Celoxica

# DK4

## DK Libraries Manual

For DK version 4

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: RM-1011-2.0

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 the Americas

T: +1 800 570 7004

E: sales.america@celoxica.com

# 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 & Omissions

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

This manual does not include:

- instruction in VHDL or Verilog
- instruction in the use of place and route tools
- tutorial example programs. These are provided in the Handel-C User Manual

# 1 C++ Wide Number Library

The C++ Wide Number Library allows you to use Handel-C variables of unlimited width in C++ functions, and to manipulate those variables. If you are using ANSI-C or plugins, use the Numlib library instead.

### Classes

The library defines the classes Int and UInt (signed and unsigned integers of some defined width) and overloads the operators of those classes. It also provides methods associated with those classes, allowing you to perform standard Handel-C bit manipulation functions, such as taking and dropping bits.

You cannot cast `UInt` or `Int` variables to different widths, and may only perform operations on variables of the same type and width. To perform operations on `UInt` or `Int` variables of different widths, use Cat, Take or Drop to change the variables' width.

To convert `UInt` or `Int` variables to standard types, use the type conversion methods.

To use the C++ Number Class library, you need to add the *`InstallDir`*`\DK\Sim\Include` directory to your C++ compiler include search path. Use the `-I"`*`PathName`*`"` command line option (for Visual C++ or GCC) or refer to your C++ compiler documentation for how to do this.

In your C++ code, you need to include `hcnum.h` and specify the namespace containing the functions:

```
#include <hcnum.h>
using namespace HCNum;
```

## 1.1 Using the wide number library

The wide number library is used

- when transferring wide variables between a Handel-C program and a C/C++ program
- when translating C++ to Handel-C for simulation

This allows you to write simulation `.dll` files in C++ which are passed Handel-C variables through function calls.

## 1.2 Simple wide number library example

This simple example shows a wide variable being passed between a Handel-C file and a C++ file. The value of the Handel-C variable may be viewed in the debugger, but purely C++ variables can not.

**Handel-C:**

```
extern "C++" int 99 Fred(unsigned 2043 x);

set clock = external;
void main(void)
{
   int 99 a;
   unsigned b;

    a = Fred(b);
}
```

**C++:**

```
Int<99> Fred(UInt<2043> x)
{
    ...
}
```

## 1.3 Casting in the wide number library

As with Handel-C, you cannot cast between numbers of different widths. You can cast between Ints and UInts of the same width. You can extend or sign extend to substitute for casting between different widths.

### Example

```
Int<8> x;
UInt<7> y;

/*
 * To compare x and y
 * by casting y to an Int and
 * concatenating y with a 1-bit wide 0
 */
if (x>Cat(Int<1>(0),(Int<7>)y))
{
    ...
}
```

Visual C++ 6 has a known defect which causes the compiler to fail to infer template parameters. The example would have to be written as
`if (x>Cat<1, 7> (Int<1>(0), (Int<7>)y)) ...`

## 1.4 Types supplied

The following types are defined in the Wide Number Library to provide compiler independent definitions of non-standard types.

| Type | Description |
|---|---|
| `uint32` | unsigned 32-bit integer: equivalent to `unsigned long` or `unsigned __int32`. |
| `uint64` | unsigned 64-bit integer: equivalent to `unsigned long long` or `unsigned __int64`. |
| `int32` | signed 32-bit integer: equivalent to `signed long` or `signed __int32`. |
| `int64` | signed 64-bit integer: equivalent to `signed long long` or `signed __int64`. |

## 1.5 Int class

The `Int` template class allows you to represent signed integers of any width in C++. It consists of constructors, overloaded operators and methods for bit manipulation and input/output.

### Constructors

`Int<`***`width`***`>();`

`Int<`***`width`***`>(const `*`v`*`);`

`Int<`***`width`***`>(const char *`*`a`*`);`

The constructors allow you to construct a `Int` object of the specified width *`width`*, optionally initialized with a constant.

The initialization constant may be an `int` or a string constant.

### Example

```
Int<53> x = 725;
Int<10> y("0xf3");
Int<21> v(20057);
```

## 1.6 UInt class

The `UInt` template class allows you to represent unsigned integers of any width in C++. It consists of constructors, overloaded operators and methods for bit manipulation and input/output.

### Constructors

`UInt<`***`width`***`>();`

`UInt<`***`width`***`>(const int `***`v`***`);`

`UInt<`***`width`***`>(const char *`***`a`***`);`

The constructors allow you to construct a `UInt` object of the specified width ***`width`***, optionally initialized with a constant.

The initialization constant may be an `int` or a string constant.

### Example

```
UInt<53> x = 99;
UInt<10> y("0x8b");
UInt<21> v(20056);
```

## 1.7 Methods: Int and UInt

Identical methods are supplied for `UInt` and `Int` classes.

### Type conversion

| **Method name** | **Description** |
|---|---|
| `int32` IntOf() | Convert the current object to a `signed` int32 |
| `uint32` UIntOf() | Convert the current object to an `unsigned` uint32 |
| `int64` Int64Of() | Convert the current object to a `signed` int64 |
| `uint64` UInt64Of() | Convert the current object to an `unsigned` uint64 |

### Bit manipulation

| Method name | Description |
|---|---|
| `uint32` GetWidth | Gives the width of the current object in `uint32` |

### Input/Output

| | |
|---|---|
| `uint32 PrintString(char *Buffer, uint32 BufferLength, uint32 Base);` | Writes the value of the current object as a string to a buffer and returns required buffer size |
| `void Print(uint32 Base);` | Writes the value of the current object as a string to stdout |
| `void PrintFile(FILE *FilePtr, uint32 Base);` | Writes the value of the current object as a string to a file |
| `void WriteFile(FILE *FilePtr);` | Writes the value of the current object as raw binary data to a file |
| `void ReadFile(FILE *FilePtr);` | Reads raw binary data from a file and assigns it to the current object |

### Example

```
FILE *FilePtr = fopen("Jim", "w");
Int<53> x = 99;
x.PrintFile(FilePtr, 10);
```

## *1.7.1 Conversion to signed*

There are two methods of converting to a standard C++ signed number.

To convert a `UInt` or `Int` of 32 bits or less to a standard signed 32-bit integer in `int32`, use the method:

```
int32 IntOf()
```

If the original number is wider than 32 bits, the least significant bits will be returned.

To convert a `UInt` or `Int` of 64 bits or less to a standard signed 64-bit integer, use `int64 Int64Of()`

If the original number is wider than 64 bits, the least significant 64 bits will be returned.

**Example**

```
UInt<58> x;
Int<7> y;
int32 narrowInt;
int64 wideInt;

narrowInt = y.IntOf();
wideInt = x.Int64Of();
```

### 1.7.2 Conversion to unsigned

There are two methods of converting to a standard C++ unsigned number*.*

To convert a `UInt` or `Int` of 32 bits or less to a standard unsigned 32-bit integer in `uint32`, use

`uint32 UIntOf()`

If the original number is wider than 32 bits, the least significant 32 bits will be returned.

To convert a `UInt` or `Int` of 64 bits or less to a standard unsigned 64-bit integer, use

`uint64 UInt64Of()`

If the original number is wider than 64 bits, the least significant 64 bits will be returned.

**Example**

```
UInt<58> x;
Int<7> y;
uint32 narrowInt;
uint64 wideInt;

narrowInt = y.UIntOf();
wideInt = x.UInt64Of();
```

### 1.7.3 GetWidth method

`uint32 GetWidth()`

**Description**

`GetWidth()` returns the width of the current object in bits.

### Requirements

Header file: hcnum.h

Namespace: HCNum

### Example

```
Int<53> x = 99;
uint32 width;

width = x.GetWidth();
```

## 1.7.4 PrintString method

```
uint32 PrintString(char *Buffer, uint32 BufferLength ,
    uint32 Base)
```

### Description

PrintString() writes the value of the current object to the buffer pointed to by *Buffer* in the base specified by *Base*. If you set *Buffer* to NULL and *BufferLength* to 0, the return value indicates the maximum number of characters needed to print the string (including the terminating NULL). You can then use this to set *BufferLength* for the correct value.

### Parameters

| Name | Description | Possible values |
|---|---|---|
| *Buffer* | Character buffer to receive result | N/A |
| *BufferLength* | Length of buffer pointed to by Buffer | 0 to ($2^{32}$-1) |
| *Base* | Base to print in | 2, 8, 10 or 16 |

### Requirements

Header file: hcnum.h

Namespace: HCNum

### Example

```
Int<53> x = 99;
char hexvalue[60];
length = x.PrintString(hexvalue, 60, 16);
```

The array `hexvalue` will contain the text 0x63 (99 in base 16). `length` will be set to 5.

### 1.7.5 Print method

```
void Print(uint32 Base);
```

**Description**

`Print()` writes the value of the current object to `stdout` in the base specified by *`Base`*.

**Parameters**

| Name | Description | Possible values |
|---|---|---|
| *Base* | Base to print in | 2, 8, 10 or 16 |

**Requirements**

Header file: `hcnum.h`

Namespace: `HCNum`

**Example**

```
UInt<9> x = 447;
```

```
x.Print(10);
```

Prints 447 (the value of `x`) to `stdout`.

### 1.7.6 PrintFile method

```
void PrintFile(FILE *FilePtr, uint32 Base)
```

**Description**

`PrintFile()` writes the value of the current object to the file pointed to by *`FilePtr`* in the base specified by *`Base`*.

**Parameters**

| Name | Description | Possible values |
|---|---|---|
| *FilePtr* | Handle of file to be written to | N/A |
| *Base* | Base to print in | 2, 8, 10 or 16 |

**Requirements**

Header file: `hcnum.h`

Namespace: HCNum

### Example

```
UInt<57> x = 204;
FILE *fpointer;

fpointer = fopen( "data.out", "w" );
x.PrintFile(fpointer, 10);
fclose( fpointer );
```

## *1.7.7 WriteFile method*

```
void WriteFile(FILE *FilePtr);
```

### Description

WriteFile() writes the value of the current object to the file pointed to by *FilePtr* in raw binary data. After writing, the value of the file pointer *FilePtr* will be incremented to the new position. If an error occurs, the value of the file pointer *FilePtr* is undefined.

### Parameters

| Name | Description | Possible values |
|---|---|---|
| *FilePtr* | Handle of file to be written to | N/A |

### Requirements

Header file: hcnum.h

Namespace: HCNum

### Example

```
UInt<57> x = 204;
FILE *fpointer;

fpointer = fopen( "data.raw", "w" );
x.WriteFile(fpointer);
fclose( fpointer );
```

### *1.7.8 ReadFile method*

```
void ReadFile(FILE *FilePtr)
```

#### Description

`ReadFile()` reads the raw data from the file pointed to by *FilePtr* into the current object. The data read will be the same width as the current object. After reading, the value of the file pointer *FilePtr* will be incremented to the new position. If the end of file character is reached unexpectedly, the results are undefined.

#### Parameters

| Name | Description | Possible values |
|---|---|---|
| *FilePtr* | Handle of file to be read from | N/A |

#### Requirements

Header file: `hcnum.h`

Namespace: `HCNum`

#### Example

```
UInt<57> x;
FILE *fpointer;

fpointer = fopen( "data.in", "r" );
x.ReadFile(fpointer);
fclose( fpointer );
```

## 1.8 Functions

The following functions are supplied for bit manipulation:

#### Cat function

```
template<int W1, int W2> Int<W1 + W2>
    Cat(const Int <W1> &LHS, const Int<W2> &RHS);
```

```
template<int W1, int W2> UInt<W1 + W2>
    Cat(const UInt <W1> &LHS, const UInt<W2> &RHS);
```

`Cat()` concatenates *RHS* onto the end of *LHS* and returns the result. Equivalent to the Handel-C expression *LHS@RHS*.

### Drop function

```
template<int W1, int W2> Int<W2-W1>
    Drop(const Int <W2> &LHS);
```

```
template<int W1, int W2> UInt<W2-W1>
    Drop(const UInt <W2> &LHS);
```

`Drop()` returns all of the bits from *LHS* except the *W1* least significant bits. Equivalent to the Handel-C expression *LHS* \\ *W1*.

### Take function

```
template<int W1, int W2> Int<W1>
    Take(const Int<W2> &LHS);
```

```
template<int W1, int W2> UInt<W1>
    Take(const UInt<W2> &LHS);
```

Returns the *W1* least significant bits from *LHS*. Equivalent to the Handel-C expression *LHS* <- *W1*.

### Example

```
UInt<8> x;
UInt<7> y;
UInt<15> z;

z = Cat(x,y);

/*
 * To compare x and y
 * by concatenating y with a 1-bit wide 0
 */
if (x>Cat(UInt<1>(0),y))
{
    ...
}
```

## 1.8.1 Cat function

```
Int<W1 + W2> Cat(const Int <W1> &LHS, const Int<W2> &RHS);
```

```
UInt<W1 + W2> Cat(const UInt <W1> &LHS, const UInt<W2> &RHS);
```

### Description

`Cat()` concatenates *RHS* onto the end of *LHS* and returns a result whose width is the sum of the widths of the two operands. It is equivalent to the Handel-C expression *LHS@RHS*.

As in Handel-C, you can use `Cat` to concatenate zero with a number or sign extend to substitute for casting between different widths.

```
template<int W1, int W2> Int<W1+W2>
    Cat(const Int<W1> &LHS, const Int<W2> &RHS);
```

```
template<int W1, int W2> UInt<W1+W2>
    Cat(const UInt<W1> &LHS, const UInt<W2> &RHS);
```

### Parameters

| Name | Description | Possible values |
|---|---|---|
| *LHS* | Bits to form MS side of new number | N/A |
| *RHS* | Bits to form LS side of new number | N/A |

### Requirements

Header file: `hcnum.h`

Namespace: `HCNum`

### Example

```
UInt<8> x;
UInt<7> y;
UInt<15> z;

z = Cat(x,y);

/* To compare x and y
* by concatenating y with a 1-bit wide 0
*/
if (x>Cat(UInt<1>(0),y)) // Equivalent to if (x > ((unsigned 1) 0 @ y)) in
Handel-C
{
    ...
}
```

Visual C++ 6 has a known defect which causes the compiler to fail to infer template parameters. The example would have to be written as `z = Cat<8, 7>(x,y);` and `x>Cat<1, 7> (...)`

### 1.8.2 Drop function

`Int<W2-W1> Drop(const Int <W2> &LHS);`

`UInt<W2-W1> Drop(const UInt <W2> &LHS);`

#### Description

`Drop()` returns all of the bits from *LHS* except the *W1* least significant bits. It is equivalent to the Handel-C expression *LHS* `\\` *W1*.

```
template<int W1, int W2> Int<W2-W1>
    Drop(const Int<W2> &LHS);
```

```
template<int W1, int W2> UInt<W2-W1>
    Drop(const UInt<W2> &LHS);
```

#### Parameters

| Name | Description | Possible values |
|---|---|---|
| *W1* | Number of LS bits to drop | 0 to width of *Var* |
| *LHS* | Number to take bits from | N/A |

#### Requirements

Header file: `hcnum.h`

Namespace: `HCNum`

#### Example

```
UInt<7> x;
UInt<15> z;

x = Drop<8>(z);
```

Visual C++ 6 has a known defect which causes the compiler to fail to infer template parameters. The example would have to be written as `x = Drop<8, 15>(z);`

### 1.8.3 Take function

`Int<W1> Take(const Int<W2> &LHS);`

`UInt<W1> Take(const UInt<W2> &LHS);`

#### Description

`Take()` returns the *W1* least significant bits from *LHS*. It is equivalent to the Handel-C expression *LHS* `<-` *W1*.

```
template<int W1, int W2> Int<W1>
    Take(const Int<W2> &LHS);
```

```
template<int W1, int W2> UInt<W1>
    Take(const UInt<W2> &LHS);
```

### Parameters

| Name | Description | Possible values |
|---|---|---|
| *W1* | Number of LS bits to return | 0 to width of Var |
| *LHS* | Number to take bits from | N/A |

### Requirements

Header file: hcnum.h

Namespace: HCNum

### Example

```
UInt<7> x;
UInt<15> z;

x = Take<7>(z);
```

Visual C++ 6 has a known defect which causes the compiler to fail to infer template parameters. The example would have to be written as x = Take<7, 15>(z);

## 1.9 Operators supported by wide number library

The wide number library provides the following overloaded operators:

```
+     -     *     /     %     ==    !=    >
>=    <     <=    ~     &     |     ^
!     &&    ||    ++    --    +=    -=
*=    /=    %=    ^=    &=    |=
```

(where * is the multiplication operator and & is the bitwise AND)

These follow the Handel-C rules of width and types (for example, + will operate on two Ints of the same width or two UInts of the same width but not on different widths or mixtures of Int and UInt).

### Shift operators

Shift operators are non-standard, in that the right-hand operand must be an `unsigned int`.

`>>` `<<` `<<=` `>>=`

### Example

```
UInt<32> x = 64;
UInt<16> y = 4;
UInt<32> z;
z = x >> y.UIntOf();
z <<= 9;
```

# 2 Numlib library

The Numlib library contains routines to deal with values that are greater than 64 bits wide. The numbers are stored in a `NUMLIB_NUMBER` structure and the routines use this structure to operate on. There are routines to convert `NUMLIB_NUMBER` structures to 32 and 64-bit values.

You can use these routines in your plugin by including the header file `numlib.h` and linking with the appropriate Numlib library:

- If you are using Microsoft Visual C++ as your backend compiler, link to `numlib.lib`.
- If you are using GCC, link to `numlibgcc.lib`.

The numlib library files are installed in the `DK\Sim\Lib` directory.

Use the Numlib library if you are using ANSI-C or using plugins. If you are using C++, use the Wide Number library instead.

## 2.1 Arithmetic operations

These routines are supplied in the Numlib library to deal with values that are greater than 64 bits wide. The numbers are stored in a `NUMLIB_NUMBER` structure and the routines use this structure to operate on. You can use these routines in your plugin by including the header file `numlib.h`.

`include "numlib.h"` (in C or C++ code for plugin)

All operations are Handel-C like, and require that parameters are of the correct width. In some cases information about the sign of values must be provided. Note that in Handel-C you can only do divisions between variables with the same type and the same sign (signed by signed or unsigned by unsigned).

```
EXPORT void NumLibUMinus(NUMLIB_NUMBER *a, NUMLIB_NUMBER *b);
b = -a
```

```
EXPORT void NumLibAdd(NUMLIB_NUMBER *a, NUMLIB_NUMBER *b, NUMLIB_NUMBER *Re
sult)
Result = a + b
```

```
EXPORT void NumLibSubtract(NUMLIB_NUMBER *a, NUMLIB_NUMBER *b, NUMLIB_NUMBE
R *Result)
Result = a - b
```

```
EXPORT void NumLibMultiply(NUMLIB_NUMBER *a, NUMLIB_NUMBER *b, NUMLIB_NUMBE
R *Result)
Result = a * b
```

```
EXPORT void NumLibDivide(NUMLIB_NUMBER *a, NUMLIB_NUMBER *b, int Signed, NUMLIB_NUMBER *Result)
Result = a / b.
```

All numbers treated as signed or unsigned, depending on the value of `Signed`.

```
EXPORT void NumLibMod(NUMLIB_NUMBER *a, NUMLIB_NUMBER *b, int Signed, NUMLIB_NUMBER *Result)
Result = a % b.
```

All numbers treated as signed or unsigned, depending on the value of `Signed`.

```
EXPORT void NumLibDivMod(NUMLIB_NUMBER *a, NUMLIB_NUMBER *b, int Signed, NUMLIB_NUMBER *DivResult, NUMLIB_NUMBER *ModResult)
DivResult = a / b, ModResult = a % b.
```

All numbers treated as signed or unsigned, depending on the value of `Signed`.

## 2.2 Bitwise operations

These routines are supplied in the Numlib library to deal with values that are greater than 64 bits wide. The numbers are stored in a `NUMLIB_NUMBER` structure and the routines use this structure to operate on. You can use these routines in your plugin by including the header file `numlib.h`.

`include "numlib.h"` (in C or C++ code for plugin)

### *2.2.1 Logical operations*

```
EXPORT void NumLibNot(NUMLIB_NUMBER *a, NUMLIB_NUMBER *Result)
Result = ~a
EXPORT void NumLibAnd(NUMLIB_NUMBER *a, NUMLIB_NUMBER *b, NUMLIB_NUMBER *Result)
Result = a & b
```

```
EXPORT void NumLibOr(NUMLIB_NUMBER *a, NUMLIB_NUMBER *b, NUMLIB_NUMBER *Result)
Result  = a | b
```

```
EXPORT void NumLibXor(NUMLIB_NUMBER *a, NUMLIB_NUMBER *b, NUMLIB_NUMBER *Result)
Result  = a ^ b
```

### 2.2.2 Concatenation operations

In all the functions the int32 and int64 values are left aligned in line with the plugin interface. The results must be greater than 64 bits wide.

```
EXPORT void NumLibCat64_32(uint64 *a, unsigned long wa,
unsigned long *b, unsigned long wb, NUMLIB_NUMBER *Result)
```

Concatenate wa bits of 64-bit a and wb bits of 32-bit b and place it in value pointed to by Result.

```
Result = (int wa) a @ (int wb) b
```

```
EXPORT void NumLibCat32_64(unsigned long *a, unsigned long wa,
uint64 *b, unsigned long wb, NUMLIB_NUMBER *Result)
```

Concatenate wa bits of 32-bit a and wb bits of 64-bit b and place it in value pointed to by Result.

```
Result = (int wa) a @ (int wb) b
```

```
EXPORT void NumLibCat64_64(uint64 *a, unsigned long wa, uint64 *b, unsigned
long wb, NUMLIB_NUMBER *Result)
```

Concatenate wa bits of 64-bit a and wb bits of 64 bit b and place it in value pointed to by Result.

```
Result = (int wa) a @ (int wb) b
```

```
EXPORT void NumLibCat32_n(unsigned long *a, unsigned long wa,
NUMLIB_NUMBER *b,NUMLIB_NUMBER *Result)
```

Concatenate wa bits of 32-bit a with value b and place it in value pointed to by Result.

```
Result = (int wa) a @ b
```

```
EXPORT void NumLibCatn_32(NUMLIB_NUMBER *a, unsigned long *b,
unsigned long wb,  NUMLIB_NUMBER *Result)
```

Concatenate value a with wb bits of 32-bit b and place it in value pointed to by Result.

```
Result = a @ (int wb) b
```

```
EXPORT void NumLibCat64_n(uint64 *a, unsigned long wa, NUMLIB_NUMBER *b,
NUMLIB_NUMBER *Result)
```

Concatenate wa bits of 64-bit a with value b and place it in value pointed to by Result.

```
Result = (int wa) a @ b
```

```
EXPORT void NumLibCatn_64(NUMLIB_NUMBER *a, uint64 *b, unsigned long wb,
NUMLIB_NUMBER *Result)
```

Concatenate value a with wb bits of 64-bit b and place it in value pointed to by Result.

```
Result =  a @ (int wb) b
```

```
EXPORT void NumLibCat(NUMLIB_NUMBER *a, NUMLIB_NUMBER *b,
NUMLIB_NUMBER *Result);
```

Concatenate value a with value b and place it in value pointed to by Result.

```
Result= a @ b
```

### 2.2.3 Drop operations

```
EXPORT void NumLibDrop32(NUMLIB_NUMBER *a, unsigned long b,
unsigned long *Result)
```

Drop b bits from a and place it in 32-bit Result. Does not need to occupy the whole of Result.

```
Result = a \\ b
```

```
EXPORT void NumLibDrop64(NUMLIB_NUMBER *a, unsigned long b,
uint64 *Result)
```

Drop b bits from a and place it in 64-bit Result. Does not need to occupy the whole of Result.

```
Result = a \\ b
```

```
EXPORT void NumLibDrop(NUMLIB_NUMBER *a, unsigned long b,
NUMLIB_NUMBER *Result)
```

Drop b bits from a and place it in Result.

```
Result = a \\ b
```

### 2.2.4 Take operations

```
EXPORT void NumLibTake32(NUMLIB_NUMBER *a, unsigned long b,
unsigned long *Result)
```

Take b bits from a and place it in 32-bit Result. Does not need to occupy the whole of Result.

```
Result= a <- b
```

```
EXPORT void NumLibTake64(NUMLIB_NUMBER *a, unsigned long b,
uint64 *Result)
```

Take b bits from a and place it in 64-bit Result. Does not need to occupy the whole of Result.

```
Result= a <- b
```

```
EXPORT void NumLibTake(NUMLIB_NUMBER *a, unsigned long b,
NUMLIB_NUMBER *Result)
```

Take b bits from a and place it in Result.

```
Result= a <- b
```

### 2.2.5 Shift operations

```
EXPORT void NumLibLSL(NUMLIB_NUMBER *a, unsigned long b,
NUMLIB_NUMBER *Result)
Result =  a << b
```

```
EXPORT void NumLibLSR(NUMLIB_NUMBER *a, unsigned long b,
NUMLIB_NUMBER *Result)
```

`Result = a >> b`. Logical right-shift: the top bits are zero-padded.

```
EXPORT void NumLibASR(NUMLIB_NUMBER *a, unsigned long b,
NUMLIB_NUMBER *Result)
```

`Result = a >> b` Arithmetic right-shift: the top bits are sign-extended.

### 2.2.6 Bit selection operations

```
EXPORT void NumLibBitRange32(NUMLIB_NUMBER *a, unsigned long b,
unsigned long c, unsigned long *Result)
```

32-bit value pointed to by `Result = a [b - 1 : c]`

```
EXPORT void NumLibBitRange64(NUMLIB_NUMBER *a, unsigned long b,
unsigned long c, uint64 *Result)
```

64-bit value pointed to by `Result= a [b - 1: c]`

```
EXPORT void NumLibBitRange(NUMLIB_NUMBER *a, unsigned long b,
unsigned long c, NUMLIB_NUMBER *Result)
Result = a [b - 1: c]
```

### 2.2.7 Bit insertion operations

```
EXPORT void NumLibInsert32(unsigned long *a, unsigned long wa,
unsigned long s, NUMLIB_NUMBER *Result)
```

Insert bits of `a` into `Result` with LSB at position `s`. Width `a` is `wa` and `a` is <= 32 bits wide.

```
EXPORT void NumLibInsert64(uint64 *a, unsigned long wa,
unsigned long s, NUMLIB_NUMBER *Result)
```

Insert bits of `a` into `Result` with LSB at position `s`. Width `a` is `wa` and `a` is <= 64 bits wide.

```
EXPORT void NumLibInsert(NUMLIB_NUMBER *a, unsigned long s,
NUMLIB_NUMBER *Result)
```

Insert bits of `a` into `Result` with LSB at position `s`.

## 2.3 Comparison operations

These routines are supplied in the Numlib library to deal with values that are greater than 64 bits wide. The numbers are stored in a `NUMLIB_NUMBER` structure and the routines use this structure to operate on. You can use these routines in your plugin by including the header file `numlib.h`.

`include "numlib.h"` (in C or C++ code for plugin)

`EXPORT unsigned long NumLibCompareEq(NUMLIB_NUMBER *a, char *b)`

Return result of comparison of number `a` to string `b`

Equivalent to:

```
    NUMLIB_NUMBER *Temp;
    unsigned long Res;

    NumLibNew(&Temp, a->Width);
    NumLibSet(b, Temp);
    NumLibEquals(a, Temp, &Res);
    NumLibFree(Temp);
    return Res;
```

`EXPORT void NumLibEquals(NUMLIB_NUMBER *a, NUMLIB_NUMBER *b, unsigned long *Result)`

Return result of (`a == b`)

`EXPORT void NumLibNotEquals(NUMLIB_NUMBER *a, NUMLIB_NUMBER *b, unsigned long *Result)`

Return result of (`a != b`)

`EXPORT void NumLibSGT(NUMLIB_NUMBER *a, NUMLIB_NUMBER *b, unsigned long *Result)`

Return result of (`a > b`) (`a` and `b` signed)

`EXPORT void NumLibSGTE(NUMLIB_NUMBER *a, NUMLIB_NUMBER *b, unsigned long *Result)`

Return result of (`a >= b`) (`a` and `b` signed)

`EXPORT void NumLibSLT(NUMLIB_NUMBER *a, NUMLIB_NUMBER *b, unsigned long *Result)`

Return result of (`a < b`) (`a` and `b` signed)

`EXPORT void NumLibSLTE(NUMLIB_NUMBER *a, NUMLIB_NUMBER *b, unsigned long *Result)`

Return result of (`a <= b`) (`a` and `b` signed)

`EXPORT void NumLibUGT(NUMLIB_NUMBER *a, NUMLIB_NUMBER *b, unsigned long *Result)`

Return result of (`a > b`) (`a` and `b` unsigned)

`EXPORT void NumLibUGTE(NUMLIB_NUMBER *a, NUMLIB_NUMBER *b, unsigned long *Result)`

Return result of (`a >= b`) (`a` and `b` unsigned)

`EXPORT void NumLibULT(NUMLIB_NUMBER *a, NUMLIB_NUMBER *b, unsigned long *Result)`

Return result of (`a < b`) (`a` and `b` unsigned)

```
EXPORT void NumLibULTE(NUMLIB_NUMBER *a, NUMLIB_NUMBER *b, unsigned long
*Result)
```

Return result of (a <= b) (a and b unsigned)

```
EXPORT void NumLibCond(unsigned long *Condition, NUMLIB_NUMBER *a,
NUMLIB_NUMBER *b, NUMLIB_NUMBER *Result);
```

Return result of Condition ? a : b

Equivalent to:

```
if (*Condition==0)
{
    NumLibCopy(b, Result);
}
else
{
    NumLibCopy(a, Result);
}
```

## 2.4 File I/O and print: Numlib library

These routines are supplied in the Numlib library. You can use these routines in your plugin by including the header file numlib.h. Files for use by numlib routines must use the local NumlibFileOpen and NumlibFileClose routines

include "numlib.h" (in C or C++ code for plugin)

```
EXPORT void NumLibPrint(unsigned long Base, int Signed, NUMLIB_NUMBER *Source)
```

Print value pointed to by Source to standard output in Base (display as signed or unsigned according to Signed). If Signed is non-zero, number is treated as signed (e.g. "-1"). If Signed is zero, numbers will be treated as unsigned (e.g. "255")

```
EXPORT void NumLibPrintFile(FILE *FilePtr, unsigned long Base, int Signed,
NUMLIB_NUMBER *Source)
```

Write value pointed to by Source to file pointed to by FilePtr as above. FilePtr must be a pointer returned by NumlibFileOpen.

```
EXPORT unsigned long NumLibPrintString(char *Buffer, unsigned long BufferLength,
unsigned long Base, int Signed, NUMLIB_NUMBER *SourceIn)
```

Write value pointed to by SourceIn as string to Buffer in given Base (length of Buffer given in BufferLength ). BufferLength is the maximum length that will be written. If Signed is non-zero, number is treated as signed (e.g. "-1"). If Signed is zero, numbers will be treated as unsigned (e.g. "255"). If you call the function with Buffer set to NULL, it returns the maximum space required for the string.

`EXPORT FILE *NumLibOpenFile(char *filename, char *mode)`

Open the file "*filename*" in mode "*mode*" and return a pointer to the file. Mode may be one of `r`, `w`, `a`, `r+`, `w+` or `a+`.

`EXPORT void NumLibCloseFile(FILE *FilePtr)`

Close the file pointed to by `FilePtr` .

`EXPORT void NumLibWriteFile(NUMLIB_NUMBER *a, FILE *FilePtr)`

Write value pointed to by `a` in binary format to file pointed to by `FilePtr`. `FilePtr` must be a pointer returned by `NumlibFileOpen`.

`EXPORT void NumLibReadFile(NUMLIB_NUMBER *a, FILE *FilePtr)`

Read binary format number from a file pointed to by `FilePtr` and put the result in `a`. `FilePtr` must be a pointer returned by `NumlibFileOpen`. This is the reverse of `NumLibWriteFile`. The width of `a` must be correct. E.g.

```
NUMLIB_NUMBER *Fred;
FILE *FilePointer = NumLibReadFile("file.dat", "rb");
NumLibNew(&Fred, 453);
NumLibReadFile(Fred, FilePointer);
```

## 2.5 General number-handling routines

These routines are supplied in the Numlib library to deal with values that are greater than 64 bits wide. The numbers are stored in a `NUMLIB_NUMBER` structure and the routines use this structure to operate on. You can use these routines in your plugin by including the header file `numlib.h`.

`include "numlib.h"` (in C or C++ code for plugin)

`EXPORT void NumLibSet(char *a, NUMLIB_NUMBER *Result)`

Set value pointed to by `Result` to the value of string `a`.

For example:

```
NUMLIB_NUMBER *Fred;
NumLibNew(&Fred, 453);
NumLibSet("12452164748478321948732050832 94",
            Fred);
```

`EXPORT void NumLibCopy(NUMLIB_NUMBER *Source, NUMLIB_NUMBER *Result)`

Copy value pointed to by `Source` to value pointed to by `Result`.

`EXPORT uint32 NumLibBits(NUMLIB_NUMBER *a)`

Calculate the width of value pointed to by `a` and return number of bits (i.e. return the width of `a` specified in `NumLibNew`).

`EXPORT void NumLibSetBit(NUMLIB_NUMBER *a, uint32 Bit, int Value)`
Set bit `Bit` of variable pointed to by `a` to `Value` (0 or 1).

`EXPORT int NumLibGetBit(NUMLIB_NUMBER *a, uint32 Bit)`
Get value of bit `Bit` of variable pointed to `by a`.

`EXPORT int32 NumLibGetLong(NUMLIB_NUMBER *a)`
Convert value pointed to by `a` to 32 bits and return it. The least significant bits are used and the result is right aligned (i.e. normal numbers not plugin style numbers).

`EXPORT int64 NumLibGetLongLong(NUMLIB_NUMBER *a)`

Convert value pointed to by `a` to 64 bits and return it. The least significant bits are used and the result is right aligned (i.e. normal numbers not plugin style numbers).

## 2.6 Number allocation and de-allocation

These routines are supplied in the Numlib library to deal with values that are greater than 64 bits wide. The numbers are stored in a `NUMLIB_NUMBER` structure and the routines use this structure to operate on. You can use these routines in your plugin by including the header file `numlib.h`.

`include "numlib.h"` (in C or C++ code for plugin)

`EXPORT void NumLibNew(NUMLIB_NUMBER **Num, unsigned long Width)`
Allocate `Width` space for value indirectly pointed to by `Num`. Provide pointer to space acquired in `Num`.

For example:

```
NUMLIB_NUMBER *Fred;
NumLibNew(&Fred, 453);
```

`EXPORT void NumLibFree(NUMLIB_NUMBER *Num)`
Free allocated space for value pointed to by `Num`.

For example:

```
NumLibFree(Fred);
```

# 3 Introduction to the Plugin API

The Plugin Application Program Interface (API) defines how to write plugins to connect to the Handel-C simulator.

- A plugin is a program that runs on the PC and connects to a Handel-C clock or interface. It can be written in any language which supports C-calling conventions.
- The simulator expects the plugin to support various function calls and some data structures. The simulator also has an error function that can be called by the plugin (callback functions).

A numlib library is supplied to allow you to use numbers greater than 64 bits wide in your plugin.

## 3.1 Function name retention in C++

When creating a DLL, some C++ compilers may use a modified version of the name that a function has in a source file. To prevent this from happening you must either compile your plugin as a C file, or, if you are compiling it as C++, you must use an `extern` declaration to force the compiler to use the C linkage convention, which will leave function names unchanged.

To specify that a function should be linked using the C linkage convention in C++, place the string `extern "C"` immediately before the function definition. e.g.

```
#define dll __declspec(dllexport)

extern "C"
dll void PlugInOpen(HCPLUGIN_INFO *Info, unsigned long NumInst)
{
    /*
     * this function intentionally left blank,
     * initializing before the first simulation is run
     */
}
```

## 3.2 Specifying plugins in Handel-C source code

Plugins are specified in the Handel-C source code using the extlib, extinst and extfunc specifications.

These specifications may be applied to clocks or interface definitions.

### Clock example:

```
set clock = external "P1"
              with {extlib="plugin.dll", extinst="instance0"};
```

### Interface example

In the case of interface definitions, the specifications may be specified for individual ports or for the interface as a whole. For example:

```
interface bus_in(unsigned 4 Input) BusName()
              with {extlib="plugin.dll",
                    extinst="some instance string",
                    extfunc="BusNameGetValue"};
interface bus_ts(unsigned 4 Input with {extlib="plugin.dll",
                    extinst="some instance string",
                    extfunc="BusNameGetValue"})
          BusName(unsigned 4 Output with {extlib="plugin.dll",
                    extinst="some instance string",
                    extfunc="BusNameSetValue"},
                  unsigned 1 Enable with {extlib="plugin.dll",
                    extinst="some instance string",
                    extfunc="BusNameEnable"});
```

## 3.3 Simulator interface to plugins

Your plugin is identified to the simulator by:

- The name of the compiled .dll (the compiled plugin)
- The function calls that pass data between the plugin and the Handel-C program
- The instance name

These data are passed to the simulator using the following with specifications:

| | |
|---|---|
| extlib | Specifies the name of the DLL. No default. |
| extinst | Specifies an instance string. No default. |
| extfunc | Specifies the function to call to pass data to the plugin or get data from the plugin. Defaults to PlugInSet() for passing data to the plugin and PlugInGet() to get data from the plugin. |

### Data widths in the simulator

The simulator uses 32-bit, 64-bit or arbitrary width representations for data as appropriate. The Plugin API functions use pointers to long or unsigned long for data widths less than or equal to 32 bits, pointers to long long, unsigned long long, int64 or unsigned int64 for data widths greater than 32 bits but less than or equal to 64 bits, and pointers to NUMLIB_NUMBER * for data widths greater than 64 bits.

Data stored in long, unsigned long, long long, unsigned long long or int64 and unsigned int64 types is left-aligned. This means it if it is less than the full width of the word, it will occupy the most significant bits in the word and not the least significant bits. For example, 3 stored as a 3-bit wide number in a 32-bit word is represented as 0x60000000.

Where 32-bit or 64-bit widths are used, data is stored in the most significant bits.

## 3.4 Data structures

The C header file: plugin.h provides the data structure declarations required for any plugin.

| | |
|---|---|
| Structure passed on startup: | HCPLUGIN_INFO |
| Callback data structure: | HCPLUGIN_CALLBACKS |

### *3.4.1 HCPLUGIN_INFO*

The HCPLUGIN_INFO data structure passes essential information from the simulator to the plugin on startup.

The data structure declarations required for plugins are provided in the C header file: plugin.h.

```
typedef struct
{
    unsigned long Size;
    void *State;
    HCPLUGIN_CALLBACKS CallBacks;
} HCPLUGIN_INFO;
```

### Members

*Size* Set to sizeof(HCPLUGIN_INFO) as a corruption check.

*State* Simulator identifier which must be used in callbacks from the plugin to the simulator. This value should be passed in future calls to any function in the *CallBacks* structure.

*CallBacks* Data structure containing pointers to the callback functions from the plugin to the simulator.

### *3.4.2 Callback data structure*

The HCPLUGIN_CALLBACKS structure is a member of the HCPLUGIN_INFO structure passed to the PlugInOpen() function on startup. It contains pointers to the callback functions. The only one currently available for use with the Plugin API is PlugInError. You can call the PluginError function in your plugin to pass error messages to the simulator

The data structure declarations required for plugins are provided in the C header file: plugin.h.

### HCPLUGIN_CALLBACKS

*Size* should be set to size of(HCPLUGIN_CALLBACKS).

```
typedef struct
{
    unsigned long Size;
    HCPLUGIN_ERROR_FUNC PluginError ;
    HCPLUGIN_GET_VALUE_COUNT_FUNC PluginGetValueCount;
    HCPLUGIN_GET_VALUE_FUNC PluginGetValue;
} HCPLUGIN_CALLBACKS;
```

## 3.5 Simulator to plugin functions

These functions are called by the simulator to send information to the plugin. They are called when simulation begins and ends, and at points in the simulator clock cycle.

The plugin may act upon the call or do nothing. The plugin must implement the function with identical parameters. PlugInSet and PlugInGet may be replaced by user-defined names but the other function names must remain the same.

| When used | Function call | How often |
|---|---|---|
| First use of simulator in DK session | PlugInOpen | once per plugin |
| Start of simulation | PlugInInOpenInstance | once per instance of plugin |
| | PlugInOpenPort | once per interface port using the plugin |
| Simulator data transfer | PlugInSet | called when data on a port sending data TO the plugin changes |
| | PlugInGet | called whenever the simulator wishes to read data FROM the plugin |
| Start of simulated clock cycle | PlugInStartCycle | |
| Middle of cycle | PlugInMiddleCycle | called immediately before the simulator variables are updated |
| End of cycle | PlugInEndCycle | |
| End of simulation | PlugInClosePort | once per interface port using the plugin |
| | PlugInCloseInstance | once per instance of the plugin |
| End of DK session | PlugInClose | once per plugin |

### *3.5.1 PlugInOpen*

`void PlugInOpen(HCPLUGIN_INFO *Info, unsigned long NumInst)`

The simulator calls this function the first time that the plugin `.dll` is used in a DK session. Each simulator used will make one call to this function for each plugin specified in the source code.

*Info* Pointer to structure containing simulator callback information.

*NumInst* Number of instances of the plugin specified in the source code. One call to PlugInOpenInstance() is made for each of these instances.

### 3.5.2 PlugInOpenInstance

```
void *PlugInOpenInstance(char *Name, unsigned long NumPorts)
```

This function is called each time you start a simulation. It is called once for each instance of the plugin in the Handel-C source code. An instance is defined by the string used in the `extinst` specification. An instance is considered unique if a unique string is used. Thus the same instance may be used to connect a single `PlugInOpenInstance` call (identified by the `extinst` string) to a number of ports.

Your implementation of the function should return a pointer used to identify the instance in future calls from the simulator. This pointer may be used as you wish (for example, it may point to a new class created when `PlugInOpenInstance` is called). The instance pointer will be passed to future calls to PlugInOpenPort(), PlugInSet(), PlugInGet(), PlugInStartCycle(), PlugInMiddleCycle(), PlugInEndCycle() and PlugInCloseInstance(). It is not used by the simulator.

*Name* String specified in the `extinst` specification in the source code.

*NumPorts* Number of ports associated with this instance. One call to PlugInOpenPort() will be made for each of these ports.

### 3.5.3 PlugInOpenPort

```
void *PlugInOpenPort(void *Instance, char *Name,
    int Direction, unsigned long Bits)
```

This function is called each time you start a simulation. It is called once for each interface port associated with this plugin in the source code. The plugin should return a pointer to a variable used to identify the port in future calls from the simulator. This value will be passed to future calls to PlugInGet(), PlugInSet(), and PlugInClosePort().

The pointer returned by by `PlugInOpenPort` may be used as you wish. For example, it may be used to point to a structure or a new class that is associated with that port. It allows you to preserve information without using a global variable. It is not used by the simulator

*Instance* Value returned by the PlugInOpenInstance() function.

*Name* Name of the port from the interface definition in the source code.

*Direction* Zero for a port transferring data from plugin to simulator, non-zero for a port transferring data from simulator to plugin.

*Bits* Width of port.

### 3.5.4 PlugInSet (default name)

```
void PlugInSet(void *Instance, void *Port,
    unsigned long Bits, void *Value)
```

This function is called by the simulator to pass data from simulator to plugin. You may use any name you wish for this function (specified by `extfunc`) but the parameters must remain the same. It is guaranteed to be called every time the value on the port changes but may be called more often than that.

*Instance* Value returned by the PlugInOpenInstance() function.

*Port* Value returned by the PlugInOpenPort() function.

*Bits* Width of port.

*Value* Pointer to value. If *Bits* is less than or equal to 32 bits then this is a `long *` or `unsigned long *`. If *Bits* is less than or equal to 64 bits then this is an `int64 *` or `unsigned int64 *`. If *Bits* is greater than 64 bits then this is a `NUMLIB_NUMBER **`.

Data stored in `long`, `unsigned long`, `int64` and `unsigned int64` types is left-aligned. This means it occupies the most significant bits in the word and not the least significant bits. For example, 3 stored as a 3-bit wide number in a 32-bit word is represented as 0x60000000.

Functions which operate on `NUMLIB_NUMBER` structures are provided in the Numlib library.

Where 32-bit or 64-bit widths are used, data is stored in the most significant bits.

### 3.5.5 PlugInGet (default name)

```
void PlugInGet(void *Instance, void *Port,
    unsigned long Bits , void *Value)
```

This function is called by the simulator to get data from the plugin. You may use any name you wish for this function (specified by `extfunc`) but the parameters must remain the same. It is guaranteed to be called at least once every clock cycle but may be called more often than that.

| | |
|---|---|
| *Instance* | Value returned by the `PlugInOpenInstance()` function. |
| *Port* | Value returned by the `PlugInOpenPort()` function. |
| *Bits* | Width of port. |
| *Value* | Pointer to value. If *Bits* is less than or equal to 32 bits then this is a `long *` or `unsigned long *`. If *Bits* is less than or equal to 64 bits then this is a `long long` (GNU type) `*`, `unsigned long long *`, `__int64` (Microsoft specific type) `*` or `unsigned __int64 *`. If *Bits* is greater than 64 bits then this is a `NUMLIB_NUMBER **`.<br>Data stored in `long`, `unsigned long`, `__int64` and `unsigned __int64` types is left-aligned. This means it occupies the most significant bits in the word and not the least significant bits. For example, 3 stored in a 3-bit wide number in a 32-bit word is represented as 0x60000000.<br>Functions using `NUMLIB_NUMBER` structures are provided in the Numlib library. |

> Where 32-bit or 64-bit widths are used, data must be stored in the most significant bits. You must left-shift the number into the MSBs so it will be read correctly by the Handel-C code.

### 3.5.6 PlugInStartCycle

```
void PlugInStartCycle(void *Instance)
```

This function is called by the simulator at the start of every simulation cycle.

| | |
|---|---|
| *Instanc e* | Value returned by the PlugInOpenInstance() function. |

### 3.5.7 PlugInMiddleCycle

`void PlugInMiddleCycle(void *Instance)`

This function is called by the simulator immediately before any variables within the simulator are updated. You may use it to perform any appropriate action.

| | |
|---|---|
| *Instanc e* | Value returned by the `PlugInOpenInstance()` function. |

### 3.5.8 PlugInEndCycle

`void PlugInEndCycle(void *Instance)`

This function is called by the simulator at the end of every simulation cycle. You may use it to perform any appropriate action.

| | |
|---|---|
| *Instance* | Value returned by the `PlugInOpenInstance()` function. |

### 3.5.9 PlugInClosePort

`void PlugInClosePort(void *Port)`

The simulator calls this function when the simulator is shut down. It is called once for every call made to `PlugInOpenPort()`. It is passed the pointer that you provided in PlugInOpenPort(). This function allows you to perform any clean-up operations required (for example, if you created a new class when `PlugInOpenPort` was called, you may now destroy that class).

| | |
|---|---|
| *Port* | Pointer returned by the PlugInOpenPort() function. |

### 3.5.10 PlugInCloseInstance

`void PlugInCloseInstance(void *Instance)`

The simulator calls this function when the simulator is shut down. It is called once for every call made to `PlugInOpenInstance()`. It allows you to perform any clean-up operations required (for example, if you created a new class when `PlugInOpenInstance` is called, you may now destroy that class).

*Instance* Pointer returned by the PlugInOpenInstance() function.

### 3.5.11 PlugInClose

```
void PlugInClose(void)
```

The simulator calls this function when the simulator is shut down. It is called once for every call made to PlugInOpen().

## 3.6 Simulator callback error function

The simulator callback error function can be used by plugins to pass error messages to the Handel-C program.

The plugin receives a pointer to the function in the *Info* parameter of the PlugInOpen() function call made by the simulator at startup.

### 3.6.1 HCPLUGIN_ERROR_FUNC

The data structure declarations required for plugins are provided in the C header file: plugin.h.

```
typedef void (*HCPLUGIN_ERROR_FUNC)(void *State, unsigned long Level,char
*Message);
```

The plugin should call this function to report information, warnings or errors. These messages will be displayed during debug in the GUI Output window. In addition, an error will stop the simulation.

*State* State member from the HCPLUGIN_INFO structure passed to the PlugInOpen() function.

*Level* 0 Information
1 Warning
2 Error

*Message* Message string.

# 4 Plugins supplied

The following plugins can be used to help simulate Handel-C programs. They are installed in *InstallDir*\DK\Plugins.

| | |
|---|---|
| DKShare.dll | allows a port to be used by more than one plugin |
| DKSync.dll | synchronizes Handel-C simulations so that they run at the correct rate relative to one another |
| DKConnect.dll | connects simulation ports together so that data can be exchanged between simulations |

## 4.1 Connecting simulations together

DKConnect.dll allows you to connect two simulations together.

**Example**

To connect the simulations of two programs together, you use DKConnect.dll to connect them both to the same instance. In the example below, data from program A is sent via the port seg7_output.encode_out to the SS(7) instance of DKConnect.dll, and data is read from that instance into program B via the port seg7_input.in.

```
// Program A interface
interface bus_out() seg7_output(unsigned 7 encode_out)
   with { extlib="DKConnect.dll",
          extinst="SS(7)", extfunc="DKConnectGetSet"};

// Program B interface
interface bus_in(unsigned 7 in) seg7_input()
   with {extlib="DKConnect.dll",
          extinst="SS(7)", extfunc="DKConnectGetSet"};
```

### *4.1.1 DKConnect.dll syntax*

You connect a simulation to DKConnect.dll by specifying the following in the with specification for a port:

extlib="DKConnect.dll",

extinst="*terminalName* (*width*) [[*bitRange*] ]", `extfunc="DKConnectGetSet"`

Where:

*terminalName* is the name of the virtual terminal that the port is connected to. It may be any Handel-C identifier. All ports connected to *terminalName* are connected together. The terminal will be created if it does not exist.

*width* is the width of the terminal in bits. This must be the same for every occurrence of the same terminal name.

*[bitRange]* is optional. It specifies which bits of the port are connected to which bits of the terminal. If used, bitRange must specify the connections for all bits within the port. Port bits are defined by their position within *bitRange*; terminal bits are specified by value. The first (leftmost) value in *bitRange* represents the most significant port bit, and the last (rightmost) value the least significant port bit. Terminal bits can be specified as an inclusive range [*n:n*], or a number. To leave a port bit unconnected, specify X as its terminal bit value.

If *bitRange* is omitted, bit 0 of the port will be connected to bit 0 of the terminal, bit 1 to bit 1 etc.

### 4.1.2 DKConnect bit range

The string `extinst = "connect1(16)[13,14,X,X,11:8]"`

connects an 8-bit port to a 16-bit terminal `connect1` with the cross-connections below.

| **Port bits** | **Terminal bits** |
|---|---|
| 0 | 8 |
| 1 | 9 |
| 2 | 10 |
| 3 | 11 |
| 4 | X |
| 5 | X |
| 6 | 14 |
| 7 | 13 |

## 4.2 Sharing a port between plugins

You can share a port between two or more plugins using `DKShare.dll`.

- Output ports can be shared to distribute the same data to multiple plugins.
- Input ports can be shared so that more than one plugin can feed data into the program; for example, to simulate tri-state ports.

If more than one plugin provides data to the same port on the same clock cycle, the last piece of data fetched is the one used.

If a plugin is used within a DKShare share record then all other instances of that plugin must also occur within DKShare records. If you do not want to share another instance of the plugin, then connect the plugin to the port in a single Share record.

### *4.2.1 DKShare.dll syntax*

To share a port, the `with` specification of the port or interface must contain:

```
extlib = "DKShare.dll"
```

```
extfunc = "DKShareGetSet"
```

```
extinst = "ShareRecords "
```

The *ShareRecords* string consists of a Share record for every plugin which a port needs to be connected to.

**Share records**

Share records have the following syntax:

```
Share={extlib=<lib-name>, extinst=<extinst-string>, extfunc=<func-name>}
```

The items within angle brackets have the same meaning as they have when they occur as the `extlib`, `extinst` and `extfunc` fields.

| | Possible values | Default | Meaning |
|---|---|---|---|
| `extlib` | Name of a plugin `.dll` | None | Specify external plugin for simulator |
| `extinst` | Instance name (with optional parameters) | None | Specify simulation instance used |
| `extfunc` | Name of a function in the plugin | `PlugInSet` or `PlugInGet` depending on port direction | Specify external function in the simulator for this port |

```
interface bus_out() seg7_output(encode_out)
    with {extlib="DKShare.dll",
          extinst="Share={extlib=<7segment.dll>,
                   extinst=<A>,
                   extfunc=<PlugInSet>}
                   Share={extlib=<DKConnect.dll>,
                          extinst=<SS(7)>,
                          extfunc=<DKConnectGetSet>}",
          extfunc="DKShareGetSet"
         };
```

## 4.3 Synchronizing multiple simulations

If you want to simulate multiple programs with different clock periods, you can use `DKSync.dll`.

You inform the synchronizer of the relative clock rates of the programs. The synchronizer then suspends each simulation until it can complete a cycle in step with other simulations.

### *4.3.1 DKSync.dll syntax*

To invoke `DKSync.dll`, you use the following with specifications in the `set clock` statement:

```
extlib="DKSync.dll"
extfunc="DKSyncGetSet"
extinst="clockPeriod"
```

The *clockPeriod* string must contain a positive integer that represents the period of the clock. This is assumed to be in the same time units for all simulations that are to be synchronized.

```
set clock = external "P1" with {extlib="DKSync.dll",
   extinst="100", extfunc="DKSyncGetSet"};
```

### Using the same clock rate for more than one main function

If you want to set the same clock period for more than one main function, you need to append the *clockPeriod* for extinst with a suffix, to prevent the same clock being built for both main functions. For example:

```
set clock = external "a1" with {extlib="DKSync.dll",
   extinst="90:a1", extfunc="DKSyncGetSet"};
set clock = external "a2" with {extlib="DKSync.dll",
   extinst="90:a2", extfunc="DKSyncGetSet"};
```

(The a1 and a2 suffixes used for the extinst values do not need to be the same as the clock pin names.)

## 4.4 Plugins example: using DKSync.dll

This example consists of two separate Handel-C projects: Project A and Project B.

### Project A

- Increments a modulo-10 counter every cycle and outputs the value of the counter to the 7segment.dll plugin
- Outputs the value of the counter to the terminal called SS(7) every cycle

### Project B

- Increments a modulo-10 counter on alternate cycles and outputs the value of the counter to the 7segment.dll plugin
- On alternate cycles, reads the value from the terminal called SS(7) and outputs it to the 7segment.dll plugin

Project A’s cycles are 100 time units long. Project B’s cycles are 50 time units long. If you ran a simulation of the project, you would need to step through Project B twice for every step of project A.

To simulate the code in these source files, you would:

1. Create a workspace with two projects, Project A and Project B; one containing each source code file.

2. Create a new project with the project type System (select File>New>Project and then click on the System icon).
3. In the System-type project, select Project>Dependencies and check Project A and Project B as dependencies.
4. Select Build>Rebuild All.
5. Press Advance (Ctrl + F11) to start stepping through the simulation.

### 4.4.1 Plugins example: Project A source code

```
set clock = external "P1" with {extlib="DKSync.dll",
    extinst="100", extfunc="DKSyncGetSet"};

signal unsigned 7 encode_out;
interface bus_out() seg7_output(unsigned 7 output = encode_out)
    with {extlib="DKShare.dll",
          extinst="Share={extlib=<7segment.dll>,\
                          extinst=<A>,\
                          extfunc=<PlugInSet>}\
                          Share={extlib=<DKConnect.dll>,\
                                 extinst=<SS(7)>,\
                                 extfunc=<DKConnectGetSet>}",
          extfunc="DKShareGetSet"
         };

//Define values to light 7-segment display from 0 - 9
rom unsigned 7 encoder[10] =
     {0x01,0x4f,0x12,0x06,0x4c,0x24,0x20,0x0f,0x00,0x04};

void main(void)
{
    unsigned 4 count;

    count = 0;

    while(1)
    {
        par
        {
            count = (count==9) ? 0 : (count+1);
            encode_out = encoder[count];
        }
    }
}
```

### 4.4.2 Plugins example: Project B source code

```
set clock = external "P1" with {extlib="DKSync.dll",
    extinst="50", extfunc="DKSyncGetSet"};
signal unsigned 7 encode_out;

interface bus_out() seg7_output(unsigned 7 output = encode_out)
    with {extlib="7segment.dll",
          extinst="B",
          extfunc="PlugInSet"};

interface bus_in(unsigned 7 in) seg7_input()
    with {extlib="DKConnect.dll",
          extinst="SS(7)",
          extfunc="DKConnectGetSet"};

//Define values to light 7-segment display from 0 - 9
rom unsigned 7 encoder[10] =
    {0x01,0x4f,0x12,0x06,0x4c,0x24,0x20,0x0f,0x00,0x04};

void main(void)
{
    unsigned 4 count;

    count = 0;

    while(1)
    {
        par
        {
            count = (count==9) ? 0 : (count+1);
            encode_out = encoder[count];
        }
        encode_out = seg7_input.in;
    }

}
```

## 4.5 Writing a plugin: example

This example shows how to invoke the simulator to plugin functions. It consists of three files:

| | |
|---|---|
| Example Handel-C file | invokes the plugin through interfaces |
| Example plugin file | contains the plugin functions |
| C header file: plugin.h | defines the plugin structures |

### *4.5.1 C header file: plugin.h*

The `plugin.h` header file contains declarations of the data structures required for any plugin. The file is provided at installation in the directory *InstallDir*`\DK\Sim\Include`.

### *4.5.2 Writing plugins example: plugin file*

This example shows the use of the functions provided and the need to include empty functions.

```
#include <stdio.h>
#include <string.h>
#include <malloc.h>
#include <assert.h>

#include "numlib.h"
#include "plugin.h"

typedef struct
{
    char *fname; // Filename
    FILE *fptr;

    // Temporary storage for read or write value
    unsigned long value;

    // Port direction, 1 = simulator to plugin, 0 = plugin to simulator
    int Direction;
    int RisingEdge; // Set to 1 on a rising clock edge.
} InstanceInfo;

__declspec(dllexport)
void PlugInOpen(HCPLUGIN_INFO *Info, unsigned long NumInst)
{
}

__declspec(dllexport)
void *PlugInOpenInstance(char *Name, unsigned long NumPorts)
{
    // Allocate memory to store an InstanceInfo structure
    InstanceInfo *rval = malloc(sizeof(InstanceInfo));

    // Allocate memory to store the filename.
    rval->fname = malloc(strlen(Name) + 1);

    strcpy(rval->fname,Name);
    rval->RisingEdge = 0;

    return (void *)rval;
}

__declspec(dllexport)
```

```
void *PlugInOpenPort(void *Instance_, char *Name,
    int Direction, unsigned long Bits)
{
    InstanceInfo *Instance = (InstanceInfo *)Instance_;

    if (strcmp(Name,"CLK"))
    {
        // This is not a clock port
        Instance->Direction = Direction;

        if (Direction)
        {
            // simulator to plugin, so open file for writing
            Instance->fptr = fopen(Instance->fname,"w");
        }
        else
        {
            // plugin to simulator; opens file and reads first value
            Instance->fptr = fopen(Instance->fname,"r");
            fscanf(Instance->fptr,"%d",&Instance->value);
        }

        return (void *)Instance;
    }
    else
    {
        // This is a clock port.
        Instance->fptr = NULL;

        return NULL;
    }
}

/*
* PlugInClock is a user-written function which is used instead of
*  the default function of PlugInSet
*/
__declspec(dllexport)
void PlugInClock(void *Instance, void *Port_,
    unsigned long Bits, void *Value)
{
    static uint32 oldValue;
    // Check for a rising clock edge.
    if (*(uint32 *)Value && !oldValue)
    {
        ((InstanceInfo *)Instance)->RisingEdge = 1;
    }

    oldValue = *(uint32 *)Value;
}

__declspec(dllexport)
void PlugInGet(void *Instance, void *Port_,
    unsigned long Bits, void *Value)
{
    InstanceInfo *Port = (InstanceInfo *)Port_;

    *(uint32 *)Value = Port->value;

}

__declspec(dllexport)
void PlugInSet(void *Instance, void *Port_, unsigned long Bits, void
*Value)
{
    InstanceInfo *Port = (InstanceInfo *)Port_;

    Port->value = *(uint32 *)Value;
}

__declspec(dllexport)
void PlugInStartCycle(void *Instance)
{
}

__declspec(dllexport)
void PlugInMiddleCycle(void *Instance)
{
}

__declspec(dllexport)
void PlugInEndCycle(void *Instance_)
{
    InstanceInfo *Instance = Instance_;

    // If there has been a rising clock edge in this cycle…
    if (Instance->RisingEdge)
    {
        if (Instance->Direction)
        {
            // Write value to file.
            fprintf(Instance->fptr,"%d\n",Instance->value);
        }
        else
        {
            // Read next value from file.
            fscanf(Instance->fptr,"%d\n",&Instance->value);
        }

        Instance->RisingEdge = 0;
    }
}

__declspec(dllexport)
void PlugInClosePort(void *Port_)
{
    InstanceInfo *Port = (InstanceInfo *)Port_;

    if (Port)
    {
        fclose(Port->fptr);
    }
}

__declspec(dllexport)
void PlugInCloseInstance(void *Instance)
{
    free(((InstanceInfo *)Instance)->fname);
    free(Instance);
}

__declspec(dllexport)
void PlugInClose(void)
{
}
```

### 4.5.3 Writing plugins example: Handel-C file

```
/*
 *  User-written PlugInClock function replaces the
 * default name of PlugInSet here
 */

set clock = external "P1"
    with {extlib = "PluginDemo.dll",
          extinst = "test.txt", extfunc = "PlugInClock"};

unsigned 5 x;

#undef WRITING
#ifdef WRITING

interface bus_out() ob1(unsigned 5 out = x)
    with {extlib = "PluginDemo.dll",
          extinst = "test.txt", extfunc = "PlugInSet"};

void main(void)
{
    while(1)
        x++;
}

#else

interface bus_in(unsigned 5 in) ib1()
    with {extlib = "PluginDemo.dll",
          extinst = "test.txt", extfunc = "PlugInGet"};

void main(void)
{
    while(1)
        x = ib1.in;
}

#endif
```

# 5 Index