

Content Processing Development Kit

CP-DK User Manual

For CP-DK v1.0

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.

Neither the whole nor any part of the information contained in, or the product described in, this document may be adapted or reproduced in any material form except with the prior written permission of the copyright holder.

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.

Copyright © 2002 Celoxica Limited. All rights reserved.

Authors: MDA, SPGC

Document number: CP-DK-User-1.0

Table of contents

Table of contents

TABLE OF CONTENTS	3
ASSUMPTIONS	5
OMISSIONS	5
1. INTRODUCTION	6
2. REQUIREMENTS	7
3. DESIGN FLOW	8
3.1 HARDWARE DESIGN FLOW	8
3.2 HW-SW DEVELOPMENT FRAMEWORK STACK	9
4. CPP BOARD FEATURES	10
5. BOARD SUPPORT PACKAGE	11
5.1 HEADER, LIBRARY AND SOURCE FILES	11
5.1.1 Handel-C	11
5.1.2 C / C++	12
6. BSP USAGE AND API REFERENCE	13
6.1 LOCAL DEVICE ACCESS	13
6.1.1 SRAMs and LEDs	13
6.1.2 Simulation	14
6.2 AGENT BUS INTERFACE (ABI)	14
6.2.1 DDR memory access	14
6.2.2 DDR memory simulation	16
6.2.3 Host I/O	16
6.2.4 Host I/O simulation	17
6.3 STANDARD AGENT INTERFACE (SAI)	17
6.3.1 Host I/O	17
6.3.2 SAI Simulation	19
7. USING THE CP-DK	20
7.1 SETTING UP A PROJECT IN DK	20
7.2 DESIGNING AN AGENT USING CP-DK	20
8. EXAMPLES	22
8.1 PURPOSE OF THE EXAMPLES	22
8.2 REQUIREMENTS	22
8.3 BUILDING AND RUNNING THE EXAMPLES	22
8.4 LED TEST	22

Table of contents

8.5 SRAM ACCESS	23
8.6 DDR ACCESS	23
8.7 Host I/O	23
8.8 INTERRUPTS	23

Assumptions

Assumptions

This manual assumes that you:

- Intend to use the Content Processing Development Kit to develop software and hardware components of the Content Processing Development Platform.
- Are familiar with software programming techniques, C and C++.
- Are familiar with Celoxica DK and Handel-C or have access to the DK and Handel-C manuals.

Omissions

This manual does not include:

- Description of the Handel-C language or usage of the DK design environment.
- Specific instruction on the development environments and drivers for software. These are provided in the *CP-DK Agent Driver Development* and *SAI Reference Guides*.
- Detailed information on the CPP hardware platform or installation instructions. These are provided in the *CPP Install Guide*.

1. Introduction

This manual describes the Content Processing Development Kit (CP-DK) for the Content Processing Platform (CPP). The CPP is a PCI card incorporating reconfigurable logic components, namely Field Programmable Gate Arrays (FPGAs). The card is intended for use as a co-processor in a CPU host system, such as a Windows or Linux Server or a Linux Cluster Node. The platform is intended to enable:

- **Algorithm Acceleration:** by exploiting the parallelism in algorithms to increase performance with implementation in custom (parallel) hardware i.e. FPGA's.
- **Algorithm Offload:** by transferring computations to the coprocessor to free CPU resource.

The CP-DK includes components for both *hardware* development, namely the content of the FPGA's on the CPP and *software* development, namely the application and drivers on the host system. In summary these components are:

Hardware CP-DK

- Celoxica DK and supporting libraries allowing:
 - Development of the FPGA content ("hardware") on the CPP platform
 - Co-design of software and hardware components of the system

Software CP-DK

- Software libraries API's and drivers allowing
 - Development of host programs ("software") and associated drivers that target the CPP

This manual describes the *Hardware CP-DK* and is organized as follows. An overview of the design flow is given with requirements for use. The features of the Content Processing Platform are summarized in order to introduce the features of the supporting library and hardware API. Finally, a guide to usage of DK and the library is given with examples.

2. Requirements

- The DK Design Suite (v1.1 SP1 or later) and CP-DK libraries (v1.0 or later)
- An IBM-compatible PC, with Windows 2000 or XP for development
- A C++ compiler (see DK requirements for version numbers)
 - Microsoft Visual C++
 - GNU GCC
 - Borland C++
- FPGA Place and Route tools
 - Xilinx ISE 5.1.02i (or later version)
- CPP PCI Card

3. Design Flow

The hardware and software co-development flow using CP-DK is summarized in figure 1. The flow allows developers to use a common methodology and similar languages for both the hardware and software components in a host system containing a CPP. In particular the flow provides:

- **A minimal tool chain:** Celoxica DK design suite, C/C++ compiler and Xilinx Place and Route tools.
- **A common language base:** – C and Handel-C.
- **API's for common interfacing and platform abstraction:** Simplifying application development

The software-side development flow and support are described in detail elsewhere. Here we note that the software application is developed using drivers that manage

- Configuration of the CPP (i.e. FPGA's)
- Transfer of data between the host and CPP

3.1 Hardware Design Flow

The CPP platform may be installed on either a Windows or Linux host. However, developing FPGA content using DK requires a Windows machine. Like the software development flow, the design of Handel-C applications is to a well-defined API where the API functions are contained in a library or Board Support Package (BSP). In the

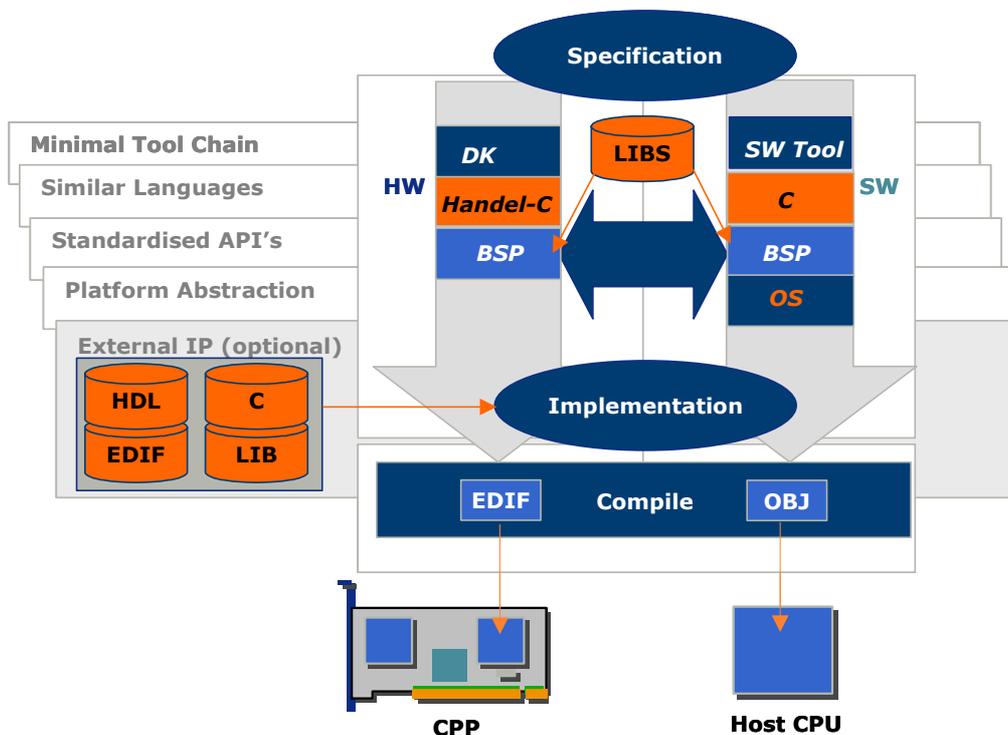


Fig 1. Development Flow

hardware case, however, there is no operating system (OS) on the FPGA. The Handel-C is implemented directly as *combinational logic*. DK transforms the Handel-C *program* into a netlist graph (in EDIF format) that describes the logic components and the connections between them. This process is known as *synthesis*. The EDIF representation is used as input to FPGA vendor *place and route* tools. These generate a configuration bitstream ("bitfile") used to program the FPGA. In the current flow the bitstream is converted to an "Agent file", using a custom utility. The Agent file stores additional information used by the platform software drivers and utilities to configure the FPGA's on the CPP board.

3.2 HW-SW Development Framework Stack

The hardware-software interface for agents is defined in a hardware Board Support Package (BSP) and a software Agent Device Driver (ADD) interface. The software device driver is separated into two levels, a Base Driver which is interfaced by a number of unique ADDs which manage the configuration and communication with the Agents on the CPP board. The hardware BSP contains support for communication with the host, and supports both in-band and out-of-band communication.

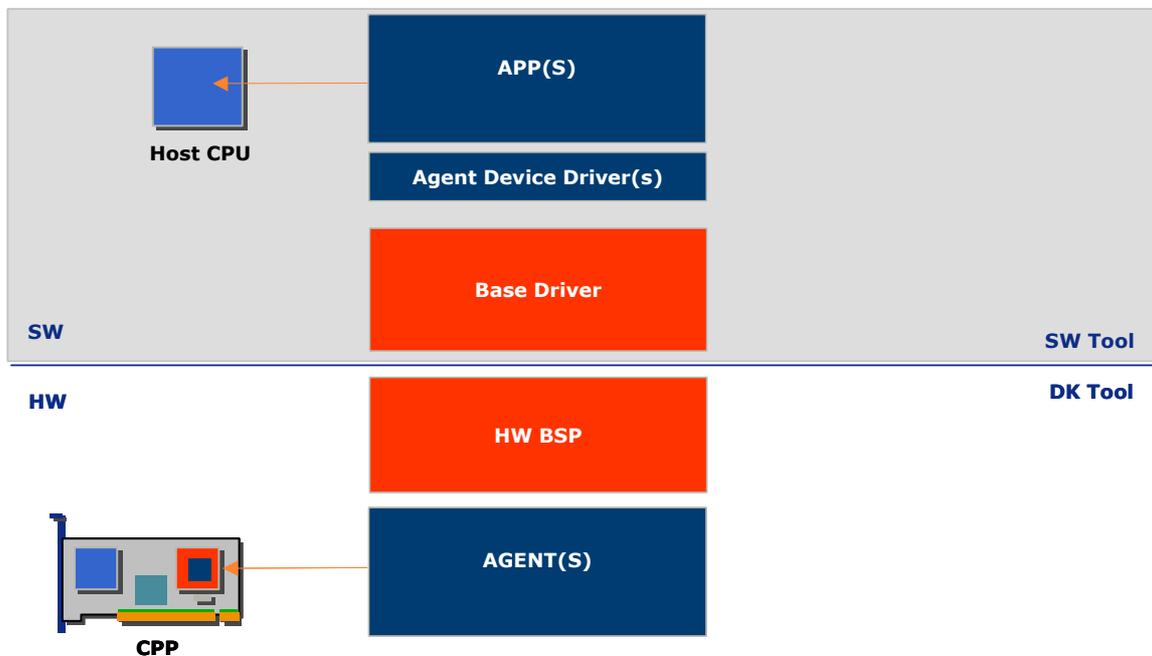


Fig 2. Development Framework Stack

4. CPP board features

This section gives an overview of the CPP board (Figure 2), as an aid to understanding the operation of the Hardware BSP. The CPP has two Virtex-II 1000 FPGAs for application use, known as "Content Processing Engines". Each is connected to

- Two 1MB banks of 18-bit SRAM,
- Eight LEDs
- An auxiliary connector.
- A "Content Processing Controller", or CPC that contains the interface to the PCI bus, and access to 256MB DDR SDRAM.

The entire system is clocked at 133 MHz.

The software side of the system is used to interface via drivers across the PCI bus. This allows the software side to access the DDR SDRAM and the CPEs directly both via the CPC.

In the design of the software drivers for the system it is assumed that each FPGA can contain either one or two hardware acceleration "Agents" at any one time. Communication with these Agents may be via one of two methods:

- Agent Bus Interface (ABI)
 - Direct out-of-band and interrupt communication between Host and FPGA via PCI bus.
 - In-band communication between Host and FPGA via DDR memory accesses.
- Standard Agent Interface (SAI)
 - FIFO-like data transfer between host and Agent suitable for streaming applications.

Similarly the BSP for the hardware side of the system allows FPGA content (i.e. the "Agents" or "Handel-C Applications") to be developed using either the ABI or SAI.

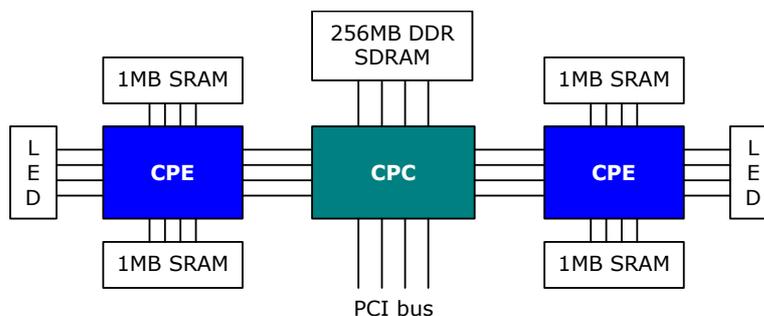


Fig 3. Basic CPP Architecture.

5. Board Support Package

The BSP consists of two major components: local device access (LEDs and SRAM and host IO/DDR access using either the ABI or SAI. When using the ABI two Agents may be implemented in each FPGA and operate in parallel. When using the SAI only one Agent may be implemented in each FPGA. **For CP-DK v1.0, the SAI interface should be treated as experimental and unsupported.** Note that either the SAI or the ABI may be implemented and not both. The next section describes the BSP functions both for implementation and simulation. The block diagram in Figure 4 illustrates the organisation of the Board support package

5.1 Header, library and source files

The CP-DK includes library and header files for building projects for Debug and EDIF, and source code for simulation functions.

5.1.1 Handel-C

The Handel-C libraries are in the directory `/hardware/lib/`. The appropriate library should be specified in the Linker settings for the project in DK. A Handel-C header file is located in the directory `/hardware/include/`. The `BSP.hcl` and `SAI.hcl` are for use with hardware

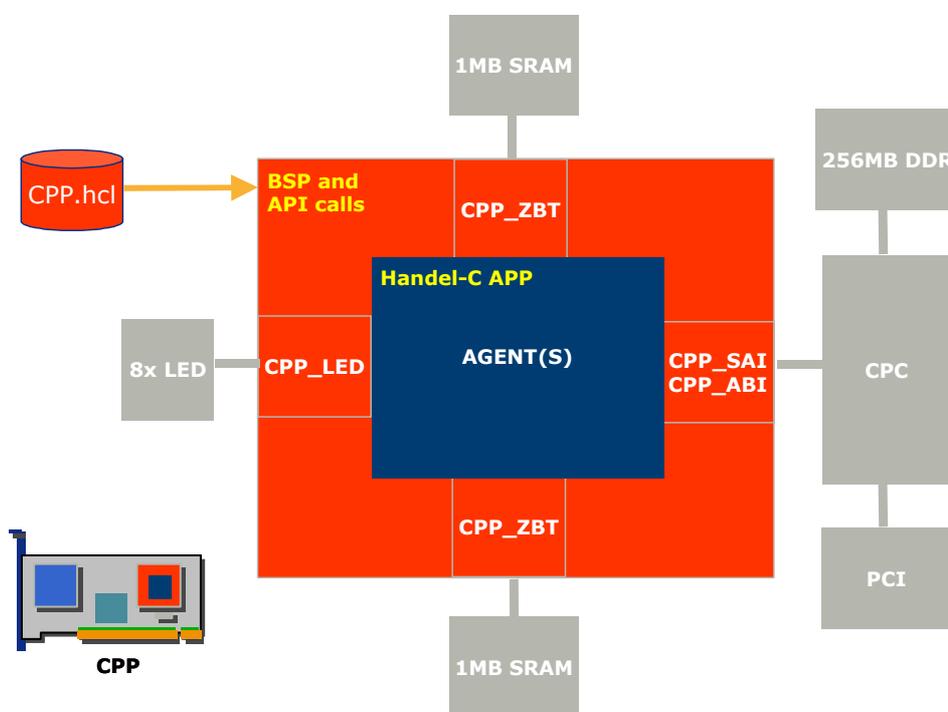


Fig 4. Organisation of the Board Support Package.

targets. The `BSP_sim.hcl` and `SAI_sim.hcl` are for use with simulation targets.

5.1.2 C / C++

A project built for Debug using CP-DK must include the object file `cpp_sim.obj` which is located in `/hardware/lib`. This contains the pre-compiled C++ functions for simulating the CPP. If the use wishes to examine the source for these functions, the relevant files are located in `/software/windows/source/`.

The supplied BSP contains simple debugging information which hooks into the DK debugger. The is scope for third parties to develop a full "virtual platform" debugging environment to stimulate the DK simulator (Figure 5).

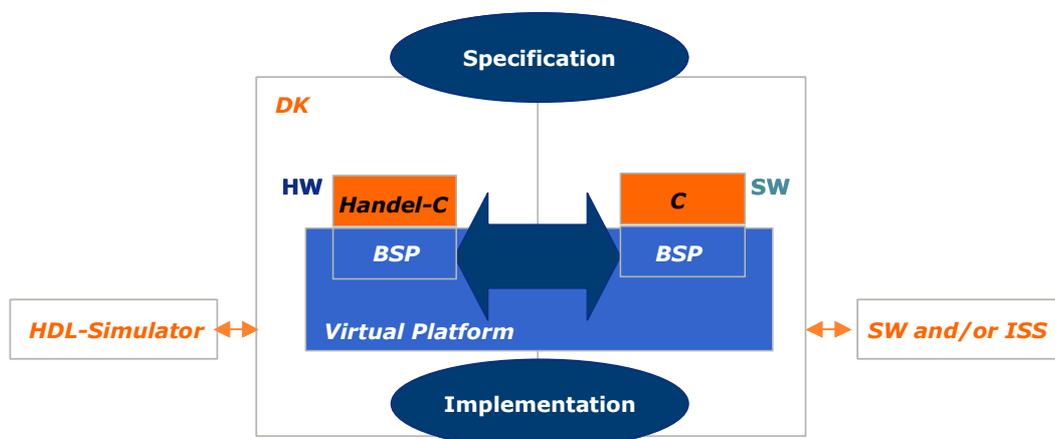


Fig 5. API's enable rapid co-verification.

6. BSP Usage and API Reference

6.1 Local device access

6.1.1 SRAMs and LEDs

Each Agent FPGA on the CPP has two banks of SRAM, each organised as 18 bits x 524288 words, and eight single LEDs. The type of SRAM is ZBT (or NT-RAM), meaning that reads and writes can be performed back-to-back without any intermediate delays. The SRAM operates in PL2 pipelined mode, so the data lags two clock cycles behind the address for both read and write operations. This means that to maintain maximum throughput a new address should be set every cycle, and also data read or written every cycle, but with a startup delay of two clock cycles. Described below are the procedures provided by the BSP for accessing the LEDs and SRAM:

`CPP_SetLEDs(Data)`

Pass an unsigned 8 bit data item to this procedure, and it will be written to the bank of 8 LEDs. The procedure takes one clock cycle to execute, and the data passed will be latched and remain displayed on the LEDs until the procedure is called again.

`CPP_zbt#_set_address(Address, ReadOrWrite)`

where # is either 0 or 1

These macros are used to request a single word to be read from or written to SRAM bank 0 or 1. They must be called for every word, there is no auto-advance of address supported. Address is a 19-bit variable, and ReadOrWrite should be set equal to `zbt_read` or `zbt_write`, which themselves are defined as being equal to 1 and 0. These macros execute in one clock cycle, and the relevant `read_data` or `write_data` procedure should be called two clock cycles later to complete the transaction.

`CPP_zbt#_read_data(Data)`

where # is either 0 or 1

These macros read data from SRAM bank 0 or 1. The `set_address` procedure for the relevant SRAM bank must have been called two cycles earlier with the ReadOrWrite parameter set to `zbt_read`. These procedures execute in a single cycle, writing the 18-bit data from the SRAM to the Data parameter.

`CPP_zbt#_write_data(Data)`

where # is either 0 or 1

These macros write data to SRAM bank 0 or 1. The `set_address` procedure for the relevant SRAM bank must have been called two cycles earlier with the ReadOrWrite parameter set to `zbt_write`. These procedures execute in a single cycle, writing the 18-bit Data parameter to the SRAM.

6.1.2 Simulation

The LEDs are not visibly simulated, but their state can be observed through the value of the `bsp_LED_interface` variable, which is displayed in the "locals" debug window of DK, or can be viewed in a "watch" window by entering its name.

The SRAM is internally simulated in the BSP, including the pipelined access to it. The normal SRAM access procedures can still be used, although users must call `CPPSramDriver()` in parallel with any other code at the top level of their main program. This driver simulates the pipelined nature of the physical SRAM, and can also be called in non-simulation builds of an Agent, although it will have no effect there. Having called the driver, all other SRAM access procedures will behave as in the hardware version.

6.2 Agent Bus Interface (ABI)

The ABI provides access to several methods of communication with the host:

- **Interrupts.** The Agent can send an interrupt to the host computer.
- **Direct I/O.** The host and Agent can send and receive 32-bit words in either direction, to communicate control information across a low bandwidth link.
- **DDR memory.** The Agent and the host can initiate transfers to and from DDR memory, providing a very high bandwidth communications channel.

The ABI includes a variety of data and control signals, so for a DK user, the BSP provides a simpler interface, using Handel-C procedure calls to transfer data to and from the host and the DDR memory.

6.2.1 DDR memory access

The CPP board includes 256MB of DDR memory connected to the CPC, which manages access to it. The memory is intended to be used to buffer input and output data for an Agent, and an Agent might normally expect to receive an address from the host (via direct I/O) to tell it where to read and write data in the DDR memory. No memory management or protection is currently implemented, and there could be up to four separate Agents in each CPP board, so they must respect each others DDR memory space.

Note that for best efficiency, DDR transfers should be of larger numbers of words, as there is an overhead to setting up each transfer. Write transfers *may* complete quickly without blocking, as there is internal buffering in the CPC, read transfers will experience a latency of 40 clock cycles or more between the transfer being set up and the first word becoming available at the ABI. If possible, the user's program should initiate DDR read requests well ahead of the data being required, to mask some or all of this latency.

Due to the latencies involved in accessing the DDR memory, there are possible read-after-write hazards which must be avoided. These arise because completion of a write operation at the ABI does not guarantee that the data has already been written to the DDR memory. The first hazard is present if an Agent uses one ABI to write to the DDR

memory, and the other to read from it, as there is no guarantee that the transfers will complete in the order they were requested. The second hazard is that an Agent may write to the DDR memory, and then interrupt the host, which could then read from the memory. Both of these hazards can be avoided by performing a dummy read through whichever ABI was used for the DDR write, and not proceeding any further until the read has completed. This avoids the hazard because each individual ABI **does** serialise DDR requests, and the hazards only arise through access from both ABIs or from the host, so if the read has completed, the user can be sure that the write was also completed.

Each transfer of data to or from the DDR memory must be set up individually and completed before another is started through the ABI. A transfer is set up by sending the ABI an address and length, specifying the offset (in words) into the DDR memory and the number of words to read or write. After setting up a transfer, the specified number of 8-byte words **must** be read or written, and only then will the ABI accept further DDR transfer requests. Following is a summary of the procedures which the BSP provides for DDR access.

`ABI#_WriteDDRInit(Address,Length)`

where # is either 0 or 1

Call these procedures to initialise a write transfer to DDR memory through ABI0 or ABI1. Address is the word in the DDR memory at which to start the transfer, and Length is the number of words to write. The DDR memory capacity is 256MB, and the words are 8 bytes, so the total number of words is 2^{25} . Address and Length are therefore both 25-bit values. The call to either of these procedures will take a single clock cycle. After setting up a write transfer through an ABI, the specified number of words must then be written to that ABI before another transfer can be set up.

`ABI#_WriteDDRData(Data)`

where # is either 0 or 1

Call these procedures to write an 8 byte word to DDR memory through ABI0 or ABI1. A DDR write transfer must have already been set up, or else the ABI will not acknowledge receipt of the data and the procedure call will not complete. A call to either of these procedures will take at least one clock cycle to complete, but there is no upper limit on how long it will take, as it must wait for the ABI to acknowledge receipt of the data. In a sustained transfer of a large amount of data, the latency of each call will be two clock cycles on average, due to the bandwidth of the bus from the Agent FPGA to the CPC.

`ABI#_ReadDDRInit(Address,Length)`

where # is either 0 or 1

Call these procedures to initialise a read transfer from DDR memory through ABI0 or ABI1. Address is the word in the DDR memory at which to start the transfer, and Length is the number of words to read. The DDR memory capacity is 256MB, and the words are 8 bytes, so the total number of words is 2^{25} . Address and Length are therefore both 25-bit values. The call to either of these procedures will take a single clock cycle. After setting up a read transfer through an ABI, the specified number of words must then be read from that ABI before another transfer can be set up. Note that after calling either of these procedures, there will be a delay of at least 40 clock cycles before data becomes available at the ABI.

ABI#_ReadDDRData(Data)

where # is either 0 or 1

Call these procedures to read an 8 byte word from DDR memory through ABI0 or ABI1. A DDR read transfer must have already been set up, or else the ABI will not have any data ready to read, and the procedure call will not complete. A call to either of these procedures will take at least one clock cycle to complete, but there is no upper limit on how long it will take, as it must wait for the ABI to make the data available. In a sustained transfer of a large amount of data, the latency of each call will be two clock cycles on average, due to the bandwidth of the bus from the Agent FPGA to the CPC.

6.2.2 DDR memory simulation

Rather than simulate the entire 256MB DDR memory, and host access to it, the BSP simulation uses files on disk to represent data from the DDR memory. During the simulation, the user is prompted (via the DK Simulation Console) to enter filenames for reading and writing as required, allowing test data sets to be easily prepared in a file, and the results saved to a different file for analysis. If a DDR read or write is requested using an offset into the memory, the user is given the option of ignoring the offset. This would be useful in situations where the offset is very large compared to the length of the transfer, as it will result in smaller files being used to represent the DDR memory. Each ABI can have a read or write transfer in progress, as with the hardware version, though it is not recommended to perform two simultaneous transfers on the same filename in simulation.

All the normal DDR access procedures are still used, and the BSP will redirect all transfers to files on disk. The only difference a user may see is that the latency (in terms of the number of clock cycles) in simulation will be lower than in hardware.

6.2.3 Host I/O

As well as transferring data through the DDR memory, an Agent can communicate directly with the host, although at a lower data rate. It is also possible for the Agent to interrupt the host, for example if it requires some action from it. Described below are the procedures available in the BSP for host I/O:

ABI#_Interrupt(Data)

where # is either 0 or 1

Calling these procedures will send an interrupt through ABI0 or ABI1 to the host system. The Data parameter must be a 32-bit value, which will be transferred to the host system, allowing different events to be represented by the interrupt. These procedures take a single cycle to execute.

ABI#_ReceiveIOData(Data, Valid)

where # is either 0 or 1

Calling these procedures will test ABI0 or ABI1 to see if there is any I/O data being sent from the host. If there is, the data value will be returned, and Valid set to 1, otherwise Valid will be set to zero. The parameter Data is 32 bits, and Valid is 1 bit. Rather than

waiting for I/O, the procedure executes in a single cycle and sets the Valid bit to indicate the presence of data. As I/O data from the host is only present on an ABI for a single cycle, if the user *knows* the host will be sending data, and does not want to miss it, they should call these procedures within a single-cycle loop, which will only exit when Valid is set to 1.

`ABI#_SendIOData(Data)`

where # is either 0 or 1

Calling these procedures will send the 32-bit Data parameter to the host via ABI0 or ABI1. The procedure will execute in a single cycle, although the host will only get the data when it actually queries the CPP board to see if data is waiting.

`ABI#_Int_and_Data(Data)`

where # is either 0 or 1

These procedures have been provided to cater for a special case, which may be required in some applications. Calling them will send I/O data to the host, and simultaneously trigger an interrupt using the same data for the interrupt value. This could be used to alert the host immediately to the fact that there is I/O data waiting.

6.2.4 Host I/O simulation

All procedures for host I/O and interrupts can still be used during simulation. Procedures which would send data to the host result in messages being displayed on the DK Simulation Console. Procedures which receive data from the host will display a message on the DK Simulation Console, prompting the user to enter data, or else indicate that none is yet available.

6.3 Standard Agent Interface (SAI)

The SAI interface for LED and SRAM access is the same as for the ABI. The interface to the host communications is as follows.

6.3.1 Host I/O

`SAI_ReadFIFO(Data,SOF,EOF,BytesValid)`

Calling this procedure will read from the SAI input FIFO. This procedure blocks until data is valid. Data is 32 bits wide. SOF and EOF are flags for Start of File and End of File respectively. BytesValid indicates which of the bytes in the 32 bit word are valid.

`SAI_WriteFIFO(Data,SOF,EOF)`

Calling this procedure will write to the SAI output FIFO. This procedure blocks until the FIFO is has space. Data is 32 bits wide. SOF and EOF are flags for Start of File and End of File respectively.

```
SAI_QueryOutFIFOFull()
```

This procedure will return the state of the SAI output FIFO full flag. This is 1 if it is full, else zero.

```
SAI_QueryOutFIFOAlmostFull()
```

This procedure will return the state of the SAI output FIFO almost full flag. This is 1 if it is almost full, else zero.

```
SAI_QueryInFIFOEmpty()
```

This procedure will return the state of the SAI input FIFO empty flag. This is 1 if it is empty, else zero.

```
SAI_QueryInFIFOAlmostEmpty()
```

This procedure will return the state of the SAI input FIFO almost empty flag. This is 1 if it is almost empty, else zero.

```
SAI_SetupInFIFO(FillSize,RoomWatermark,PriorityWatermark)
```

Calling this procedure sets up the FIFO parameters. This must be called at startup, and may be called with care during operation. Make sure that FillSize is less than (512 – RoomWatermark), to leave room for data. PriorityWatermark must be lower than RoomWatermark.

```
SAI_SetupOutFIFO
(MaxFlushSize,AlmostFullWatermark,PriorityWatermark)
```

Calling this procedure sets up the FIFO parameters. This must be called at startup, and may be called with care during operation. Make sure that MaxFlushSize is less than (512 – AlmostFullWatermark), to leave room for data. PriorityWatermark must be lower than AlmostFullWatermark.

```
SAI_ReadVersion()
```

This procedure will return the 4 bit version number of the SAI.

```
SAI_ReadConfig()
```

This procedure will return the encoded user defined 24 bit config code.

```
SAI_ReadOOB(Address,Valid)
```

Calling these procedures will test the SAI to see if there is any I/O data being sent from the host. If there is, the data value will be returned, and Valid set to 1, otherwise Valid will be set to zero. The parameter Data is 32 bits, and Valid is 1 bit. Rather than waiting

for I/O, the procedure executes in a single cycle and sets the Valid bit to indicate the presence of data. As I/O data from the host is only present on an ABI for a single cycle, if the user *knows* the host will be sending data, and does not want to miss it, they should call these procedures within a single-cycle loop, which will only exit when Valid is set to 1.

SAI_WriteOOB(Data)

Calling these procedures will send the 32-bit Data parameter to the host via the SAI. The procedure will execute in a single cycle, although the host will only get the data when it actually queries the CPP board to see if data is waiting.

6.3.2 SAI Simulation

Not implemented yet.

7. Using the CP-DK

7.1 Setting up a project in DK

The BSP comprises two Handel-C libraries (one for simulation, and the other for EDIF build), a C++ object file and a Handel-C header file. To set up a new project using the BSP, follow these steps:

1. Create the project with chip type "Virtex-II"
2. Include "CPP.hch" in all Handel-C source files which will be accessing the BSP. In the source file with the main() function, #define the value MAIN_FILE before including the header, and #undef it afterwards. This is required for correct setup of the program clock.

```
#define MAIN_FILE
#include "../BSP/cpp.hch"
#undef MAIN_FILE
```

3. Go to the "Project Settings" dialog, and select the "Linker" tab. For "Debug", set the "Object/Library modules" box to the path to the BSP Debug library, and set the "Additional C/C++ Modules" box to the path to the CPP_sim.obj file. For "EDIF", set the "Object/Library modules" box to the path to the BSP EDIF library.
4. The project is now ready to build for either EDIF or Debug mode.

A batch file has been provided to automatically perform a Place And Route (PAR) at the end of an EDIF build. It is called `edifmake.bat`, and is in the `bin` directory. The template DK workspace supplied has a custom build step set up for the project which runs this batch file, and can be used as an example for creating new workspaces with an automatic PAR. Note that the path to the batch file must be correct, and that the batch file must be in the correct location within the BSP directory structure. If files are moved, pathnames in `edifmake.bat` must be edited accordingly.

7.2 Designing an Agent using CP-DK

Once a project is set up to use the CPP BSP, Agents can be designed in Handel-C with little knowledge of the hardware on which they will be implemented, other than the size and latencies of the SRAM and DDR memory. One important constraint, however, must be taken account of: the CPP board currently runs at a 133MHz clock.

The implication of this is that the Handel-C design must also be capable of running at 133MHz. To ensure this, the guidelines in the DK online help should be followed, paying particular attention to the use of the Technology Mapper and Logic Estimator, both of which can be used to help performance. When a design has been compiled for EDIF, the Xilinx ISE tools must be used to Place And Route (PAR) the EDIF to produce a bitfile from which a hardware Agent can be generated. During PAR, the Xilinx software will report the maximum speed at which the design is capable of running. It is suggested that the PAR tools be set to "maximum effort", to help achieve the 133MHz clock rate. If a design

does not PAR at 133MHz, the source code must be improved to give higher performance. This process is described in Celoxica application note AN68 *Optimization Timing Analysis* App note number 80 *Using Xilinx Design Manager with Handel-C* describes use of the Xilinx place and route tools with DK. The examples supplied with the CP-DK distribution automatically run the place and route tools if the `bitstream` target is selected.

8. Examples

8.1 Purpose of the Examples

Examples are included in the CP-DK. They are intended to illustrate basic usage of the API for the CP-DK, and do not aim to instruct the user in Handel-C or advanced use.

8.2 Requirements

For simulation or hardware build, the DK design suite must be installed, and CP-DK must also be present. The examples workspace is contained in the `/hardware/examples/` directory in the distribution. The workspace contains five projects, demonstrating the use of the LEDs, SRAM, DDR memory, interrupts and host I/O. Each of the examples is set up to build correctly for simulation and hardware, including invocation of Place And Route as a post-build step for the hardware build.

To implement the tutorial examples in hardware, you will require a CPP board installed, the *Bit2Agent* program and *Agent Configuration Tool*. These are documented separately.

8.3 Building and Running the Examples

For the following examples, simulation and hardware configuration files are stored in the `/simulation` and `/hardware` directories, respectively under each project.

To simulate, set the Active Configuration to Debug, and select "Rebuild All" from the "Build" menu. After the build is complete, set any breakpoints required and press F11 to "step into" the program or press F5 to run until a breakpoint.

To build a configuration file for the CPP board, set the Active Configuration to EDIF. "Rebuild All" from the "Build" menu to generate an EDIF file. Next set the Active Configuration to "Bitstream". "Rebuild All" from the "Build" menu will run the Place And Route tools to generate a bitfile. Next use the *Bit2Agent* program to convert the bit file into an Agent file. The Agent file can then be used to program one of the CPP board FPGA's using the *Agent Configuration Tool*.

8.4 LED test

The project named `led_test` contains a simple example of writing data to the LEDs on the CPP board. The `showStatus` macro takes an 8-bit parameter, and writes it to the CPP LEDs, then delays for approximately one second, allowing the user to see the value displayed on the LEDs. The main program contains an 8-bit variable `Data`, which is initialised to zero, then passed to `showStatus`, then incremented. These last two steps are within a `while(1)` loop, so the LEDs will indefinitely count from 0 up to 255.

For simulation make sure the "Watch Window" is visible, and select the "Locals" tab in it – the value being written to the LEDs is stored in the variable `bsp_LED_interface`.

8.5 SRAM access

The project named `sram_test` contains a program which writes pseudo-random data to the Agent SRAM banks, then reads it back and checks it is correct, lighting up LEDs to indicate its progress. The program illustrates proper use of the SRAM's pipelined nature, using two parallel loops, one generating addresses and the other generating data. The "data" loop is delayed by two cycles, to allow for the pipeline startup delay of the SRAM.

8.6 DDR access

The project named `ddr_test` contains a program which writes pseudo-random data to the DDR memory, then reads it back and checks it is correct, lighting up LEDs to indicate its progress.

Note that user input will be required through the DK Simulation Console, to specify files used to simulate the DDR memory.

8.7 Host I/O

The project named `iodata_test` contains a program which waits to receive IO data from the host, adds 1 to the data, and then returns it to the host. The value being returned will also be displayed on the LEDs.

Simulation I/O is through the DK Simulation Console, where the user can enter values to send to the Agent, and view the values being returned.

8.8 Interrupts

The project named `int_test` contains a program which continuously sends interrupts through ABI0 and ABI1 alternately. The data value sent with the interrupt is incremented each time, and is also displayed on the LEDs when running on the CPP.

Simulation interrupts will be displayed in the DK Simulation Console as the program executes.

Note that when running this example on the CPP board, the interrupts can be viewed using the Sysinternals "DebugView" utility, available for download from <http://www.sysinternals.com/ntw2k/freeware/debugview.shtml>.

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

Copyright © 2002 Celoxica Ltd. All rights reserved. Celoxica, the Celoxica logo and Handel-C are trademarks of Celoxica Limited. All other trademarks acknowledged. The information herein is subject to change and for guidance only

www.celoxica.com
