.

The name of the clock port depends on whether you specify a clock divide. If there is no clock divide, or the clock is divided by 1, the port will be called `clk`. If the clock divide is more than 1, the name of the port will be `clk_div`***`N`***, where ***N*** is the value of the divide.

Timing



When the Result_WE signal is asserted, the result can be written to the result register on the next rising clock edge.

The Result_End signal is asserted in the clock cycle before the module logic is released.

13. Integrating Handel-C with VHDL, Verilog and EDIF

There are two ways of interfacing Handel-C with external VHDL, Verilog or EDIF blocks:

- Calling a VHDL, Verilog or EDIF component from within a Handel-C project
- Calling a Handel-C component from within a VHDL, Verilog or EDIF project

The Handel-C uses an `interface` construct to communicate with the HDL/EDIF, but the way you write the connections is slightly different in these two cases.

If the Handel-C is the top level, it identifies the HDL/EDIF component it must connect to by using the component's HDL/EDIF name as the Handel-C `interface` ***`Sort`***. (For VHDL, if the ports generated by the Handel-C are of a different type to those used in VHDL, you will need a wrapper file to connect the two types of ports together.)

If the VHDL, Verilog or EDIF is the top level, the Handel-C needs to use the `port_in` and `port_out` interface sorts to provide connections to the external logic. You must then write a VHDL, EDIF or Verilog wrapper file to create the wires between the Handel-C ports and the HDL/EDIF. Sample wrapper files are provided with the examples in the `VHDL`, `Verilog` and `EDIF` directories within ***`InstallDir`***`\DK\Examples`.

Co-simulating Handel-C with Verilog and VHDL

If you want to co-simulate Handel-C code with VHDL or Verilog you can use the Co-simulation Bridge for ModelSim. This is provided in Celoxica's Platform Developer's Kit.

Reset on configuration (ROC)

Reset on configuration (ROC) is a component that defines the reset behaviour on the configuration of the FPGA. You need to link to a ROC file when you want to

- simulate VHDL or Verilog produced from DK using a simulator such as ModelSim
- compile or synthesize Handel-C to VHDL or Verilog to target Xilinx devices, unless you have specified a global reset (`set reset`).

RESET ON CONFIGURATION DIAGRAM

13. Integrating Handel-C with VHDL, Verilog and EDIF

You need to compile the appropriate VHDL (`*.vhd`) or Verilog (`*.v`) file into your work library. Two different versions of the ROC files are supplied:

- `simroc.vhd` or `simroc.v` - Simulation ROC. Use these when simulating VHDL or Verilog.
- `xilroc.vhd` or `xilroc.v` - Xilinx ROC. These instantiate the standard Xilinx ROC component. If you wish for different behaviour, you will need to replace the file. Refer to Xilinx documentation on the ROC component.

If you are targeting Altera or Actel devices, you do not need a ROC file; flip-flops are automatically reset to zero after configuration.

13.1 Integrating with VHDL blocks

If you want to co-simulate Handel-C with VHDL you can use the Co-simulation Bridge for ModelSim, provided as part of the Platform Developer's Kit.

13.1.1 Linking to the Handel-C VHDL library

Celoxica supplies the `HandelC.vhd` file which provides functions needed by all Handel-C VHDL files.

To use Handel-C VHDL, you must compile the `HandelC.vhd` file into a library called `HandelC` in Precision, LeonardoSpectrum or Synplify.

Consult the documentation for your synthesis or simulation tool on compiling library files.

If you are targeting a Xilinx device or want to simulate your VHDL code in ModelSim, you need to compile one of the supplied ROC files into your work library. You only need to do this if the global reset (`set reset`) is not specified.

- For simulation, use `simroc.vhd`
- For Xilinx devices, use `xilroc.vhd`

You do not need to use a ROC file to target Altera or Actel devices as flip-flops are automatically reset to zero after configuration.

13. Integrating Handel-C with VHDL, Verilog and EDIF

13.1.2 Writing Handel-C code to integrate with VHDL code

Using Handel-C as the top level

In a top-level Handel-C program communicating with a VHDL entity you will need:

An interface declaration:	Prototypes the interface sort. The interface sort must have the same name as the VHDL entity. If you have only one instance of the entity in your code, and you are not referring forward to a definition, you may incorporate the declaration into the definition.
An interface definition:	Names the instance and defines the data that will be transmitted.

Using VHDL as the top level

In a Handel-C program communicating with a top-level VHDL entity, you only need a port_in or port_out interface for each port going into or out of the Handel-C component.

Handel-C to VHDL: interface declaration

The VHDL interface declaration in the Handel-C code prototypes the interface sort, and is of the format:

```
interface VHDL_entity_sort
          (VHDL_to_HC_port {,VHDL_to_HC_port })        //input ports
          (VHDL_from_HC_port {, VHDL_from_HC_port});   //output ports
```

where:

- ***VHDL_entity_sort*** is the name of the HDL component. The same name must be used as the interface sort in the interface definition.
- ***VHDL_to_HC_port*** is the type and name of a port bringing data to the Handel-C code (output from VHDL) as specified in the VHDL entity.
- ***VHDL_from_HC_port*** is the type and name of a port sending data from the Handel-C code (input to VHDL) as specified in the VHDL entity.

Note that ports are seen from the VHDL side, so port names may be confusing. In Handel-C, the ports that input data TO the Handel-C must be specified first.

Handel-C to VHDL: interface definition

The VHDL interface definition in the Handel-C code creates an instance of the interface sort prototyped in the declaration. It also gives the names of the interface and port instances and defines the data that will be transmitted.

The definition is of the format:

```
interface VHDL_entity_sort
             (VHDL_to_HC_port [with portSpec]
             {, VHDL_to_HC_port [with portSpec]})
          interface_Name
             (VHDL_from_HC_port = from_HC_data [with portSpec]
             {, VHDL_from_HC_port = from_HC_data [with portSpec]});
```

where:

- ***VHDL_entity_sort*** is the interface sort that you previously declared.
- ***VHDL_to_HC_port*** is the type and name of a port bringing data to the Handel-C code (output from VHDL). This will have the same type as defined in the interface declaration.
- ***interface_Name*** is the name for this instance of the interface.
- ***VHDL_from_HC_port*** is the type and name of a port sending data from the Handel-C code (input to VHDL). This will have the same type as defined in the interface declaration.
- ***from_HC_data*** is an expression that is output from the Handel-C to the VHDL.
- with ***portSpec*** is an optional port specification.

13.1.3 Example: VHDL in a Handel-C project

The example below demonstrates how to interface Handel-C and VHDL components, when Handel-C is the top level of your design.

Handel-C code

```
set clock = external "D17";
unsigned 4 x;
interface vhdl_component
          (unsigned 4 return_val)
     vhdl_component_instance
          (unsigned 1 clk = __clock,
           unsigned 4 sent_value = x) with {busformat = "B_I"};

void main(void)
{
  unsigned 4 y;
  y = vhdl_component_instance.return_val;   // Read from VHDL component
  x = y;                                    // Write to VHDL component
}
```

VHDL code

The VHDL entity needs an interface like this to be compatible with the Handel-C:

```
entity vhdl_component is
  port (
    clk           : in   std_logic;
    sent_value_0 : in   std_logic;
    sent_value_1 : in   std_logic;
    sent_value_2 : in   std_logic;
    sent_value_3 : in   std_logic;
    return_val_0 : out  std_logic;
    return_val_1 : out  std_logic;
    return_val_2 : out  std_logic;
    return_val_3 : out  std_logic
  );
end;
```

Note that all the ports are 1-bit wide, `std_logic` types. This matches to the EDIF generated using the "B_I" `busformat`. Using a different `busformat` specification will give two 4-bit ports and one 1-bit port, but you need to ensure that the format matches the output from your synthesis tool.

13.1.4 Example: Handel-C in a VHDL project

The example below demonstrates how to interface Handel-C and VHDL components, when VHDL is the top level of your design. The Handel-C needs to have ports to its top level, so that the VHDL can connect to them.

```
unsigned 4 x;

interface port_in
        (unsigned 1 clk with {clockport=1})
    ClockPort
        ();

interface port_in
        (unsigned 4 sent_value)
    InPort
        ();

//cont...
```

```
interface port_out
        ()
    OutPort
        (unsigned 4 return_value = x);

set clock = internal ClockPort.clk;

void main(void)
{
  unsigned 4 y;
  y = InPort.sent_value;       // Read from top-level VHDL
  x = y;                      // Write to top-level VHDL
}
```

You can compile the Handel-C to EDIF or VHDL. If you compile to EDIF, you can use the busformat specification to specify the bus and wire name format.

VHDL code

The top level VHDL must instantiate the Handel-C. The way you do this is slightly different for Handel-C targeting EDIF and Handel-C targeting VHDL. The example below shows EDIF generating a bus as single wires.

Instantiating Handel-C code compiled to EDIF

```
component handelc_component
  port (
    clk          : in   std_logic;
    sent_value_0 : in   std_logic;
    sent_value_1 : in   std_logic;
    sent_value_2 : in   std_logic;
    sent_value_3 : in   std_logic;
    return_val_0 : out  std_logic;
    return_val_1 : out  std_logic;
    return_val_2 : out  std_logic;
    return_val_3 : out  std_logic
  );
end component;
```

13. Integrating Handel-C with VHDL, Verilog and EDIF

Instantiating Handel-C code compiled to VHDL

```
component handelc_component
  port (
    clk          : in std_logic;
    sent_value : in unsigned (3 downto 0);
    return_val : out unsigned (3 downto 0);
  );
end component;
```

13.1.5 Synthesizing Handel-C with external VHDL

Synthesis and place and route

When you are ready to synthesize, you may follow a VHDL or EDIF flow:

- VHDL flow

  Compile the Handel-C to VHDL.

  Use Precision, Synplify or LeonardoSpectrum to synthesize the code. Then use Xilinx, Altera or Actel tools to place and route it.
- EDIF flow

  Compile Handel-C to EDIF.

  Use Precision, Synplify or LeonardoSpectrum to synthesize any VHDL components to EDIF. Use Xilinx, Altera or Actel tools to merge the EDIF files together and place and route them.

Simulation

You can co-simulate Handel-C code with VHDL using ModelSim: compile the Handel-C for debug, and then use the Co-simulation Bridge for ModelSim supplied in the Platform Developer's Kit.

13.1.6 Connecting Handel-C EDIF to VHDL

If you compile a Handel-C file to EDIF 2.0.0 and wish to connect it to a VHDL component, you must be aware that the ports in EDIF and VHDL may be different:

- EDIF 2.0.0 ports may consist of a collection of single wires or a ***n***-wire bus.
- VHDL ports are normally described as ***n***-bit wide cables.

The format of the EDIF port can be defined using the Handel-C `busformat` specification. The particular format needed is dependent upon the synthesis tool. For example, LeonardoSpectrum generates angle-brackets to delimit buses, so the busformat specification used to generate a multi-wire bus would be `busformat = "B<N:0>"`.

13. Integrating Handel-C with VHDL, Verilog and EDIF

If you have not used `busformat` to generate a multi-wire bus, you can ensure that the generated EDIF can connect to the VHDL by listing the VHDL ports as single-bit wires.

13.2 Integrating with Verilog blocks

If you want to co-simulate Handel-C with Verilog you can use the Co-simulation Bridge for ModelSim, provided as part of the Platform Developer's Kit.

13.2.1 Linking to the Handel-C Verilog library

Celoxica supplies the `HandelC.v` file which provides functions needed by all Handel-C Verilog files.

To use Handel-C Verilog, you must add `HandelC.v` to your work library within Precision, LeonardoSpectrum or Synplify.

If you are targeting a Xilinx device or want to simulate your Verilog code using ModelSim, you need to compile one of the supplied ROC files into your work library.

- For simulation, use `simroc.v`
- For Xilinx devices, use `xilroc.v`

You only need to do this if the global reset (`set reset`) is not specified. You do not need to use a ROC file to target Altera or Actel devices as flip-flops are automatically reset to zero after configuration.

13.2.2 Writing Handel-C code to integrate with Verilog code

Using Handel-C as the top level

In a top-level Handel-C program communicating with a Verilog module you will need:

An interface declaration:	Prototypes the interface sort. The interface sort must have the same name as the Verilog module. If you have only one instance of the Verilog module in your code, and you are not referring forward to a definition, you may incorporate the declaration into the definition.
An interface definition:	Names the instance and defines the data that will be transmitted.

Using Verilog as the top level

In a Handel-C program communicating with a top-level Verilog entity, you only need a `port_in` or `port_out` interface for each port going into or out of the Handel-C component.

13. Integrating Handel-C with VHDL, Verilog and EDIF

Handel-C to Verilog: interface declaration

The Verilog interface declaration in the Handel-C code prototypes the interface sort, and is of the format:

```
interface Verilog_module_sort
          (Verilog_to_HC_port {,Verilog_to_HC_port })      //input ports
          (Verilog_from_HC_port {,Verilog_from_HC_port});  //output ports
```

where:

- ***Verilog_module_sort*** is the name of the Verilog module. The same name must be used as the interface sort in the interface definition.
- ***Verilog_to_HC_port*** is the type and name of a port bringing data to the Handel-C code (output from Verilog) as specified in the Verilog module.
- ***Verilog_from_HC_port*** is the type and name of a port sending data from the Handel-C code (input to Verilog) as specified in the Verilog module.

Note that ports are seen from the Verilog side, so port names may be confusing. In Handel-C, the ports that input data TO the Handel-C must be specified first.

Handel-C to Verilog: interface definition

The Verilog interface definition in the Handel-C code creates an instance of the interface sort prototyped in the declaration. It also gives the names of the interface and port instances and defines the data that will be transmitted.

The definition is of the format:

```
interface Verilog_module_sort
              (Verilog_to_HC_port [with portSpec]
               {, Verilog_to_HC_port [with portSpec]})
          interface_Name
              (Verilog_from_HC_port = from_HC_data [with portSpec]
                 {, Verilog_from_HC_port = from_HC_data [with portSpec]});
```

where:

- ***Verilog_module_sort*** is the interface sort that you previously declared.
- ***Verilog_to_HC_port*** is the type and name of a port bringing data to the Handel-C code (output from Verilog). This will have the same type as defined in the interface declaration.
- ***interface_Name*** is the name for this instance of the interface.
- ***Verilog_from_HC_port*** is the type and name of a port sending data from the Handel-C code (input to Verilog). This will have the same type as defined in the interface declaration.
- ***from_HC_data*** is an expression that is output from the Handel-C to the Verilog.
- with ***portSpec*** is an optional port specification.

13.2.3 Example: Verilog in a Handel-C project

The example below demonstrates how to interface Handel-C and Verilog components, when Handel-C is the top level of your design.

Handel-C code

```
set clock = external "D17";
unsigned 4 x;


interface verilog_component
          (unsigned 4 return_val)
     verilog_component_instance
          (unsigned 1 clk = __clock,
          unsigned 4 sent_value = x)
          with {busformat = "B_I"};

void main(void)

{
  unsigned 4 y;
  y = verilog_component_instance.return_val;  // Read from Verilog
component
  x = y;                                       // Write to Verilog component
}
```

Verilog code

The Verilog module will need an interface like this to be compatible with the Handel-C:

```
module verilog_component(clk, sent_value_0, sent_value_1, sent_value_2,
                         sent_value_3, return_val_0, return_val_1,
                         return_val_2, return_val_3);
  input clk;
  input sent_value_0;
  input sent_value_1;
  input sent_value_2;
  input sent_value_3;
  output return_val_0;
  output return_val_1;
  output return_val_2;
  output return_val_3;

endmodule
```

Note that all the ports are 1-bit wide. This matches to the EDIF generated using the "B_I" busformat. Using a different busformat specification will give give two 4-bit ports and one 1-bit port, but you need to ensure that the format matches the output from your synthesis tool.

13.2.4 Example: Handel-C in a Verilog project

The example below demonstrates how to interface Handel-C and Verilog components, when Verilog is the top level of your design. The Handel-C needs to have ports to its top level, so that the Verilog can connect to them.

```
unsigned 4 x;

interface port_in
        (unsigned 1 clk with {clockport=1})
    ClockPort
        ();

interface port_in
        (unsigned 4 sent_value)
    InPort
        ();

interface port_out
        ()
    OutPort
        (unsigned 4 return_value = x);

set clock = internal ClockPort.clk;

void main(void)
{
  unsigned 4 y;
  y = InPort.sent_value;              // Read from top-level Verilog
  x = y;                              // Write to top-level Verilog
}
```

You can compile the Handel-C to EDIF or to Verilog. If you compile to EDIF, you can use the busformat specification to specify the bus and wire name format.

Verilog code

The top level Verilog must instantiate the Handel-C. The way you do this is slightly different for Handel-C targeting EDIF and Handel-C targeting Verilog. The example below shows EDIF generating a bus as single wires

13. Integrating Handel-C with VHDL, Verilog and EDIF

Instantiating Handel-C code compiled to EDIF

```
module handelc_component(clk, sent_value_0, sent_value_1, sent_value_2,
                         sent_value_3, return_val_0, return_val_1,
                         return_val_2, return_val_3);
  input  clk;
  input  sent_value_0;
  input  sent_value_1;
  input  sent_value_2;
  input  sent_value_3;
  output return_val_0;
  output return_val_1;
  output return_val_2;
  output return_val_3;

endmodule
```

Instantiating Handel-C code compiled to Verilog

```
module handelc_component(clk, sent_value, return_val);

  input  clk;
  input  [3:0] sent_value;
  output [3:0] return_val;

endmodule
```

13.2.5 Synthesizing Handel-C with external Verilog

Synthesis and place and route

When you are ready to synthesize, you may follow a Verilog or EDIF flow:

- Verilog flow

  Compile the Handel-C to Verilog.

  Use Precision, Synplify or LeonardoSpectrum to synthesize the code. Then use Xilinx, Altera or Actel tools to place and route it.

- EDIF flow

  Compile Handel-C to EDIF.

  Use Precision, Synplify or LeonardoSpectrum to synthesize any Verilog components to EDIF. Use Xilinx, Altera or Actel tools to merge the EDIF files together and place and route them.

13. Integrating Handel-C with VHDL, Verilog and EDIF

Simulation

You can co-simulate Handel-C code with Verilog using ModelSim: compile the Handel-C for debug, and then use the Co-simulation Bridge for ModelSim supplied in the Platform Developer's Kit.

13.2.6 Connecting Handel-C EDIF to Verilog

If you compile a Handel-C file to EDIF 2.0.0 and wish to connect it to a Verilog component, you must be aware that the ports in EDIF and Verilog may be different:

- EDIF 2.0.0 ports may consist of a collection of single wires or a ***n***-wire bus.
- Verilog ports are normally described as ***n***-bit wide cables.

The format of the EDIF port can be defined using the Handel-C `busformat` specification. The particular format needed is dependent upon the synthesis tool. For example, LeonardoSpectrum generates angle-brackets to delimit buses, so the busformat specification used to generate a multi-wire bus would be `busformat = "B<N:0>"`.

If you have not used `busformat` to generate a multi-wire bus, you can ensure that the generated EDIF can connect to the Verilog by listing the Verilog ports as single-bit wires.

13.3 Integrating with EDIF blocks

13.3.1 Connecting Handel-C EDIF to external EDIF

To integrate Handel-C with raw EDIF:

- Use Handel-C as the top level of your design and instantiate one or more EDIF components as black boxes, by defining interfaces*.*

OR

- Use EDIF as the top level of your design and instantiate one or more Handel-C components as black boxes. Handel-C ports to the top level are declared using `port_in` and `port_out` interfaces.

Port formats

If you compile a Handel-C file to EDIF and want to connect it to a raw EDIF component, you must be ensure that port formats match between a component instantiation and a component instance. The Handel-C `busformat` specification allows you to specify EDIF bus formats on a per-port basis, allowing maximum flexibility to connect to raw EDIF from any source.

Simulating a Handel-C/EDIF design

If you want to simulate a design composed of EDIF and Handel-C blocks, use your place and route tools to generate a post-PAR annotated VHDL netlist. The netlist can then be used to run a timing-accurate simulation using ModelSim.

13.3.2 Writing Handel-C code to integrate with external EDIF

In a Handel-C program communicating with EDIF you will need:

An interface declaration:	Prototypes the interface sort. The interface sort must have the same name as the black box or primitive. If you have only one instance of the logic block in your code, and you are not referring forward to a definition, you may incorporate the declaration into the definition.
An interface definition:	Names the instance and defines the data that will be transmitted.

Handel-C to EDIF: interface declaration

The EDIF interface declaration in the Handel-C code prototypes the interface sort, and is of the format:

```
interface EDIF_symbol                           //name of EDIF symbol
   (EDIF_to_HC_port {,EDIF_to_HC_port })        //input ports to Handel-C
   (EDIF_from_HC_port {, EDIF_from_HC_port}); //output ports from Handel-C
```

where:

- ***EDIF_symbol*** is the name of the EDIF symbol. The same name must be used as the interface sort in the interface definition.
- ***EDIF_to_HC_port*** is the type and name of a port bringing data to the Handel-C code (output from EDIF) as specified in the unwrapped EDIF symbol.
- ***EDIF_from_HC_port*** is the type and name of a port sending data from the Handel-C code (input to EDIF) as specified in the unwrapped EDIF symbol.

Note that ports are seen from the EDIF side, so port names may be confusing. In Handel-C, the ports that input data TO the Handel-C must be specified first.

Handel-C to EDIF: interface definition

The EDIF interface definition in the Handel-C code creates an instance of the interface sort prototyped in the declaration. It also gives the names of the interface and port instances and defines the data that will be transmitted.

The definition is of the format:

```
interface EDIF_symbol
             (EDIF_to_HC_port [with portSpec]
              {, EDIF_to_HC_port [with portSpec]})
                 interface_Name
             (EDIF_from_HC_port = from_HC_data [with portSpec]
                {, EDIF_from_HC_port = from_HC_data [with portSpec]});
```

where:

- ***EDIF_symbol*** is the interface sort that you previously declared.
- ***EDIF_to_HC_port*** is the type and name of a port bringing data to the Handel-C code (output from EDIF). This will have the same type as defined in the interface declaration.
- ***interface_Name*** is the name for this instance of the interface.
- ***EDIF_from_HC_port*** is the type and name of a port sending data from the Handel-C code (input to EDIF). This will have the same type as defined in the interface declaration.
- ***from_HC_data*** is an expression that is output from the Handel-C to the EDIF.
- with ***portSpec*** is an optional port specification, e.g. busformat.

13.3.3 Example: Handel-C in an EDIF project

The example below demonstrates how to interface Handel-C and external EDIF components, when the external EDIF is the top level of your design.

Handel-C code

The Handel-C needs to have ports to its top level so that the EDIF can connect to them.

```
unsigned 4 val;

interface port_in
        (unsigned 1 clk)
    ClockPort
        ()
        with {busformat = "B<N:0>"};

interface port_in
        (unsigned 4 sent_val)
    ValInPort
        ()
    with {busformat = "B<N:0>"};

interface port_out
        ()
    ValOutPort
        (unsigned 4 return_val = val)
        with {busformat = "B<N:0>"};

//cont...
```

```
set clock = internal ClockPort.clk;

void main(void)
{
    while(1)
    {
        par
        {
            val *= ValInPort.sent_val;
        }
    }
}
```

EDIF netlist produced from the Handel-C code

The part of the netlist that describes the interface reads:

```
(interface
  (port (array (rename clk "clk<0:0>") 1) (direction INPUT))
  (port (array (rename sent_val "sent_val<3:0>") 4) (direction INPUT))
  (port (array (rename return_val "return_val<3:0>") 4) (direction OUTPUT))
 )
```

EDIF code

The external EDIF code needs to instantiate the EDIF block generated from the Handel-C code:

```
(cell edif_component
  (cellType GENERIC)
  (view view_1
    (viewType NETLIST)
    (interface
      (port (array (rename clk "clk<0:0>") 1) (direction INPUT))
      (port (array (rename sent_val "sent_val<3:0>") 4) (direction INPUT))
      (port (array (rename return_val "return_val<3:0>") 4) (direction
OUTPUT))
    )
  )
)
```

13.3.4 Example: EDIF component in a Handel-C project

The example below demonstrates how to interface Handel-C and external EDIF components, when Handel-C is the top level of your design.

Handel-C code

```
set clock = external "D17";
unsigned 4 x;
interface edif_component
(
    unsigned 4 return_val
)
edif_component_instance
(
    unsigned 1 clk = __clock,
    unsigned 4 sent_val = x
)
with
{
    busformat = "B"
};

void main(void)
{
    unsigned 4 y;

    while(1)
    {
        y = edif_component_instance.return_val; // read from EDIF component
        x += y; // write to EDIF component
    }
}
```

EDIF netlist produced from Handel-C code

The code above generates a component (black-box) instantiation in the EDIF netlist, which looks like this:

```
(cell edif_component
  (cellType GENERIC)
  (view view_1
    (viewType NETLIST)
    (interface
      (port (array clk 1) (direction INPUT))
      (port (array sent_val 4) (direction INPUT))
      (port (array return_val 4) (direction OUTPUT))
    )
  )
)
```

EDIF code

There needs to be an EDIF netlist for the black-box component, called edif_component, with ports which look like this:

```
  (interface
    (port (array clk 1) (direction INPUT))
    (port (array sent_val 4) (direction INPUT))
    (port (array return_val 4) (direction OUTPUT))
  )
```

13.4 Examples: integrating Handel-C with VHDL, Verilog and EDIF

VHDL examples

InstallDir\DK\Examples\VHDL

- Example 1: combinational circuit example (Handel-C top-level)
- Example 2: register bank circuit example (Handel-C top-level)
- Example 3: FIR filter example (VHDL top-level wrapper)

Verilog examples

InstallDir\DK\Examples\Verilog

- Example 1: combinational circuit example (Handel-C top-level)
- Example 2: register bank circuit example (Handel-C top-level)
- Example 3: FIR filter example (Verilog top-level wrapper)

13. Integrating Handel-C with VHDL, Verilog and EDIF

EDIF example

InstallDir`\DK\Examples\EDIF`

- Example 3: FIR filter example (EDIF top-level wrapper)

13.4.1 Integration examples: running

To synthesize the VHDL, Verilog or EDIF integration examples, you must:

1. For Verilog or VHDL examples, choose the HDL output style: select **Project**>**Settings**>**Linker**, and then chose an output style from the drop-down list. Choose the style that matches your RTL synthesis tool, or else choose **Generic**.
2. Compile the Handel-C code to VHDL, Verilog or EDIF, as appropriate.
3. For Verilog or VHDL examples, pass the compiled Handel-C and the HDL model to your synthesis tool.
4. Run the place and route.

You can only compile these examples if you have the full vesion of DK. Nexus PDK does not allow you to produce VHDL, Verilog or EDIF code.

13.5 Examples of interfacing to VHDL

Examples are supplied of three projects involving interfaces to VHDL blocks. The examples are installed in the directory `DK\Examples\Vhdl`.

Each consists of a Handel-C workspace, the VHDL code file for the circuit, a VHDL wrapper file that links the VHDL to the Handel-C, and a Handel-C file that connects to the VHDL circuit. If the Handel-C is the top level, it connects to the VHDL via an entity interface. If the VHDL is top-level, the Handel-C connects using `port_in` and `port_out` interfaces.

You can only compile these examples if you have the full version of DK.

13.5.1 Combinational circuit example: VHDL

The combinational circuit example (***InstallDir***`\DK\Examples\VHDL\Example1`) consists of these files:

`ttl7446.vhd`	VHDL code that describes the combinational circuit
`ttl7446_wrapper.vhd`	VHDL code that connects the combinational circuit to the Handel-C code
`ttl7446_test.hcc`	Handel-C code that uses the combinational circuit

You can open the files in a text editor such as Notepad. The example also includes DK workspace and project files.

Combinational circuit example: interface code to VHDL

The example defines an interface sort that has port names of the same name and type as the VHDL signals in the circuit to be integrated. The interface sort must be the same as the VHDL model's name.

The interface from `tt17746_test.hcc` is:

```
interface TTL7446
        (unsigned 7 segments, unsigned 1 rbon)
    decode
        (unsigned 1 ltn=ltnVal, unsigned 1 rbin=rbinVal,
         unsigned 4 digit=digitVal, unsigned 1 bin=binVal);
```

TTL7446 is the name of the interface sort.

Ports declared by the interface

Port name	Port direction	Port type
ltn	out	std_logic
rbin	out	std_logic
digit	out	unsigned (3 downto 0)
bin	out	std_logic
segments	in	unsigned (6 downto 0)
rbon	in	std_logic

13.5.2 Register bank example: VHDL

The register bank example (***InstallDir***`\DK\Examples\VHDL\Example2`) consists of these files:

reg32x1k.vhd	VHDL code that describes the register bank circuit
reg32x1k_wrapper.vhd	VHDL code that connects the register bank circuit to the Handel-C code
reg32x1k_test.hcc	Handel-C code that uses the register bank

You can open the files in a text editor such as Notepad. The example also includes DK workspace and project files.

13. Integrating Handel-C with VHDL, Verilog and EDIF

Register bank example: interface code to VHDL

The example defines an interface sort that has port names of the same name and type as the VHDL signals in the circuit to be integrated. The interface sort must be the same as the VHDL model's name.

The interface from `reg32x1k_test.hcc` is:

```
interface reg32x1k
        (unsigned 32 data_out)
    registers
        (unsigned 10 address = addressVal
          with {extpath = {registers.data_out}},
        unsigned 32 data_in = data_inVal,
        unsigned 1 ck = __clock,
        unsigned 1 write = writeVal);
```

`reg32xlk` is the name of the interface sort.

Ports declared by the interface

Port name	Port direction	Port type
data_out	in	unsigned (31 downto 0)
address	out	unsigned (9 downto 0)
data_in	out	unsigned (31 downto 0)
ck	out	std_logic
write	out	std_logic

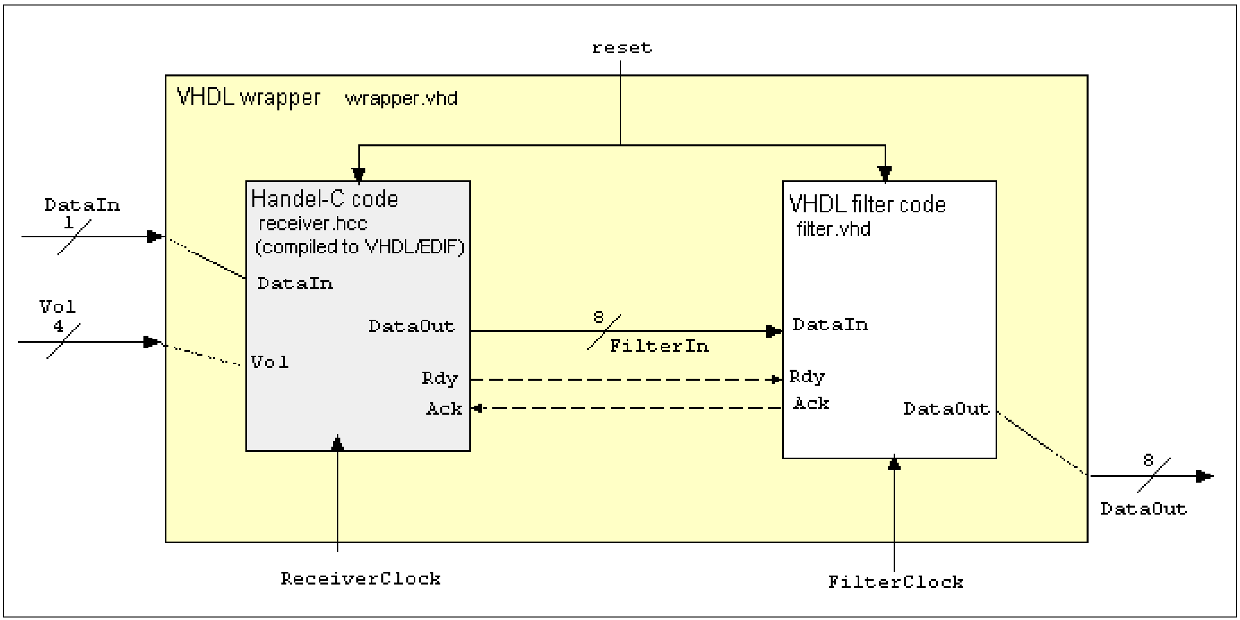
13.5.3 FIR filter example files: VHDL

The FIR filter example (***InstallDir***\DK\Examples\VHDL\Example3) consists of these files:

`filter.vhd`	VHDL code that describes the FIR filter
`wrapper.vhd`	VHDL code that connects the VHDL filter to the Handel-C receiver
`receiver.hcc`	Handel-C code that receives a bit stream, performs a volume change on it if required and converts it to 8-bit data

You can open the files in a text editor such as Notepad.

13. Integrating Handel-C with VHDL, Verilog and EDIF



LOGIC BLOCKS IN THE FIR FILTER EXAMPLE

The example also includes DK workspace and project files.

FIR filter example: interface code to VHDL

The Handel-C receiver is a component in a VHDL design. If your top-level code is VHDL, you must use the port_in and port_out interface types to communicate with the VHDL. The interfaces must have port names of the same name and type as the VHDL signals in the wrapper connecting to the Handel-C component to be integrated.

The interfaces between receiver.hcc and the VHDL wrapper are:

```
interface port_in             //interface type must be port_in or port_out
    (unsigned 1 DataIn)       // single bit input port - name used in VHDL
    ReadData                  // name of instance of port_in
    ()                        // no output ports
    with {std_logic_vector = 1}; //standard logic ports

interface port_out            //interface type must be port_in or port_out
    ()                        // no input ports
    WriteData                 // name of instance of port_out
    (unsigned 8 DataOut = Bytes_out) //output port (name used in VHDL)
    with {std_logic_vector = 1};     //standard logic ports

interface port_out            //interface type must be port_in or port_out
    ()                        // no input ports
    WriteRdy                  // name of instance of port_out
    (unsigned 1 Rdy = DataReady);   //name of output signal and its value
```

13. Integrating Handel-C with VHDL, Verilog and EDIF

```
interface port_in           //interface type must be port_in or port_out
    (unsigned 1 Ack)        // single bit input port - name used in VHDL
    ReadAck                 // name of instance of port_in
    ();                     // no output

interface port_in           //interface type must be port_in or port_out
    (unsigned 4 Vol)        //4 bit wide input port (name used in VHDL)
    Volume                  // name of instance of port_in
    ()                      // no output
    with {std_logic_vector = 1}; //standard logic ports
```

Ports declared by the interfaces

Port name	Port direction	Port type
DataIn	in	std_logic_vector (0 downto 0)
DataOut	out	std_logic_vector (7 downto 0)
Rdy	out	std_logic
Ack	in	std_logic
Vol	in	std_logic_vector (3 downto 0)

13.6 Examples of interfacing to Verilog

Examples are supplied of three projects interfacing to Verilog blocks. The examples are installed in the directory `DK\Examples\Verilog`.

Each consists of a Handel-C workspace, a Verilog code file for a circuit, a Verilog wrapper file that links the Verilog to the Handel-C, and a Handel-C file. If the Handel-C file is the top level, it connects to the Verilog via a module interface. If the Verilog is top-level, the Handel-C connects using port_in and port_out interfaces.

You can only compile these examples if you have the full version of DK.

13.6.1 Combinational circuit example: Verilog

The combinational circuit example (***InstallDir***`\DK\Examples\Verilog\Example1`) consists of these files:

`ttl7446.v`	Verilog code that describes the combinational circuit
`ttl7446_wrapper.v`	Verilog code that connects the combinational circuit to the Handel-C code
`ttl7446_test.hcc`	Handel-C code that uses the combinational circuit

You can open these files in a text editor such as Notepad. The example also includes DK workspace and project files.

Combinational circuit example: interface code to Verilog

The example defines an interface sort that has port names of the same name as the Verilog signals in the circuit to be integrated. The interface sort must be the same as the Verilog model's name.

The interface from `ttl7746_test.hcc` is:

```
interface TTL7446
        (unsigned 7 segments, unsigned 1 rbon)
    decode
        (unsigned 1 ltn=ltnVal, unsigned 1 rbin=rbinVal,
         unsigned 4 digit=digitVal, unsigned 1 bin=binVal);
```

TTL7446 is the name of the interface sort.

Ports declared by the interface

Port name	Port direction
`ltn`	out
`rbin`	out
`digit`	out
`bin`	out
`segments`	in
`rbon`	in

13.6.2 Register bank example: Verilog

The register bank example (***InstallDir***`\DK\Examples\Verilog\Example2`) consists of these files:

`reg32x1k.v`	Verilog code that describes the register bank circuit
`reg32x1k_wrapper.v`	Verilog code that connects the register bank circuit to the Handel-C code
`reg32x1k_test.hcc`	Handel-C code that uses the register bank

You can open the files in a text editor such as Notepad. The example also includes DK workspace and project files.

Register bank example: interface code to Verilog

The example defines an interface sort that has port names of the same name as the Verilog signals in the circuit to be integrated. The interface sort must be the same as the Verilog model's name.

The interface from reg32x1k_test.hcc is:

```
interface reg32x1k
        (unsigned 32 data_out)
    registers
        (unsigned 10 address = addressVal
          with {extpath = {registers.data_out}},
        unsigned 32 data_in = data_inVal,
        unsigned 1 ck = __clock,
        unsigned 1 write = writeVal);
```

reg32xlk is the name of the interface sort.

Ports declared by the interface

Port name	Port direction
data_out	in
address	out
data_in	out
ck	out
write	out

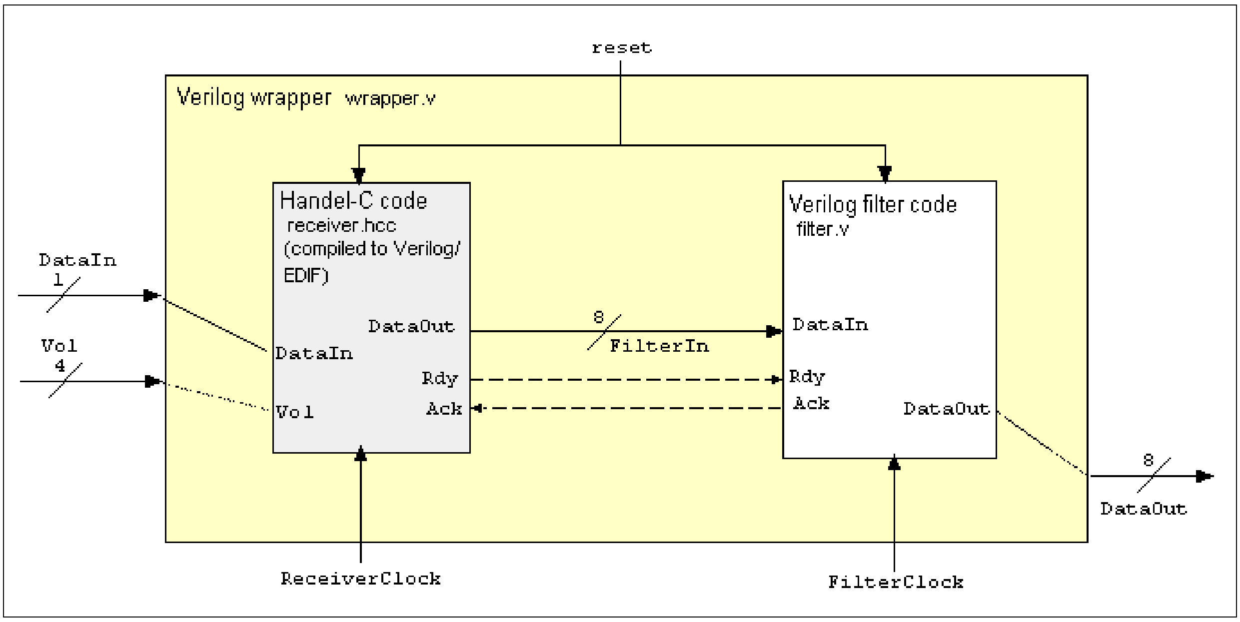
13.6.3 FIR filter example files: Verilog

The FIR filter example (***InstallDir***\DK\Examples\Verilog\Example3) consists of these files:

filter.v	Verilog code that describes the FIR filter
wrapper.v	Verilog code that connects the Verilog filter to the Handel-C receiver
receiver.hcc	Handel-C code that receives a bit stream, performs a volume change on it if required and converts it to 8-bit data

You can open the files in a text editor such as Notepad.

13. Integrating Handel-C with VHDL, Verilog and EDIF



LOGIC BLOCKS IN THE FIR FILTER EXAMPLE

The example also includes DK workspace and project files.

FIR filter example: interface code to Verilog

The Handel-C receiver is a component in a Verilog design. If your top-level code is Verilog, you must use the port_in and port_out interface types to communicate with the Verilog. The interfaces must have port names of the same name and type as the Verilog signals in the wrapper connecting to the Handel-C component to be integrated.

The interfaces between receiver.hcc and the Verilog wrapper are:

```
interface port_in           //interface type must be port_in or port_out
    (unsigned 1 DataIn)     // single bit input port - name used in Verilog
    ReadData                // name of instance of port_in
    ();                     // no output ports

interface port_out          //interface type must be port_in or port_out
    ()                      // no input ports
    WriteData               // name of instance of port_out
    (unsigned 8 DataOut = Bytes_out); //output port (name used in Verilog)

interface port_out           //interface type must be port_in or port_out
    ()                       // no input ports
    WriteRdy                 // name of instance of port_out
    (unsigned 1 Rdy = DataReady);  //name of output signal and its value
```

```
interface port_in          //interface type must be port_in or port_out
    (unsigned 1 Ack)       // single bit input port - name used in Verilog
    ReadAck                // name of instance of port_in
    ();                    // no output ports

interface port_in          //interface type must be port_in or port_out
    (unsigned 4 Vol)       //4 bit wide input port (name used in VHDL)
    Volume                 // name of instance of port_in
    ()                     // no output
    with {std_logic_vector = 1}; //standard logic ports
```

Ports declared by the interfaces

Port name	Port direction
DataIn	in
DataOut	out
Rdy	out
Ack	in
Vol	in

13.7 Example of interfacing to EDIF

An example is supplied of a project interfacing to a toplevel EDIF wrapper file which in turn interfaces to another EDIF module. The example is installed in the subdirectory `DK\Examples\EDIF`.

It consists of an EDIF code file for the circuit, an EDIF wrapper file that links the EDIF to the Handel-C, and a Handel-C file that connects to the EDIF wrapper via `port_in` and `port_out` interfaces.

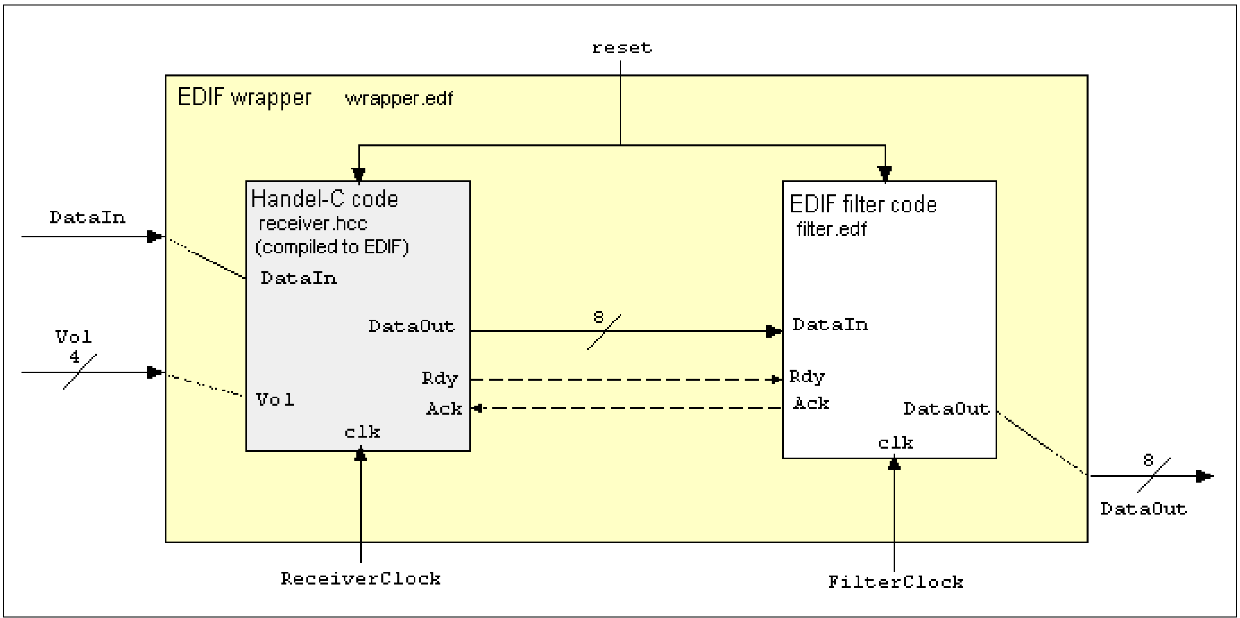
You can only compile this examples if you have the full version of DK.

13.7.1 FIR filter example files: EDIF

The FIR filter example (***InstallDir***`\DK\Examples\EDIF\Example3`) consists of these files:

`filter.edf`	EDIF code that describes the FIR filter
`wrapper.edf`	EDIF code that connects the EDIF filter to the Handel-C receiver
`receiver.hcc`	Handel-C code that receives a bit stream, performs a volume change on it if required and converts it to 8-bit data

You can open the files in a text editor such as Notepad.



LOGIC BLOCKS IN THE FIR FILTER EXAMPLE

FIR filter example: interface code to EDIF

The Handel-C receiver is a component in a EDIF design. If your top-level code is EDIF, you must use the port_in and port_out interface types to communicate with the EDIF. The interfaces must have port names of the same name and type as the EDIF signals in the wrapper connecting to the Handel-C component to be integrated.

The interfaces between receiver.hcc and the EDIF wrapper are:

```
interface port_in       //interface type must be port_in or port_out
    (unsigned 1 rst)    // single bit  reset input port - name used in EDIF
    ResetPort           // name of instance of port_in
    ();                 // no output ports

interface port_in       // interface type must be port_in or port_out
    (unsigned 1 clk)    // single bit clock input port - name used in EDIF
    ClockPort           // name of instance of port_in
    ();                 // no output ports

interface port_in           //interface type must be port_in or port_out
    (unsigned 1 DataIn)     // single bit input port - name used in EDIF
    ReadData                // name of instance of port_in
    ()                      // no output ports
    with {busformat = "B<N:0>"}; // specify an array bus format
```

```
interface port_out          //interface type must be port_in or port_out
    ()                      // no input ports
    WriteData               // name of instance of port_out
    (unsigned 8 DataOut = Bytes_out) //output port (name used in EDIF)
    with {busformat = "B<N:0>"};     //specify an array bus format

interface port_out           //interface type must be port_in or port_out
    ()                       // no input ports
    WriteRdy                 // name of instance of port_out
    (unsigned 1 Rdy = DataReady);    //name of output signal and its value

interface port_in          //interface type must be port_in or port_out
    (unsigned 1 Ack)       // single bit input port - name used in EDIF
    ReadAck                // name of instance of port_in
    ();                    // no output ports
```

Ports declared by the interfaces

Port name	Port direction
DataIn	in
DataOut	out
Rdy	out
Ack	in

14. Utilities

The DK package includes the following utilities.

bmp2raw	converts BMP image files to a format suitable for input to the Handel-C simulator.
raw2bmp	generates BMP image files from a file generated by the Handel-C simulator.

They are located in ***InstallDir***\DK\Examples\Handel-C\ExampleC\Data.

These utilities can handle both raw binary and text file formats. This is useful if a conventional C program requires raw binary input and output whereas the simulator requires text input and output.

The raw data format can be configured to have the colour bits in any order to allow simulation of applications requiring non-standard bit patterns (e.g. 5-6-5 bit RGB format).

Example

For an example of how to use these utilities, see the Edge detector example.

14.1 bmp2raw utility

The bmp2raw utility converts BMP image files into raw binary or text format. The text format is suitable for input into the Handel-C simulator. Files can be converted back to BMP format using the raw2bmp utility.

The utilities are located in ***InstallDir***\DK\Examples\Handel-C\ExampleC\Data.

The general usage of the bmp2raw utility is:

bmp2raw [-b] ***BMPFile RAWFile RGBFile***

where:

BMPFile	is the source image file
RAWFile	is the destination raw data file
RGBFile	is a file describing the format of the pixels in the raw data file

Adding the –b flag as the first command line option causes the utility to generate a raw binary file rather than a text file. To see the difference, consider a file containing the numbers 0 to 3. The text version (no –b option) would look like this:

```
0x00
0x01
0x02
0x03
```

The binary version (created with –b option) would not be visible when loaded into an editor. Instead, a hex dump of the file might look like this:

```
00000000 00 01 02 03 ** ** ** ** ….****
```

The format of the raw data file can be controlled with the *RGBFile* specified on the command line. This tells the utility where to place each colour bit in the words in the raw data file. Internally, the pixels in the BMP file are expanded to 8 bits for each of red, green and blue.

RGBfile description file format

Example *RGBFile*

The description file works by starting counting at bit 7 of the colour specified by the identifier word and works down through the bits of that colour placing each bit in the specified location in the destination word. The destination word will automatically be created wide enough to contain the most significant bit specified (up to 32 bits wide in total).

See the *RGBFile* worked example for an illustration of the following options:

- You need not specify 8 locations for each colour. The least significant bits of each colour will be dropped if fewer than 8 locations are specified.
- You can specify multiple identifiers of the same colour. The bit counter will continue to count down from the value reached for that colour each time you specify the colour again.

14.1.1 RGBFile *example*

There is an example file `8BPPdest.rgb` provided with the `bmp2raw` utility to perform a common conversion.

It can be used to extract the red component from source image and generate an 8-bit per pixel raw image. This is useful for greyscale images.

```
8BPPdest.rgb
red
7
6
5
4
3
2
1
0
green
blue
```

14.1.2 bmp2raw RGBFile example

`8BPPDest.rgb` is an example file provided with the bmp2raw utility in ***InstallDir***`\DK\Examples\Handel-C\ExampleC\Data`. It extracts the red component from source images and generates an 8-bit per pixel raw image. This is useful for greyscale images. You can examine the file by opening it in Notepad.

14.1.3 bmp2raw RGB description file format

Red

Location for bit 7 of red
Location for bit 6 of red
Location for bit 5 of red
Location for bit 4 of red
Location for bit 3 of red
Location for bit 2 of red
Location for bit 1 of red
Location for bit 0 of red

Green

Location for bit 7 of green
Location for bit 6 of green
Location for bit 5 of green
Location for bit 4 of green
Location for bit 3 of green
Location for bit 2 of green
Location for bit 1 of green
Location for bit 0 of green

Blue

Location for bit 7 of blue
Location for bit 6 of blue
Location for bit 5 of blue
Location for bit 4 of blue
Location for bit 3 of blue
Location for bit 2 of blue
Location for bit 1 of blue
Location for bit 0 of blue

14.2 raw2bmp utility

The `raw2bmp` utility is the reverse of the `bmp2raw` utility. It converts raw text or binary files to BMP image files. The main use of the `raw2bmp` utility is to allow viewing of the output from image processing applications with the standard Windows Paint utilities.

The `raw2bmp` utility is located in ***InstallDir***`\DK\Examples\Handel-C\ExampleC\Data`.

The general usage of the `raw2bmp` utility is as follows:

`raw2bmp [-b]` ***`Width RAWFile BMPFile RGBFile`***

`Width`	the width of the image. The height will be calculated from this parameter and the source file length.
`RAWFile`	source file containing raw data.
`BMPFile`	destination image file.
`RGBFile`	file describing the format of the pixels in the raw data file.

Adding the -b flag as the first command line option causes the utility to read a raw binary file rather than a text file.

14.2.1 RGBFile *worked example*

In the *RGBFile* description file you need not specify 8 locations for each colour. The least significant bits of each colour will be dropped if fewer than 8 locations are specified. In the example below, the least significant 6 bits of red and blue and the least significant 4 bits of green are dropped.

To generate 8-bit pixels in the raw file with the following bit pattern:

Raw file bit number	Colour bit	
(Most significant) 7	7	Red
6	7	Green
5	7	Blue
4	6	Blue
3	6	Green
2	6	Red
1	5	Green
(Least significant) 0	4	Green

use the following RGBFile:

```
Red
7
2
Green
6
3
1
0
Blue
5
4
```

Each pixel number and identifier (`Red`, `Green` or `Blue`) must appear on a separate line.

You may also specify multiple identifiers of the same colour. The bit counter will continue to count down from the value reached for that colour each time you specify the colour again. For example, the above file could also be written like this:

```
Red
7
Green
6
Blue
5
Red
2
Green
3
1
Blue
4
Green
0
```

14.2.2 RGBFile format

With the raw2bmp utility the format of the *RGBFile* describing where each bit is located in the raw data word is similar to the file used by the `bmp2raw` utility. Indeed, for some pixel formats (such as in the *RGBFile* worked example) a common file may be used.

As an example of where a different file may be required, consider the conversion of 8 bit per pixel greyscale images to a BMP image. Here, each bit must be duplicated in the red, green and blue components of the destination BMP file.

For example:

```
red
7
6
5
4
3
2
1
0
green
7
6
5
4
3
2
1
0
blue
7
6
5
4
3
2
1
0
```

14.2.3 raw2bmp RGBFile example

`8BPPsrc.rgb` is an example file provided with the raw2bmp utility in ***`InstallDir`***`\DK\Examples\Handel-C\ExampleC\Data`. It duplicates each bit of an 8-bit per pixel raw file to red, green and blue components. You can examine the file by opening it in Notepad.

15. Troubleshooting

My plugins don't work any more

The names of the plugins supplied with DK to connect Handel-C simulations together have changed. You need to update any references to these in your code. The files affected are: `DK1Connect.dll`, `DK1Share.dll`, and `DK1Sync.dll`. These have been renamed to `DKConnect.dll`, `DKShare.dll`, and `DKSync.dll`.

The previous simulator (netlist simulator) supported undocumented features, such as the API functions `HCPLUGIN_GET_VALUE_COUNT_FUNC` and `HCPLUGIN_GET_VALUE_FUNC`. It also supported the `HCPLUGIN_VALUE` data structure. These are not supported by the new simulator. If you have used them, you will have to simulate code containing these functions with the old simulator.

Values can now be passed to and from Handel-C by calling parameterized C or C++ functions from Handel-C and Handel-C functions from C or C++.

DK library functions/macros don't work any more

DK libraries are now only supplied with the file extensions `.hch` (header) and `.hcl` (library file). You may need to update references to them in your code.

The standard macro library (`stdlib.hch`) and fixed-point library (`fixed.hch`) now form part of the Platform Developer's Kit. If you have used macros from these libraries you will need to update references to them on the **Linker** tab in Project Settings or the **Directories** tab in Tool options.

My variables have weird values and they used to be fine

With previous versions of the compiler, some local non-static variables may have defaulted to 0.

You must now explicitly assign local non-static variables to zero (or some other value); their default initial value is undefined.

My code is too large/too slow

Use the results of the logic estimator to pinpoint areas of your code using the most resources.

Look at the application notes and other resources on the Celoxica Web site.

I don't understand the error messages

Look at the Error message descriptions in the DK User Guide.

I need more information

Look at the Celoxica technical library at:
`http://www.celoxica.com/technical_library/default.asp`

15. Troubleshooting

15.1 Error messages

Most error messages are relatively intuitive. Some of the less obvious ones will be due to system problems, such as files being corrupted, unavailable or in the wrong format, or the system not having enough disk space to write to a file.

Some of the error messages are listed below in alphabetical order with a brief explanation.

The simulator also forwards errors from plugins that have been written using the Plugin API.

Compiler and simulator error messages

`"Arithmetic operations are not permitted on a 'void' pointer"`

You cannot perform arithmetic on a void pointer because the size of the object being pointed to is not known. For example:

```
void *p;
++p; // not allowed
```

`"Assignment loses 'const' qualifier"`

You cannot perform assignments that would potentially allow modification of data qualified as `const` . For example:

```
{
const int 4 ci = 5;
const int 4 * ptr_ci;
int 4 * ptr_i;
ptr_ci = & ci;
ptr_i = ptr_ci; //banned assignment; if this were allowed...
* ptr_i = 3; //...then this would change the value of ci
}
```

`"At least one pulse specified by 'string' crosses Handel-C clock cycle boundary"`

You have specified a clock pulse length for the RAM clock which does not lie inside a Handel-C clock cycle. Either the `clkpulselen` is too large, or you have offset it too much.

`"Attempt to access partial struct/union 'string'"`

Struct or union not fully defined. E.g.

```
struct S;
S x;
x.Bill; without the definition
struct S
{
   int Bill;
};
```

`"Bi-directional interfaces using the 'string' standard are not supported by current family"`

There is a list of the I/O standards supported by different devices in the Handel-C Language Reference.

`"Call to recursive function 'string'. (Not supported by Handel-C)"`

Functions cannot be recursive in Handel-C. Use macro procedures or macro expressions instead.

`"Cannot compile object - not all information is known"`

Could not infer a width or type etc. E.g. `int undefined x;`

`"Cannot have a 'shared expr' of this type"`

You may only use integral types, pointers and aggregates as the return type for a shared expr.

`"Cannot initialize 'ports' memory"`

You cannot initialize a memory where the ports specification is non-zero. For example:

```
ram raz[2] = {1, 2} with {ports = 1}; //illegal
```

`"Cannot target EDIF - not all information is known"`

Could not infer a width or type etc. E.g. `int undefined x;`

`"Cannot target RTL level Verilog - not all information is known"`

Could not infer a width or type etc. E.g. `int undefined x;`

`"Cannot target RTL level VHDL - not all information is known"`

Could not infer a width or type etc. E.g. `int undefined x;`

`"Cannot target simulator - not all information is known"`

Could not infer a width or type etc. E.g. `int undefined x;`

`"Cast loses 'const' qualifier"`

You cannot perform type conversions that would potentially allow modification of data qualified as const. For example,

```
{
   const int 4 ci = 5;
   const int 4 * ptr_ci;
   int 4 * ptr_i;
   ptr_ci = & ci;
   ptr_i = (int 4 *) ptr_ci; //banned cast; if allowed...
   * ptr_i = 3; //...then this would change the value of ci
}
```

`" 'chanin' is only supported in simulation target"`

chanin and chanout are used to create channels when simulating buses. (The DK simulator cannot determine when input and output should occur when simulating buses.)

15. Troubleshooting

`" 'chanout' is only supported in simulation target"`

chanin and chanout are used to create channels when simulating buses. (The DK simulator cannot determine when input and output should occur when simulating buses.)

`" '-cl' option specified without '-s' option"`

The -cl option is used when targeting the simulator (-s) via the command line compiler.

`" 'const' or 'volatile' qualifier cannot be used on a channel. Move qualifier to underlying type?"`

You cannot define a const channel. However, the channel could have a const type.

```
const chan <int 8> x; //not allowed

chan <const int 8> x; //OK
```

`" 'const' or 'volatile' qualifier cannot be used on a signal. Move qualifier to underlying type?"`

You cannot define a const signal. However, the signal could have a const type.

```
const signal <int 8> x; //not allowed

signal <const int 8> x; //OK
```

`"Could not check out licence for ... Please check installation of FlexLM."`

Check the details of the floating licence file; you may not be licensed for certain Handel-C HDL outputs. The location of the floating licence file is set by the environment variable `LM_LICENSE_FILE`.

`"Could not create temporary file"`

Your hard disk may be full, or there may already be a read-only file of the same name.

`"Could not determine which clock to use for ''string''.`

An object requiring a clock was built but the compiler couldn't work out which clock it should be connected to. Probably caused by an unused object (the compiler finds clocks from an object's use and not its declaration).

`"Could not expand 'typeof'"`

You are using typeof on an object of unknown type.

`"Could not infer information about this object"`

You may have declared a pointer of unknown width and not used it, declared a variable of unknown width and then never used it in a context where the compiler could infer the width.

`"Could not infer width of enumerated type"`

Probably due to defining an enum that is never used.

`"Design contains an unbreakable combinational cycle"`

The Handel-C compiler tries to break combinational code loops by inserting delay statements. It is better to do this explicitly. For example,

```
while (x!=3)
{
   delay;
}
```

`"Error while compiling simulation output ('string')"`

The back end simulation compiler (e.g. VC++) failed to compile the simulation output. (E.g. not enough disk space, could not find file, illegal option specified in -cl, internal compiler error etc.).

`"External tool not found (preprocessor or backend C compiler not in path)"`

Error when the compiler cannot run the C preprocessor or the C compiler used to compile the simulation .dll.

`" 'extern "C" ' and 'extern "C++" ' not supported for EDIF, VHDL or Verilog output"`

You can only link to C or C++ code when building for Release or Debug.

`"'extern "C" ' and 'extern "C++" ' not supported for netlist simulator"`

You can only link to C or C++ code when using the new (fast) simulator. Check that **Use netlist simulator** is not ticked on the Linker tab in Project Settings.

`"Evaluation of ... is not supported"`

The expression evaluator in the Watch window cannot display expressions containing function calls, let, select, trysema, strings, & or assert.

`"Handel-C does not support side effects in expressions"`

Expressions cannot take any clock cycles in Handel-C. For example, if (a<b++) is not permitted because the ++ operator has the side effect of assigning b+1 to b which requires one clock cycle.

`"Illegal function declaration"`

You may have missed the parentheses from your function declaration.

`"Illegal 'macro proc' expansion"`

You have probably used a macro proc instead of a macro expr.

`"Illegal ports for technology primitive 'string'"`

You have the wrong number of ports, or the ports are of the wrong width. For example, you could have declared two output ports for an AND primitive.

`"Illegal ports on standard bus type"`

A built-in interface sort has been declared with the wrong number of input and/or output ports. For instance, a bus_in may only have one input port, a bus_out may only have one output port, a bus_ts may only have one input port, one output port and one tristate condition, and so on.

`"Illegal return type for a function."`

You can only return integers, structs or arrays from a function.

15. Troubleshooting

`"Illegal right hand side for '&' operator"`

You have tried to find the address of something without an address (e.g., a constant).

`"Illegal type for off-chip memory"`

You have attempted to store an architectural type or a structure in an off-chip memory.

`"Illegal use of identifier 'string'"`

Probably caused by using a typedef name as a variable.

`"Illegal use of 'releasesema()' "`

Missing `trysema()` statement.

`"Illegal value for 'base' spec (defaulting to base 10)"`

base specification not 2, 8, 10 or 16.

`"Integer used as a pointer must be zero"`

Probably caused by casting or comparing a constant to a pointer. You can only do so with 0 (the null pointer) e.g. `(int *)0;`.

`"Invalid input file"`

infile in wrong format.

`"IO standard selection ('standard' spec) is not supported for clock sources not assigned to dedicated clock inputs ('clockport' spec)"`

In some Xilinx devices, you can only specify I/O standards for clocks on dedicated clock input pins. These pins are chosen by default by the DK compiler, but you can disable this by setting the clockport specification to zero.

`"'macro expr' declarations have differing parameters"`

Prototype and declaration vary in number of parameters.

`"'macro proc declarations have differing parameters"`

Prototype and declaration vary in number of parameters.

`"Memory cannot be declared as both 'offchip' and 'ports'"`

Caused by declaring memory as off-chip with the offchip specification and declaring it as on-chip in foreign code using the ports specification.

`"Memory forms do not match"`

Caused by comparing two types of memory (e.g. one is ram int x[1] and the other is rom int y[1])

`"No more than one endpoint of a cross clock domain channel communication may be a prialt statement"`

If you have a channel communicating between two clock domains, you can only have a prialt on one side.

`"Object cannot be stored in ram/rom/wom memory"`

You have attempted to store an architectural type in a memory.

15. Troubleshooting

`"Pin 'string' feeds multiple sequential blocks, which may lead to unexpected  behaviour. Consider using a clocked interface"`

You have an input which is not synchronized with the Handel-C clock which is feeding multiple blocks. The values in the blocks may be different on the same clock cycle. For example, if it is feeding two flip-flops, if the first flip-flop is updated before the clock cycle and the second afterwards, both flip-flops can be read after the first clock cycle but only one will have been updated. To prevent this behaviour, use a clocked interface or bus.

`"Pointer offset or array index must be integral"`

Indices to arrays and offsets to pointers must be expressions of integral type. They cannot be types, or non-integral expressions. For example:

```
struct MyStruct s;
int *p;
int i [4];
*(p + s); // not allowed
i [s]; // not allowed
```

`"Port 'string' appears more than once in design (port_in or port_out with no identifier?)"`

You have two ports declared with the same name (or possibly without a name).

`"Port 'string' appears more than once in external component declaration"`

You have two ports declared with the same name in the same interface.

`"Receive from channel in more than one clock domain"`

Channels that connect between clock domains must be unidirectional.

`"Send to channel in more than one clock domain"`

Channels that connect between clock domains must be unidirectional.

`"'shared expr' declarations have differing parameters"`

Prototype and declaration vary in number of parameters.

`"Simulator is not responding. Terminate simulation process?"`

This message appears on a dialog if you use the **Break** or **Stop Debugging** commands when stepping through C/C++ code in DK. Select Yes to stop the simulation.

`"Source code contains preprocessor statements"`

There still appear to be pre-processor statements in your code after pre-processing (maybe be caused by unrecognized #statement).

`"Syntax error"`

Syntax error in source code.

`"Timing constraints specified using the 'string' spec may not be zero"`

You cannot have a bus_in interface with the intime specification set to zero, or a bus_out interface with the outtime specification set to zero, as signals cannot be passed in or out in zero time.

`"Unknown specification identifier - 'string'"`

Unknown object specification identifier (with `{`***`spec_identifier`*** `= ...}`)

`"Unsupported family: 'string' "`

Check whether your device is supported by DK. See the Summary of supported devices in the Handel-C Language reference.

`"Unterminated string constant"`

Missing closing quotes.

`"Un-supported synthesis tool: 'string'"`

Check the list of supported VHDL/Verilog synthesis tools in the DK User Guide.

`"Variable 'string' is used from more than one clock domain"`

Data must be passed to different clock domains using a channel or an interface. Variables cannot be shared between clock domains

`"'with' cannot be used on a declaration"`

Object specifications (e.g. `with {busformat = "B[N:0]"}`) can only be applied to definitions of objects, not to declarations.

`"'with' on anonymous declaration is not permitted"`

You have used 'with' on a declaration with no name e.g. `struct s{} with {show = 1}`

15.1.1 DK environment error messages

```
"DK cannot continue with Find in Files.
Details:"
```

File could not be opened or read.

```
"DK design suite could not insert the project file in to the workspace.
Details:"
```

File could not be opened or read.

`"DK design suite could not load the browse-info database file "`

File could not be opened or read.

```
"DK design suite could not start the simulator.
Details:"
```

File could not be opened or read.

`"None of the simulator DLLs have any clocks defined."`

You have no main programs associated with clocks in your compiled code.

`"The simulator '`***`string`***`' does not have any clocks defined."`

You have built a function with no clock and attempted to simulate it. You should have a clocked main function that interfaces to the unclocked function.

`"The symbol '`***`string`***`' is not defined."`

The cursor is not on a known symbol or a symbol has not been selected in the file.

`"There is no browse information for the project` ***`string`***`."`

You did not have **Save browse info** selected when you compiled the file.

15.2 Warning messages

Most warning messages are relatively intuitive. Some of the less obvious ones are listed in alphabetical order with a brief explanation. Some of the error messages are also described in the DK User Guide.

`"'base' not supported on aggregate members - ignoring"`

You can only apply the base specification to the whole of a `struct`, `interface` or `mpram`, not to the individual elements.

`"Breaking combinational cycle (continue statement) - may alter timing"`

The Handel-C compiler tries to break combinational code loops. It is better to do this explicitly, e.g. by inserting a `delay` statement.

`"Cannot open delay file. Timing estimation will revert to logic levels"`

The logic estimator uses a file (`DelayFile.hcd`) for storing delays through logic elements for different devices. This file may be corrupt or missing.

`"Current FPGA family not supported by technology mapper"`

A list of the devices supported for technology mapping is given in the DK User Guide.

`"Data specs ignored for EDIF bus '`***`string`***`' "`

If you use a data specification and a busformat specification, the data specification will be ignored.

`"Declaration of '`***`string`***`' shadows function parameter"`

A local variable has the same name as a function parameter.

`"Function '`***`string`***`' may be recursive"`

Functions can not be recursive in Handel-C. Use macro procedures or macro expressions instead.

`"Illegal character in input (ignored)"`

Likely to be due to a non-ASCII character in an input file.

`"IO standard selection ('standard' specification) not supported for HDL output - ignored"`

You can only use the standard specification (e.g. with `{standard = "HSTL_III"}`) for EDIF output.

`"Netlist expansion 'for area' not supported for current device family - performing default expansion"`

The `-N+area` option optimizes arithmetic hardware for size in Actel devices. It is not supported for other devices. If you are using the GUI, and are targeting EDIF output, check that the **Expand Netlist for** option on the **Compiler** tab in **Project Settings** is set to **Speed**, not **Area**.

15. Troubleshooting

"Passing Handel-C type through '...' - cannot automatically check types"

If you are using extern "C" or extern "C++" and use an ellipsis in the function declaration, DK cannot perform type checking. For example:

```
extern "C" int printf(const char *format, ...);
// no type checking
```

"Properties specification on black box interface '***string***' is ignored"

You can only use the properties specification for VHDL or Verilog output if you have set the bind specification to 1.

"Property specs ignored for VHDL"

You can only use the properties specification for EDIF output.

"Property specs ignored for Verilog"

You can only use the properties specification for EDIF output.

"Pulse position list for spec '***string***' is empty - memory will not be clocked during %s cycles"

If you have used the rclkpos or wclkpos specifications with empty lists, memory will not be clocked during the read clock or write clock cycles. For example, memory will not be clocked during the write clock cycle if you use the code below:

```
mpram
{
   rom int 1 ro[16] with {rclkpos = {1}, clkpulselen = 0.5};
   wom int 1 wo[16] with {wclkpos = {}, clkpulselen = 0.5};
}MyMpram;
```

"Sharing pin '***string***' between tri-state buses - possible enable conflicts"

Do not enable both sources at the same time; this could lead to hardware damage.

"'***string***' specification not supported on interfaces using differential IO standards - ignored"

If you have used a specification which conflicts with the use of a differential I/O standard, it will be ignored. For example:

```
interface bus_in (unsigned 2 datain) I() with {standard =
"LVDS25",
data = {"P1", "P2"}, {"P3", "P4"}, strength = 2} // strength spec
will be ignored
```

"Timing constraint ('***string***' spec) not supported on ports by P+R for current device family - ignored"

You cannot use intime on port_in interfaces, or outtime on port_out interfaces for some device types.

"Timing constraint ('%s' spec) not supported on generic interface ports by P+R for current device family - ignored"

You cannot use intime or outtime on generic interfaces for some device types.

`"'True' dual-port mode not supported by Stratix M512 blocks - setting block type to AUTO"`

You can only use dual-port RAMs with one ROM port and one WOM port in M512 blocks. If you want to use an MPRAM with two RAMs, target the 4K or MRAM instead.

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Customer Support at http://www.celoxica.com/support/

Celoxica in Europe

T: +44 (0) 1235 863 656

E: sales.emea@celoxica.com

Celoxica in Japan

T: +81 (0) 45 331 0218

E: sales.japan@celoxica.com

Celoxica in Asia Pacific

T: +65 6896 4838

E: sales.apac@celoxica.com

Celoxica in the Americas

T: +1 800 570 7004

E: sales.america@celoxica.com