Institut für Datentechnik
und Kommunikationsnetze

Technische Universität
Braunschweig

OSIRIS
Command Language
Description

Revision 1.2

IDA-OCL-0001

May 2003

Prepared by
Tim Wittrock

Table of Contents

1 Scope

1.1 Purpose of this Document

This document is a short description of the **O**SIRIS **c**ommand **l**anguage (called OCL) and the use of the OCL-compiler. The language is used to create functions that operate the experimental devices and process data from the hardware.

Chapter 2 will provide a short introduction into the tasks and qualities of the language. After that, chapter 3 describes the main features of OCL for structured programming (like conditional execution and loops). In chapter 4 you will find information of built-in data types and how to define new types on demand. The next chapter (5) shows, how to use these types by declaring variables, and in chapter 6 you will learn how to access the data represented by these variables. Since not all values have to be variables, chapter 7 provides information for using constant values in your program. Since you probably want to manipulate data inside your programs, chapter 8 explains data manipulations like calculations and array operations. Chapter 9 will teach you how to use functions (user defined as well as built-in). Special commands that are used for time and event controlling are shown in chapter 10.

After that, we leave the description of OCL itself and read about the built-in preprocessor, which is used for commenting and definition of symbol replacements in chapter 11. Chapter 12 helps to understand the messages you may see during compilation or at runtime. Having presented a list of all keywords used by OCL in chapter 13, the subject of chapter 14 is customizing and using the OCL compiler and the ground part of the UDP manager.

1.2 Change Record

Date	Revision	Author	Affected Sections
12/18/2000	0.1	Wittrock	All sections
02/28/2001	0.3	Wittrock	All sections revised, new: compiler options
04/05/2001	0.4	Wittrock	All sections slightly revised
05/06/2001	0.5	Wittrock	logging information
07/10/2001	0.6	Wittrock	POP handling, OCL init-file, complete revision
07/18/2001	0.7	Wittrock	New configuration option “search path”
08/07/2001	0.8	Wittrock	New command “start”
01/07/2002	0.9	Wittrock	New command “halt”, new compiler options

Date	Revision	Author	Affected Sections
09/03/2002	1.1	Wittrock	New declaration "PM" (global variables)
05/15/2003	1.2	Wittrock	Token code compression; new settings option: maximum number of messages per second; auto start of UDPs; change in behavior of POP save command.

1.3 Abbreviations

RTL	Runtime Library
OCL	Osiris Command Language
OCT	Osiris Command Token

UDP	User-defined Procedure
POP	Persistent Osiris Program

1.4 Documentation Conventions

The following text will specify the typing conventions that are used to explain the OSIRIS command language syntax.

- `Keywords` have to be typed exactly as shown in the documentation. ¿ represents an end of line (new line).
- ***References*** refer to other elements. It can be for example a user-defined name or a complex expression defined in the documentation.
- **:=** following a ***reference*** specifies the definition of the ***reference***.
- **[** ***Expressions*** **]** enclosed in square brackets are optional and may be omitted.
- Enclosing **(** ***expressions*** **)** in brackets is used for grouping.
- **(** ***Expressions*** **)+** in brackets followed by a plus may appear one or more times.
- **(** ***Expressions*** **)*** in brackets followed by a asterisk may be omitted or appear one or more times.
- ***Statements*** are lists of expressions separated by semicolon.
- The vertical rule **|** between two expressions indicates an alternative. You may choose either the expression on the left or on the right side.

```
Examples are printed this way   // comments are placed behind “//”
```

2 Introduction

This is the documentation of the OSIRIS command language (OCL). This language allows the creation of complex programs to solve even complicated tasks. The programs are processed by a two-pass-compiler and executed by a token interpreter running on the DPU. The language supports multiprocessing and time- or event-driven execution. It can be used for

- Data processing
- Commanding and batch processing
- System control

To provide easy programming, the OCL syntax is similar to the syntax of the well-known programming language C. All of the tasks mentioned above can use common functions for data processing, so the user does not have to solve the same problems multiple. The main differences to C are the following:

+ Full runtime range checking of array indices.
+ Complex data types allowed as return values of functions
+ Mathematical operations on all elements of an array at once
+ Global variables at explicitly specified addresses can be used to access special memory areas
- No dynamic memory management functions and pointers
- Identifiers have to be unique. No multiple use of the same identifier for a type and a variable at the same time.

3 Basics

As mentioned before, OCL is used for many different tasks. Because of this, the configuration is divided in different parts.

- The interfaces of all UDPs and RTL functions present at the DPU are stored inside the file `OCLsym.txt`. This file is generated automatically by the OCL compiler and should not be edited by user. All following runs know functions generated by an earlier compiler run (unless they are deleted explicitly from symbol table) without any repetition of the function declarations. A simpler version of this symbol table (without exact parameter information) is available at DPU.

- Data types and global variables that are used repeatedly can be defined in a common file that is included before each compilation. In contrast to function definitions, data type definitions are *not* saved for later compiler runs.

- The OCL compiler can be used to compile a couple of UDPs at once as well as commands for immediate or delayed execution.

- The OCL compiler generates a command sequence for upload (and execution) of the token code and writes it to a file that can be sent to the DPU's UDP-manager.

- Sources of all compiled UDPs are archived in combination with their time stamps and other additional information.

3.1 Notation of Numbers

There are four different ways to specify constant integer values:

- Decimal, the standard, consisting of digits from `0` – `9`.
- Hexadecimal, specified by leading `0x` (zero x) followed by a string of `0` – `9`, `a` – `f` or `A` – `F`.
- Octal, specified by a leading `0` (zero) followed by a string of `0` – `7`.
- Dual, specified by leading `0b` (zero b) followed by a string of `0` and `1`.

The decimal number 29 for example may be represented by `29`, `0x1d`, `035` or `0b1101`. These notations are all equivalent.

3.2 Symbols Scope

Of course, you do not have to work with fix values, but you can use abstract symbols. These symbols can be used anywhere inside that scope in which they have been defined (or in a sub-scope of it). Symbol definitions are not valid outside this scope. It is not allowed to define a symbol multiple times inside the same scope, but you can overwrite the definition of a symbol inside of a sub-scope. Such a redefinition does not affect the definition of the embedding scope but only inside the sub-scope. After leaving the sub-scope, the former definition of the symbol is restored and can be used as before.

Whenever a symbol is used, the compiler looks for its current definition. It starts in the current scope. As long as the symbol is not found, the search will be continued in the next upper scope. A scope is enclosed by braces like that:

{ ***scope*** **}**

A scope may contain as many sub-scopes as desired, for instance:

{ ***scope*** **{** ***sub-scope1*** **} { {** ***sub-sub-scope*** **}** ***sub-scope2*** **} }**

❖ **Example**

```
{
  unsigned long A; // define A to be an unsigned long integer variable
  A = 0;          // set the value of A to zero
  {               // enter a sub-scope
    long B;       // define B to be an long integer variable
    A = 2;        // set the symbol A (char) from the outer scope to 2
    B = A;        // assign A’s content to the long integer B
  }               // leave the scope, A contains 2, B is lost.
  {               // enter a new sub-scope
    long A;       // create a new symbol A. The A from the outer scope is not
                  // accessible inside this scope any more
    // A = B;     // since B is neither defined inside this sub-scope, nor inside
                  // the outer scope, it is not accessible here.
    A = 12345;
  }               // leave this scope, the new definition and value of A is lost.
  A = A + 1;      // increment the unsigned long variable A. It now contains the
                  // value 3 (remember the change from 0 to 2 inside the first
                  // sub-scope!)
}
```

3.3 Conditional Execution

The syntax of conditional execution of program sequences is similar to C. The only exception is, that there are no `goto`s implemented in OCL.

3.3.1 if then else

`if` `(` ***expression*** **`)`** ***compound_statement*** **[`else`** ***compound_statement*** **]**

This statement is equivalent to the ANSI C statement. If the value of ***expression*** is non-zero (or `true`), the first ***compound_statement*** is executed. Otherwise (if ***expression*** is zero resp. `false`) the optional `else`-branch is executed. In nested ***if***-statements the first ***else*** is assumed to belong to the last ***if***.

3.3.2 .. ? .. : ..

The result of an expression like this

bool_expression **`?`** ***yes_expression*** **`:`** ***no_expression***

is ***yes_expression***, if the ***bool_expression*** is true (or non-zero, anyway) and ***no_expression*** otherwise. Only the resulting expression will be evaluated. The ***a ? b : c*** expression cannot be used as destination of an assignment or as non-constant reference parameter.

```
long lMax, l1, l2;
...
lMax = (l1 > l2) ? l1 : l2;   // this assigns the maximum of l1 and l2 to lMax
```

The type of the result of this expressions is the smallest common type of both possible results, if both are simple types. If the ***yes_expression*** and the ***no_expression*** are more complex, they have to be exactly of the same type. If necessary, use explicit type conversions to get fitting types.

```
unsigned long auc[4];
bool          b;
...
auc = b ? “yes“ : “no\0“;   // both arrays have to be of the same size!
```

3.3.3 switch - case

`switch` `(` ***switch_expr*** **`)` `{` (** **`case`** ***case_expr*** **`:`** ***statement_list*** **)* [** **`default`** **`:`** ***statement_list*** **]** **`}`**

This control flow statement is similar to the C switch-command (but not 100% equivalent).

case_expr **:=** ***expression*** **| (** ***low_limit*** **`..`** ***high_limit*** **) (** **`,`** ***case_expr*** **)***

low_limit **:=** ***high_limit*** **:=** ***expression***

The values of the ***case_expr***s are successively evaluated and compared to the value of the ***switch_expr*** (which is evaluated only once, on entering the switch statement). The first fitting **`case`**'s ***statement_list*** will be executed. Opposite to C, the ***expression***s may be variables as well and the ***switch_expr*** does not have to be of a simple type.

statement_list may contain any code as well as the **`break`** command, which causes the machine immediately to continue the execution after the **`switch`** statement. If the **`break`** command is omitted, the execution will run through all following **`case`** ***statement_list***s (without evaluating the ***case_expr***) until a **`break`** or **`return`** command is reached.

The ***case_expr***s may be any constant or variable value as well as a range, defined by **`..`** between the lower and the upper limit (in this order!):

`case` `2` `..` `5` `:` ***statement_list***

This executes the ***statement_list*** on 2, 3, 4 and 5. You may also define comma-separated lists of constant or variable values or ranges as ***expression*** like this:

`case` `2` `..` `5` `,` `7` `:` ***statement_list***

This executes the ***statement_list*** on if the ***switch_expr*** is equal to 2, 3, 4, 5 or 7.

Of course it is possible to use the normal C-like syntax for specifying lists instead of using commas:

`case` `2` `..` `5` `:`
`case` `7` `:` ***statement_list***

These two lines have the same effect as the line above.

The first case that fits the switch ***expression*** will be executed, regardless of any other fitting case expression following. Therefore the `default` statement has to be the last in the list, because it fits any value and would render all further `case`s useless.

Please note that switch does **not** use jump tables. For performance reasons you should make sure that the most frequently used branches are placed on top of the list of `case`s.

3.3.4 for

`for` `(` **[** ***init*** **]** `;` ***condition*** `;` **[** ***modify_expression*** **]** `)` ***statement***

The ***init*** expression normally initializes the counter variable before the first pass. The counter variable is checked inside the ***condition*** expression at the beginning of each pass. The ***statement*** inside the loop will be repeatedly executed, as long as the ***condition***-expression is true. The optional third ***modify_expression*** modifies the counter variable at the end of each pass.

statement may contain any code as well as the `break` and the `continue` commands. The `break` command causes the machine to leave the complete `for` loop immediately and continue with the next command. In contrast to that, the `continue` command causes the machine to leave the ***statement*** of the current pass only and continue with the execution of the loop at the ***modify_expression*** and ***condition***.

3.3.5 while

`while` `(` ***condition*** `)` ***statement***

The ***condition*** is evaluated at the beginning of each pass. The loop ***statement*** will be repeatedly executed as long as the ***condition*** is true (i.e. non-zero).

statement may contain any code as well as the `break` and the `continue` commands. The `break` command causes the machine to leave the complete `while` loop immediately and continue with the next command. The `continue` command causes the machine to leave the ***statement*** of the current pass and continue with the execution of the ***condition***.

3.3.6 do while

`do` ***statement*** `while` `(` ***condition*** `)`

The loop will be executed while ***condition*** is true. Since condition is evaluated at the end of each pass, ***statement*** will be executed at least one time.

The `break` and `continue` commands work as described before (see 3.3.5).

3.4 Functions

You can build functions that provide actions and calculations that have to be done by multiple program parts. This allows using the same code from different parts without duplicating. It is possible to deliver variables into the functions and receive results from it, so the code of the function itself can be hold quite versatile.

***resulttype functionname* (*parameters*) { *body* }**

The function can be called using the ***functionname*** from any other code. The ***body*** contains all desired actions and calculations and can access the ***parameters*** provided by the calling code.The function can return a result of the type ***resulttype*** that can be used in further operations of the calling code. The calling code can use the function call like any other value of the type ***resulttype***.

It is even possible to call the function’s code recursively. Functions will be subject of section 9.1.

❖ **Example**

The fibonacci numbers are recursively defined as sum of the two predecessors:

```
fib(0) := fib(1) := 1
fib(x) := fib(x-1) + fib(x-2)
```

This definition can be put into a simple function:

```
long fib (long x)
{
  if (x < 2)
    return 1;                        // let the result be 1
  else
    return (fib (x-1) + fib (x-2));  // return the sum as result of fib (x)
}
...
{
  long Value;

  Value = fib (12);     //  assign the fibonacci number of 12 to Value
}
```

3.5 Time- And Event Controlled Execution

Some tasks may have to be repeatedly executed once in a while. This can be done by using an endless loop that waits for a certain time (**`sleep`**, see 10.1) or for the arrival of a certain event (**`wait`**, see 10.2) before continuing.

4 Data Types

Like C, the OSIRIS command language operates with variables of previously (user-) defined types. The type of a variable has to be declared explicitly before the first use (see section 3.2 for more information about the scope of type and variable definitions). Unlike C, you may assign variables of different types, as long as they only differ in the type *names*. Types that consist completely of elements of the same type at the same relative positions are accepted as equal.

Different types of the same length may be assigned, but cause warnings.

OCL does not allow forward declaration of data structures.

4.1 Data Types for Local Variables

OCL data types are very similar to C data types with only one major exception: OCL contains no pointers. For creating function calls with returned parameters references of variables are used (similar to C++).

OCL does not allow explicit type conversions of complex data types. Only simple types or arrays of simple types may be converted explicitly by the user to prevent compiler warnings. If variables of different types are mixed in an expression, the smaller type will be converted implicitly to the larger type to prevent information loss. Variables of different types with the same size may be assigned to each other but will cause warnings.

The following description will explain the main syntax of definitions.

4.1.1 Simple Data Types

Type	Description
`[signed	unsigned] long`	32 bit values. If `[signed	unsigned]` is omitted, the default `signed` is used.
`double`	64 bit precision floating point value
`bool`	1 bit unsigned value stored inside a long integer (true == 1, false == 0)
`void`	Placeholder for “no value”.

The `void` type can be useful as type of a reference parameter, because this prevents type checking in calls to external functions. Although it is possible to assign anything to a void reference, it is not possible to assign a void type to any other variable. It is not even possible to do an explicit type conversion from void to any other type. That means, it is not possible to access data of void variables by using OCL.

4.1.2 Enumeration

Enumeration types may be defined like in C:

`enum` ***enumname*** **`{`** ***valuename*** **[`=` *constant*] (`,`** ***valuename*** **[`=` *constant* `]`)* `}`**

The ***valuename***s have to be unique. They represent long integer values. If no ***constant*** is specified, the value will be set to the value of the previous ***valuename*** incremented by 1.

Enumeration variables or constants (like the ***valuename***s) may be assigned to long variables without problems, but assigning a long variable to an enumeration variable will cause warnings.

Please note that the increment and decrement operators do **not** evaluate the predecessor or successor of the current value but only increment or decrement its integer value by one. Note also that it is not possible to use a ***valuename*** to specify the (part of) a ***constant*** value of another ***valuename*** before the enumeration definition is completed.

❖ **Example**

```
enum Colors { BLACK, RED = 10, ORANGE, YELLOW, GREEN, BLUE, VIOLET, WHITE = 20 };
```

This defines the data type `Colors` and the constants `BLACK` (value 0), `RED` (defined to represent 10), `ORANGE` (11), `YELLOW` (12), `GREEN` (13), `BLUE` (14), `VIOLET` (15) and `WHITE` (20). You may use these constants instead of their values in your program whenever you want. Please note that the constants are no strings. Even when you use a enumeration constant (like `YELLOW`), the compiler will handle it as if the value 12 is specified directly.

4.1.3 User Defined Types

The user may define types of specified names by using a statement like this:

`typedef` ***basetype newtypename*** **(`[`** ***number_of_elements*** **`]`)***

This statement defines a new type named ***newtypename*** that matches the ***basetype***. The ***basetype*** may be any simple or more complex or even user-defined type. If the type definition is followed by one or more **`[`** ***number_of_elements*** **`]`**, the new type becomes an array of ***basetype*** elements with the given dimensions.

4.1.4 Arrays

Array types can be defined like in C by a variable declaration followed by one or more pairs of square brackets defining the dimension. To define an array type you have to use **`typedef`**:

`typedef` ***basetype newtypename*** **(`[`** ***number_of_elements*** **`]`)+**

This creates a new data type named ***newtype***, that is an array of ***basetype*** elements. For more information about arrays see section 5.4.

❖ **Example**

```
typedef unsigned long textlinetype [80];
typedef textlinetype  textscreentype [25];
```

The first line defines a type named `textlinetype` that consists of 80 unsigned longs. This type can be used to represent a line of a conventional text-terminal. The second line defines a type `textscreentype` that consists of 25 of those text lines. If you do not need the definition of a single text line, you may specify the `textscreentype` with the following line instead:

```
typedef unsigned long textscreentype [80][25];
```

4.1.5 Structures and Unions

Structures are bundles of variables that will be placed one after another in the memory. Unions are bundles of variables that will be placed at the same start position in the memory. The definition of structures and unions is identical to C syntax:

(`struct` | `union`) [*typename*] `{` (*vardefinition* `;`)+ `}`

vardefinition may be any definition of variables. It may be a structure or union itself. In this case, omitting the name of ***vardefinition*** will put its elements flat into the hierarchy.

If you want to use the data structure multiple times, you should define an own type. You can do this either using the keyword **`typedef`** (see 4.1.3) or specifying ***typename***.

❖ **Example**

```
struct TwoChars   // definition of the type TwoChars
{
  long c1;
  long c2;        // the address of c1 is (address of c1)+1
};
```

This is equivalent to

```
typedef struct  // definition of the type TwoChars
{
  long c1;
  long c2;
} TwoChars;
```

The definition of a union may look like that:

```
union DemoUnion
{
  struct
  {
    long c1;
    long c2;
  };
  long s;    // c1 is identical to s
};
```

You will find examples for even more complex structures in section 6.4.

4.1.6 *“Constant” Variables*

Variables and parameters can be declared to be constant values by using the keyword **`const`**. Constant variables cannot be altered. The keyword **`const`** is important if you want to allow constant values as reference parameters. The declaration of constant parameters may look like this:

```
void Something (unsigned long& aul1[], const unsigned long& aul2[]);
```

This line declares the parameter `auc2` of the function `Something` to be constant. That means, it is not allowed to use any write access to `auc2` inside the function.

5 Variable Declaration

The OSIRIS command language contains a C-like variable declaration. The only major exception is the absence of pointers, because these variables could easily cause program crashes or unexpected side effects. Use of global variables is limited to user specified memory areas. Please remember that using global variables may cause unexpected side effects, especially in multithreading environments.

5.1 Local Variables

Local variables may be declared like this:

type variablename* [`=` *constant*] (`,` *variablename* [`=` *constant*])

The ***type*** may be any of the data types mentioned in the previous chapter (4, Data Types).

The ***constant*** is used to define an initial value for the declared variable. Static variables will be set to this value only once (before the first execution). If the initial value is omitted, the variable will not be initialized and may contain random values (static variables will be initialized with 0, however)

Normally, local variables are located on the local stack, which allows fast accesses for calculations. Unfortunately, the size of the local stack is limited. To prevent early stack-overflows, the compiler will automatically generate code for storing large variables in dynamically allocated memory. Remember that this causes (internally) a little bit more overhead on accesses to elements of large variables.

5.2 Global Variables

There are two different kinds of global variables: They can be used to transfer data between different UDPs (1) or to access certain memory areas inside image memory (e.g. for image processing) (2).

5.2.1 Global variables for data exchange

For storing and transferring data between UDPs, global variables should be placed inside the program memory (better data protection). It is possible to specify the location of the variable inside a preserved area with an ***offset*** relative to the beginning of the area like this

`PM` ***type variable*** `at` ***offset*** `;`

But it is also possible to place the variable directly behind the previously declared global variable by omitting the offset, like this:

`PM` ***type variable*** `;`

The absolute address of global variables inside program memory can not be specified.

Accesses to data structures inside program memory can not be interrupted by accesses from other UDPs. This helps to ensure data integrity by preventing accesses to partial modified (= probably invalid) data structures.

```
Struct tTempSensor
{
  long lTime;
  long lTemperature;
};

PM tTempSensor oGlobTemp[20];

void ProcessTemperatures ()
{
  tTempSensor oLocalTemp[20];

  oLocalTemp = oGlobTemp;  // this assignment cannot be interrupted
  // all time entries of oLocalTemp correspond to the temperatur entries
  // now continue with the local copy, since global data may change…
  ...
  // do not use the following lines:
  oLocalTemp[0].lTime = oGlobTemp[0].lTime;
  // oGlobTemp[0] may change now
  oLocalTemp[0].lTemperature = oGlobTemp[0].lTemperature;
}

void BuiltTemperatures ()
{
  tTempSensor oLocalTemp[20];

  // fill the local temperatur and time arrays
  ...
  // write the whole array at once
  oGlobTemp = oLocalTemp;
}
```

5.2.2 Global variables for access to certain addresses

If certain memory addresses have to be accessed, global variables have to be declared as follows:

`fixed` ***type variable*** **`at`** ***address*** **`;`**

The absolute ***address*** of the global variable has to be specified explicitly inside the declaration. This kind of global variable should **not** be used for data storing, since the image memory is not well protected against data corruption.

The ***variable*** may be an array or any other type that will be stored beginning at ***address*** (inside the DPU's local memory). Please note that ***address*** should be chosen with care, since accessing illegal memory can cause program crashes (the compiler can not check the integrity of the specified memory).

After this declaration, ***variable*** can be used like any other local variable in all UDPs.

Global variables are not initialized automatically.

5.3 Static Variables

Variables may be declared static by using the keyword `static`. While 'normal' variables do not hold their values when the scope is left, static variables remain in memory and do not alter the stored value between leaving and re-entering the scope. This behavior is similar to C.

Since accessing static variables is slower than accessing internal variables you should use static variables only if necessary.

Static variables will be initialized with zero, unless another constant value is specified inside the declaration. The initialization of the static variable is done only once (at compile-time).

5.4 Arrays

Arrays may be declared like follows:

***type arrayname* ([*number_of_elements*])+**

As you see, it is possible to declare one-, two- or any other multidimensional array variable just by specifying a sequence of desired number of elements.

```
unsigned long asz[20][80];
unsigned long al[20*80];
```

The first line specifies a two-dimensional array of longs. It may be seen as a list of 20 lines with 80 columns each. Using `asz[5]` would specify the whole sixth line. `asz` specifies the whole block of 20 * 80 values.

The second line specifies a one-dimensional array `al` of the same size as `asz`. The main difference is the access to elements:

- `asz[line][column]` is equivalent to `al[line*80 + column]`.
- `asz[line]` is equivalent to `al[line*80, 80]`.
- `asz` is equivalent to `al`.

number_of_elements may be omitted in special cases:

- Inside of function headers to allow passing arrays of variable length to functions. If the number of elements is not specified, the parameter has to be declared as reference (using `&`). Runtime range checking will prevent illegal memory accesses.
- Inside of unions. The size of the array will be set to the maximum size that fits into the union without changing the union's size. Runtime range checking is not affected; it is not possible to access memory outside the union. The actual size of the array is known at compile time, so range checks are done at compile time already.
- If you specify a default value for the array variable, you may omit the number of elements of the outermost dimension, since it is given by the size of the default value. The actual size of the array is known at compile time, so no runtime checks are needed.

Note that only the size of the first dimension may be omitted, anyway. So the syntax of the declaration of arrays with unknown size at compile time is one of the following:

type varname* [ ] ([*length*])

***type arrayname* [ ] ([*number_of_elements*])* = *default_value* ;**

For instance, you may specify an array of three strings like this:

```
unsigned long asz[][7] = {“first ”, “second”, “third ” };
```

This is equivalent to the following declaration:

```
unsigned long asz[3][7] = {“first ”, “second”, “third ” };
```

Please note that the strings `“first”`, `“second”` and `“third”` are assumed to be arrays with 7 unsigned longs each. The length has to fit exactly for assignments.

6 Accessing Data

Generally, all accesses to local or global memory are handled identically. It makes no difference for the user to use local variables or external data. Basically, accessing variables is very similar to C, with a few exceptions that will be described in this chapter

6.1 "Standard" Variables

The standard variables (local or global variables) can be accessed like in C.

Although local variables use memory in the machine stack, global variables use own memory and external variables use memory mapped by the hardware driver, the user does not have to pay attention for it – the syntax of all accesses is identically.

6.2 "Constant" Variables

Variables that are declared to be constant (**`const`**) can be used only for read accesses. It is not possible to modify their contents (write access). Constant variables will probably be in most cases parameters of a function that may not be modified in any way inside the function.

```
void Something (unsigned long& auc1[], const unsigned long& auc2[])
{
  auc1[0] = ’1’;     // allowed, since auc1 is not const
  // auc2[0] = ’2’;  // error, auc2 is constant and must not be altered.
  auc1[1] = auc2[2]; // ok, auc1 may be altered and auc2 may be read
}

void Anything ()
{
  unsigned long auc[100];
  const unsigned long cauc[] = “constant”;  // assignment in definition is ok

  // Something (“forbidden”, “allowed”);  // since the first parameter is not
                                          // declared to be constant, it is not
                                          // allowed to use a constant here.
  // Something (cauc, auc);     // same as before: first parameter must not be
                                // constant. The second parameter may be variable,
                                // but can not be altered inside the function
  Something (auc, auc);  // ok, but the second parameter is not initialized yet.
                         // Since both parameters are references, auc1 and auc2
                         // will access the same memory inside the function.
  Something (auc, cauc);

  // cauc[3] = ’a’;      // not allowed, since cauc is constant!
}
```

6.3 References

References to variables are allowed only in function headers. All accesses to the referenced parameter inside the function will be accesses on the outside variable itself. A parameter is defined as reference by following the type by **&**:

type* & *parametername

The ***type*** may be any simple or user defined type. Note that **`&`** is not an address operator. Its use is allowed only in the parameter list of function declarations, not to build a reference return type. Generally, the source type and the destination (reference) type have to fit exactly, typecasting is not possible.

6.3.1 *Using Constant Values as Reference*

You cannot assign constant values to reference-parameters as long as the parameter is not specified to be constant by using the keyword **`const`** in the function declaration. Parameters declared to be constants may not be altered inside the function (see 4.1.6).

6.4 Structures and Unions

Elements of complex data structures (struct and union) can be accessed like in ANSI C:

datastructure* `.` *elementname

The ***datastructure*** may be an element of any other data structure (array, struct or union) itself. Since elements of unnamed substructures are put flat into the superior data structure, they are accessed like any other element of the superior structure (without an additional `.` that would indicate a deeper hierarchy).

You can copy whole data structures. Local structures and unions may be byte-filled with a constant value by assigning a constant char (no variables and no values that do not fit into 8 bits!) to the whole structure or union. This feature is supposed to be used for fast initialization of all bits to a defined state, so the constant will be probably in most cases be zero (no bit set) or 0xff (all bits set). The syntax is the following:

datastructure* `=` *constant_char_value

❖ **Example**

(Note: The type `TwoChars` has been defined in section 4.1.5 already.)

```
union  // definition of the variable SimpleUnion
{
  TwoChars;    // var name omitted, creating SimpleUnion.c1 (offset 0) and
               // SimpleUnion.c2 (offset 1)
  long s;      // creating SimpleUnion.s (offset 0)
  struct       // creating SimpleUnion.TwoUnsigned (offset 0)
  {
    unsigned long uc1;   // SimpleUnion.TwoUnsigned.uc1 (offset 0)
    unsigned long uc2;   // SimpleUnion.TwoUnsigned.uc2 (offset 1)
  } TwoUnsigned;
  long  ac[];  // create an array of longs that is as large as possible to
               // fit inside this union without enlarging it. The resulting
               // array ac will consinst of 2 elements here.
} SimpleUnion;

SimpleUnion.TwoUnsigned.uc1 = SimpleUnion.c2;
```

You can also use a previously user-defined type:

```
union  SimpleUnionType // definition of the type SimpleUnionType
```

```
{
  TwoChars SubChars;   // var name given, creating XYZ.SubChars.c1
                       // (offset 0) and XYZ.SubChars.c2 (offset 1)
  long s;      // creating XYZ.s (offset 0)
  struct       // creating a substructure without name. (offset 0)
  {
    unsigned long uc1;   // XYZ.uc1 (offset 0)
    unsigned long uc2;   // XYZ.uc2 (offset 1)
  };
  long  ac[];  // create an array of characters that is as large as possible to
               // fit inside this union without enlarging it. The resulting
               // array ac will consinst of 2 elements here.
};
// since this is just a type definition, there are no sub-elements accessible.
// After declaring a variable of this type, you are able to access its elements
// by replacing the XYZ with the variables name.

SimpleUnionType  SecondSimpleUnion;  // declare a variable of this type

SecondSimpleUnion.uc1 = SecondSimpleUnion.SubChars.c2;
```

Please note that `SimpleUnion` and `SecondSimpleUnion` contain different unnamed substructures, but represent the same internal structure.

6.5 Arrays

The access on single elements of Arrays is just like handled in C:

***array* [*index*]**

The ***array*** may be an element of any other data structure itself.

Arrays may be byte-filled with a constant value by assigning a constant char value to the whole array. It is not possible to use variables (even if they are declared to be constant) or non-char values as source for byte filling.

6.5.1 Accessing Ranges

You may access a subset of an array at once by specifying a range of elements. Since all specified elements has to build one block without gaps, the range definition is allowed only for the last specified dimension and may not be followed by any further element specification. There are two different ways to specify array ranges:

***array_variable* [*first* . . *last*]**

The ***first*** value has to be less than ***last***. This returns the elements of the array starting at index ***first*** and ending with index ***last***. The other way is to specify the starting element and the number of elements inside the range:

***array_variable* [*first* , *size*]**

size specifies the number of elements and has to be a value equal or greater than zero, ***first*** may be a variable that contains the value for the index of the first element of the specified range.

It is possible to use ranges for copying a part of an array into another part of the same array, even if the source and destination areas overlap.

For performance reasons you should prefer using array ranges instead of loops over array elements. This refers to assignments as well as for simple operations (see 8.2 also).

❖ **Example**

```
union DummyUnion  // size specified by the included structure: 7 bytes
{
  char ac[];      // length will be set to 7 elements  (7 bytes)
  long al[];      // length will be set to 1 element   (4 bytes)
  short as[];     // length will be set to 3 elements  (6 bytes)
  char aac[][3];  // will be set to aac[2][3] (6 bytes)

  struct   // size is 7 bytes
  {
    short s;
    long  l;
    char  c;
  };
};

void DummyFunction (char& aac[][3])   // a variable number of 3-character strings
{
  DummyUnion u;
  long i = 1;
  short as[8];  // array of 8 short elements, the resulting size is 16 bytes
  short as2[20];
  DummyUnion aaU[10][8];

  u.aac[i] = aac[4];  // accessing the parameter acc[4] may exceed the range!
                      // accessing u.aac[i] is range checked at runtime (access is
                      // limited to element 0 and 1)
  as = 0xAA;          // fill all 16 bytes of 'as' with value 0b10101010
  as2[10..17] = as;   // assignment to an 8 short long subset of as2

  aaU[3][3..5] = aaU[2][1..3];  // ok
  aaU[1..4] = aaU[5..8];        // ok
  aaU[1..4] = aaU[3,4];         // ok, the overlapped copy is handled correctly
  // aaU[3][3..5].s;  would cause an error, since it is not allowed to refine
  //                  specification (.s) after a range ([3..5])
  // aaU[1..2][2];    is not allowed for the same reason.
}
```

6.5.2 *Typecasting*

Variables of a certain type can be converted into other types explicitly by entering a type cast command:

(*typename*) *variable*

Explicit type casting can be used to suppress compiler warnings that would occur on implicit conversions.

Typecasts are possible in the following cases:

- The source type has the same physical size as the destination type. The source data itself will not be changed but handled as if it is of the destination type.

- Source and destination types are simple types. The source will be shrunk or enlarged to fit the destination type.

- Source and destination types are arrays with the same number of elements. The elements are converted like described before.

Other type conversions are not handled.

It may be helpful to use a typecast for specifying the data type of constant values. Since the data type of 0xffff can be either signed (-1) or unsigned short (65535), the result of an operation can be quite different, depending on the interpretation of the constant type. The OCL compiler takes all constants as unsigned values, as long as no negative sign occurs.

7 Constant Values

Constant values can be used instead of variables for read-only accesses. It is not possible to use constant values as destination of data. Constants may not be used as non-constant reference parameter to functions.

7.1 Simple Type Constants

The format of constant values of a simple type is described in section “Notation of Numbers” (3.1). These constants may be used as initial value, as parameter, as assignment source and even as operand of any calculation.

Constant values between 0x00000000 and 0x7fffffff have no fixed sign and may be used as signed or unsigned values, unless a negation (`-`) or type cast occurs. Values between 0x80000000 and 0xffffffff are assumed to be unsigned integers.

7.2 Character Constants

Character constants are single unsigned long values. A character constant is specified as follows:

‘ character ‘

The character may be any ASCII character as well as the special characters described below.

Some special characters can be specified inside of a string by a leading backslash:

newline (line feed)	NL (LF)	`\n`
horizontal tab	HT	`\t`
vertical tab	VT	`\v`
backspace	BS	`\b`
carriage return	CR	`\r`
form feed	FF	`\f`
alert	BEL	`\a`
backslash	\	`\\`
question mark	?	`\?`
single quote	’	`\’`
double quote	”	`\”`
octal number	***ooo***	`\`***ooo***
hex number	***hh***	`\x`***hh***

The octal and hexadecimal representation can be used to specify any character. Octal numbers may consist of up to three digits between 0 and 7; hex numbers may consist of one or two digits between 0 and f (you may use capital letters also).

7.3 String Constants

String constants are zero-terminated one-dimensional arrays of unsigned long values. A string constant is specified as follows:

“ *string* “

The string may be any sequence of characters (see 7.2 for information on non-ASCII-characters). The resulting array will contain the whole sequence with an additional zero byte at its end. The constant string source may be shorter than the destination without causing any warning or error.

A string constant may be handled like any other array.

```
...
unsigned long  szText[8] = “abcdefg”; // fill szText with ‘abcdefg\000’
szText = “abcdefghijklmnop”[2,8];     // fill szText with ‘cdefghij’
if (_memcmp (“bcdefgh”, szText))      // compare szText with ‘abcdefgh\000’
  { ... }
...
```

A string constant may be defined over multiple lines in two different ways:

- It may be ended with a double quote, intercepted by white spaces or comments and continued with a further double quote. All white spaces or comments between the inner double quotes are ignored; all other texts will cause errors.
- The string may be stopped at end of line by a single backslash and continued directly at the beginning of the next line. All white spaces in front of the second line are included into the string. It is not possible to place any ignored text inside the string with this method.

```
sz = “first part ”  // this is continued...
     “second part”; // ...here. sz contains now “first part second part”
sz = “first part\
      second part”; // sz contains now “first part       second part”
```

7.4 Complex Constants

Complex constants can be used as initial value, as parameter or as assignment source without problem. Since the type of complex constants is not determined non-ambiguous, it is not possible to use as operand of a calculation. In that case, you have to execute an explicit type cast on this constant.

It is possible to specify complete constant data structures (struct, union or array) by putting the desired values inside of braces.

7.4.1 Structures and Arrays

Since structures and arrays consist of multiple elements, it is necessary to define a list of all elements as follows:

{ *const_element* (, *const_element*)* }

This defines a constant structure or array consisting of the specified ***const_element***s. Any of the ***const_element***s may be a complex constant itself.

❖ **Example**

```
struct strConstDemo
{
  long l;
  unsigned long sz[6];
  struct
  {
    long  s;
    long  c;
  };
};

void ConstCall (const strConstDemo& p)
{
  // initialize static variable v.  v.sz is set to ‘GSEOS\000’
  static strConstDemo v = {0x12345678, “GSEOS”, {12345, 0b10101101}};

  // now set v to a new value. v.sz is set to ‘GSEOS5’ without trailing zero byte
  v = {0x87654321, {‘G’, ‘S’, ‘E’, ‘O’, ‘S’, ‘5’}, {1, 2}};

  ConstCall ({0xaffe, “call”, {0xcafe, 0xae}});
}
```

7.4.2 Unions

The definition of constant unions consists of the specification of the first sub-symbol’s value only, because all sub-symbols share the same memory and cannot be set independently.

{ ***const_element*** **}**

This expression defines a constant union with its first element set to ***const_element***. If the first element of the union is shorter than the complete union, the upper bytes of the union are filled with zero.

❖ **Examples**

```
union uniConstDemo
{
  struct
  {
    unsigned long sz[6];
    long  s;
    long  c;
  };
  long l;
};

void ConstCall2 (const uniConstDemo& p)
{
  // initialize static variable v.  v.sz is set to ‘GSEOS\000’. sz.l is not
  // accessible for union-constants
  static uniConstDemo v = {{“GSEOS”, 12345, 0b10101101}};

  // now set v to a new value. v.sz is set to ‘GSEOS5’ without trailing zero byte
  v = {{{‘G’, ‘S’, ‘E’, ‘O’, ‘S’, ‘5’}, 1, 2}};

  ConstCall2 ({{“call”, 0xcafe, 0xae}});
}
```

If you want to use complex constants in calculations, you have to specify the type of the constant using type conversion:

```
typedef long tal[4];
tal   al;
long  s;

al = s * (tal) {123, 456, 789, 101112};
```

This assigns the result of the array multiplication to `al`. Array operations like this and other data manipulations will be subject of the next chapter.

8 Data Manipulation

Data manipulation may be any calculation or re-arranging of arrays.

Internally all calculations are executed in four bytes long integer or eight bytes long floating point variables. The operands are converted into the according type before the operation.

8.1 Operations

8.1.1 Dyadic Operators

The following table lists all operators that need two operators. The resulting type of these operations is the smallest type that can store both operands.

	priority	$a \times b$	$a = a \times b$	$a = a \times 1$
Addition	3	a **+** b	a **+=** b	**++** a
Subtraction	3	a **-** b	a **-=** b	**- -** a
Multiplication	2	a ***** b	a ***=** b	
Division	2	a **/** b	a **/=** b	
Modulo	2	a **%** b	a **%=** b	
Power (**a**$^{\mathbf{b}}$)	1	a ****** b		
And (bitwise) (logical)	5 9	a **&** b a **&&** b	a **&=** b	
Or (bitwise) (logical)	7 10	a **|** b a **||** b	a **|=** b	
Exclusive or	6	a **^** b	a **^=** b	
Equal		a **==** b		
Not equal		a **!=** b		
Less than		a **<** b		
Less or equal		a **<=** b		

Greater than		a `>` b		
Greater or equal		a `>=` b		
Shift left	4	a `<<` b		
Shift right	4	a `>>` b		
Maximum	8	a `>?` b		
Minimum	8	a `<?` b		

The operands ***a*** and ***b*** may be simple types or even arrays of simple types. See the section 8.2 (Array Operations) for further information. Bitwise operations cannot be used on floating point values. The operands of logical operations are converted to boolean before execution (non-zero becomes true, zero values become false). The result type of logical operations and comparisons is boolean.

8.1.2 Monadic Operators

The built-in operators that need only one operand are shown in the following table.

	operator	annotation
Negation (numeric)	`-` a	result type is signed
Negation (bitwise)	`~` a	undefined on floating point values
Negation (logic)	`!` a	result type is boolean
Absolute value	`abs` a	result type is unsigned
sine	`sin` a	result is floating point
arcsine	`asin` a	result is floating point
cosine	`cos` a	result is floating point
arccosine	`acos` a	result is floating point
tangent	`tan` a	result is floating point
arctangent	`atan` a	result is floating point
natural logarithm	`ln` a	result is floating point
exponential e	`exp` a	result is floating point
decimal logarithm	`log` a	result is floating point
exponential 10	`10` `**` a	see 'power' operation in section 8.1.1 (Dyadic Operators)

The ***unary_expression*** “a” may be any single value or array (see section 8.2) as well as more complex expressions within brackets **( )**.

❖ **Example for Calculations**

```
{
  long l1 = 12345;
  long l2 = 24680;
  long l3;
  double d;

  l3 = l1 <? l2;       // l3 = 12345 (= minimum of 12345 and 24680)
  d = l3 * 0.321;
}
```

8.1.3 Random Values

Pseudo random values can be generated using the keyword

`random`

It returns a double value 0 ≤ x < 1. It is also possible to use this command to generate values between 0 and a specified value:

`random` **(** ***limit*** **)**

The result will be a random value of the same type as ***limit***. It will be inside the interval **[0, *limit*[**. This command can be used as well to generate arrays of random values. In this case ***limit*** has to be an array of the desired type, and its elements represent the upper limit for the value of the corresponding random element. That means, the limit of the random value may be specified separately for each element.

The random number generator can be initialized using the command

`seed` **(** **[** ***expression*** **]** **)**

This command sets the starting point of the random number generator to the value of ***expression***. If ***expression*** is omitted, the generator will be initialized with a value depending on the system time and a pseudo random number generated before the re-initialization.

❖ **Example**

```
{
  long l1 = 12345;
  unsigned long auc1[10] = “abcdefghi”;
  double d;

  auc1 = random (“123456789” – ‘1’); // no zero termination!!!
  d = random * 10;                   // 0 <= d < 10
  l1 = random (l1);
}
```

8.2 Array Operations

All mathematical and logical operations may be used on arrays as well as on single elements. This method is faster than a self-made loop over all elements.

An operation between a simple type value and an array will evaluate the operation on the value and each element of the array. The resulting array will contain the same number of elements as the input. The type of the elements is the same as the type of the result of the according non-array-operation.

An operation between two arrays will process the corresponding elements of both arrays.

Multidimensional arrays will automatically be flattened before processing. That means, that the result of an array operation will always be a one-dimensional array, regardless of the number of dimensions of the operands.

Remember: These array operations are no matrix- or vector-operations!

It is not possible to use functions that are defined for scalar parameters for processing whole arrays instead. This is a feature of built-in operations only.

❖ **Example**

```
{
  unsigned long auc1[10] = “abcdefghi”;
  unsigned long auc2[10] = “ihgfedcba”;
  unsigned long auc3[10];
  double   ad[10];

  auc3 = auc1 >? auc2; // auc3 = “ihgfefghi”
  auc3[0,5] = auc3[0,5] + (‘A’ – ‘a’);  // auc3 = “IHGFEfghi”
  ad = sin auc2;    // calculate the sine of the ASCII values
}
```

8.3 Assignment Operator

The assignment operator can be used for assignments of simple type values as well as arrays and complex data structures. Even the assignment of array ranges is allowed (and is much faster than subsequent accesses to single elements inside of loops).

Besides of this normal assignment of values to variables of the same type, the assignment operator = may be used to fill all bytes of a complex data structure (like a union, structure or an array) with a user defined value. This value has to be a character constant.

complex_data* = *constant_char

complex_data may be a union, structure or an array. All bytes of it will be set to the value ***constant_char***.

❖ **Example**

```
{
  unsigned char auc1[10] = “abcdefghi”;
  unsigned long auc2[10] = “ihgfedcba”;
  unsigned long auc3[10];
  unsigned long aul[10];

  auc3 = auc1 >? auc2; // auc3 = “ihgfefghi”
  auc3[0,5] = auc3[0,5] + (‘A’ – ‘a’);  // auc3 = “IHGFEfghi”
  auc1 = rand (“123456789” – ‘1’); // no zero termination!!!
  auc1[1,9] = auc1[0,9];  // move all elements one step up

  aul = 0xba;    // fill all elements with 0xbabababa
}
```

9 Functions

All functions represent UDPs. It is possible to use any desired UDP from within another one. Nevertheless, only UDPs that do not expect parameters can be started directly (via timeline or direct execution command), since the start mechanism does not provide parameters.

All normal compiler runs should know built-in functions by reading the symbol table file `symbols.tab` already, so you should not try to redefine these functions.

9.1 User Functions

Functions are handled very much like in C. The function definition syntax is the following:

type functionname `(` **(** ***type*** **[** `&` **]** ***parametername*** **)*** `)` **(** ***compound_statement*** **|** `;` **)**

The ***compound_statement*** may be any OCL code embedded in braces. It is possible to declare prototypes of functions by replacing the ***compound_statement*** by `;` . This is useful if you want to define functions that call a later defined function: Since at least the function header must be known when the call is to be compiled, use the prototype definition before. Please note that the function header of the later function definition must be exactly the same as the prototype declaration.

Recursive function calls are allowed.

Functions may be called by any other OCL function. The ***functionname*** is used to specify the desired function.

9.1.1 Return Value

Unlike C, OCL allows complex return types like structures or unions. If nothing is returned by the function, you have to specify the return type as `void`.

Although it is possible to use large complex types as return value you should remember that return values are stored twice on the local machine stack (created in the local memory of the called function and copied into the local memory of the calling function). Therefore you should avoid using large return types. It is not possible to use references as return type.

9.1.2 Parameters

Since there are no pointers in the OSIRIS command language, parameter transfer normally is done by copying memory into local function memory. This may result in a lack of performance, when large amounts of memory have to be copied (for example: bitmaps as parameters). To avoid extensive memory duplication, you may specify parameters as references. As referenced parameters are not copied into the local memory of the function, all modifications of reference parameters inside the function will take effect outside also. That means, you can use reference parameters for the bi-directional transfer of data to functions and back.

The declaration of a reference parameter is done by entering the reference operator `&` behind the type of the parameter. Please note that accessing reference parameters is slower than non-reference parameters. You should use reference parameters only for two reasons:

- Modifications of the parameter inside the function has to take effect outside
- The parameter contains a large amount of memory.

Empty brackets mean that there are no parameters transferred to the function.

It is not possible to define functions with a parameter list of a variable length.

It is possible to specify arrays of unknown length as parameter by leaving the square brackets empty (`[ ]`). If you do so, you must specify this parameter as a reference (using `&`). Please note that all modifications done inside the function will have effect to the array outside the function also.

❖ **Example**

```
void Increment (long& sr)
{
  ++sr;    // increment sr.
}

EndianUnion DemoFunction (long l, long& sr)
{
  EndianUnion Result;
  Increment (sr);
  if (l > 0)
    return DemoFunction (l-1, sr);  // recursive call

  Result.TwoUnsigned.ul2 = 1234;
  Result.s = sr;                    // store current sr in return value
  sr = sr * 2;
  return Result;                    // return
}

long Length (unsigned long& uc[])   // allow char-arrays of any length to be
                                    // transferred
{
  long l;
  for (l = 0; uc[l] != 0; ++l);     // run until the terminating 0 is found
  return l;                         // return the length. (note: this is not
                                    // the size of the array)
}

void main ()
{
  EndianUnion VarUnion;
  short       s;
  long        l;
  unsigned long uc[256];

  l  = 3;
  s  = 2;
  uc = "Demonstration";
  VarUnion = DemoFunction (l, s);   // after this call, the variables will contain
                                    // the following values:
                                    // l                         == 3
                                    // s                         == 12
                                    // VarUnion.s                == 6
                                    // VarUnion.TwoUnsigned.ul2  == 1234
  l = Length (uc);                  // after this call, l will contain 13
}
```

9.2 Non-OCL-functions (runtime library)

It is possible to use functions that are not written in OCL: You can use OSIRIS-built-in runtime library functions. These functions can easily be called from your normal OCL programs.

Please note that the interface between OCL and those external functions cannot provide reliable error detection and type checking, since the real parameter types of the non-OCL-functions are not known for sure by the compiler. A wrong declaration of external functions may cause fatal errors up to program crashes.

Normally, the compiler should know all built-in functions already (from the first compilation run of **`init.ocl`**, which contains the declaration of these functions). You may call any of these functions like any other (user-defined) function without additional declarations. If you have an existing symbol table file, you can skip the following section.

9.2.1 Initializing Built-in Runtime Library Functions

This section describes how to initialize the symbol table file with the headers of built-in functions. You should do this with maximum care, since corrupted symbol table information may lead to program crashes on DPU. Normally, there should be really no need at all for the user to initialize the symbol table, so most people may want to continue with the next section now.

If you really have to create a new symbol table, you have to initialize it with the built-in functions, before you compile UDPs. This initialization can be done very easily:

All you need is a file that contains the interface declarations of all built-in functions in exactly the same order of their appearance in the DPU's jump table. Make sure that the interface declarations fit the interface definitions used for the real functions of the DPU, since differences may cause fatal malfunctions! There is absolutely no automated cross-checking!

The interface declarations for built-in functions looks very much like prototype declarations of user defined functions (consisting of the return type, the RTL function name and the list of parameters). The only difference is the keyword **`extern`** in front of each declaration, like shown in the following example:

```
extern long Check (long lNoOfUnits, long lDelay);
```

To generate a new symbol table file you only have to delete the previous version of this file (if present) and call the compiler with the option **`-mc`** like this:

`ocl` **`-mc`** ***init.ocl***

The compiler will detect that there is no existing symbol table and will enter the initialization-mode automatically. The function interface definitions found in the file ***init.ocl*** will be processed and stored inside the new symbol table file.

After this initialization you may use any of the functions declared inside the file ***init.ocl*** without including this file again.

Normally the complete interface to the built-in functions should be provided with the OCL compiler (for instance inside the file ***init.ocl***). There should be no need for you to change the declarations.

Please note that differences between the OCL-declaration and the real needs of the built-in functions can cause fatal errors!

Since all built-in functions have to be declared as a whole at the beginning of the symbol table, it is not allowed to add further '**`extern`**' functions in later compiler runs. Any use of this keyword in compiler runs with an existing symbol table file will cause error messages!

10 Special Commands

10.1 Sleep

This command stops the execution for a specified number of milliseconds.

`sleep` ***time*** `;`

10.2 Wait

This command will stop the execution until the specified event occurs or the time exceeds the timeout value.

`wait` `(` `[` ***timeout*** `]` `,` ***event*** `)`

The ***timeout*** may be omitted. In this case the machine will never cancel the wait for the specified ***event***. The command returns a value, which specifies the reason of continuation:

• 0:	event occurred
• ***RC_TIME***:	timeout occurred

10.3 Signal

`signal` ***event*** `;`

This command will set the specified event.

10.4 Halt

This command stops the execution of the current UDP immediately. To continue execution, the according command has to be sent to the UDP manager by hand (see IDA-OCL-0003, sections 4.1.8 to 4.1.11).

The `halt` command is thought to be used for debugging purposes only. It should not be used inside regular UDP code.

10.5 Start

This command will start execution of the specified UDP within another token interpreter and continues the execution of the current UDP immediately.

`start` `(` ***UDP_Name*** `)`

The start command will only be successful, if the UDP's ID and name at execution time fit the ones found at compilation time. This behavior should prevent incorrect UDP calls.

The return value of the start command indicates the success:

- 0: an interpreter received the start command and will try to execute the UDP
- 1: the UDP does not exist (at least not with the former ID)
- 2: the UDP has been replaced by another one (with different name)
- 3: all interpreters are currently busy

Any other value than 0 indicates that the start command failed. Please note that this return value does not contain any information about success of the UDP execution (which probably is not finished at this time).

10.6 Size Of ...

The command **`sizeof`** can be used to derive the size of the memory used by a certain variable or data type.

`sizeof` can be used on array parameters of variable sizes. In this case the size of the referenced array will be derived at runtime.

Normally it is impossible to mix type-name and field-names of a data structure to specify an element. The only exception is inside the **`sizeof`** command: the first part of the specifier may be a type name, all following may be field specifiers.

❖ **Example**

```
long l;
l = sizeof (PreciseUnion.s);     // l = 2 (bytes), the size of the field s of the
                                 // type PreciseUnion
l = sizeof (BitPreciseUnion.s);  // l = 2, the size of the field s of the variable
                                 // BitPreciseUnion
```

11 Preprocessing

The OCL compiler built-in 'preprocessor' is not really preprocessing the input before compilation. It is rather a kind of 'text-processing in between': Whenever the parser comes across an unknown symbol, it activates the preprocessor to replace the symbol, if there is a replacement defined for it. After replacement, the compilation will continue.

Please note: Although you might define symbols that are already used by the compiler (like OCL keywords or data types, variables or function names), these replacements will never be used, because the text-processor will not be activated if the symbol is identified by the parser.

11.1 Comments

Text included in `/*` `*/` is ignored. Comments may be nested. If inside a comment a opening `/*` is found, the next `*/` will not end the comment mode.

Text following `//` is ignored until end of line. It is not possible to open or close a further comment there!

```
void main ()
{
  /* this is a comment.
    /* this is a comment inside an other comment. */
     this is still comment.
  */
  return; // last comment
}
```

Please note that `/*` or `*/` placed after a `//` are ignored. Therefore the following example would cause errors:

```
void main ()
{
  /* this is a comment.
    /* this is the inner comment // and even more commented. */
     this is still inner comment!
  */
  now the inner comment is closed, but the outer comment is still active.
  it is closed here: */
  return;
}
```

11.2 Include

It is possible to divide the OCL startup-file into several pieces and put them together during compilation via `#include` command. Whenever the `#include` command appears the compiler switches to the text of the specified file and continues compilation. After compiling the included file, the compiler switches back and continues compilation of the previous file.

`#include` < *filename* >

or

`#include` `"` ***filename*** `"`

Any text behind will be ignored until end of line is reached.

11.3 Define

You may define your own keywords by using the `#define` command

`#define` ***keyword text***

Whenever the compiler reaches an unknown symbol during compilation, it checks if the symbol is defined as `keyword`. If it is found, it will be replaced by ***text***. The keyword remains defined and may be used as long as the compiler is running or it is undefined by using the preprocessor directive

`#undef` ***keyword***

Please note that it is not possible yet to define complex macros with parameters.

text may be more than one line. In this case use `\` as the last character of the line that is continued in the next line.

11.4 Ifdef ... else ... endif

An easy way to switch between alternative source codes is using the `#ifdef` keyword. If the symbol behind `#ifdef` is already defined, the compiler compiles the code following directly until the next `#else` or `#endif` command. The code between `#else` and `#endif` is ignored. If the symbol is not defined yet, the code following the `#ifdef` is ignored and the code between `#else` and `#endif` is compiled instead. The `#ifndef` command has the opposite behavior.

❖ **Example**

```
#define MSG_INFORMATION         0
// define DEBUGMODE for creating additional debug information
#define DEBUGMODE

...

// if in DEBUGMODE create additional message
#ifdef DEBUGMODE
  _AddMessage (MSG_INFORMATION, "DEBUG", "debug point 1");
#endif

#ifndef DEBUGMODE
  _AddMessage (MSG_INFORMATION, "Info", "debugging mode off");
#else
  _AddMessage (MSG_INFORMATION, "Info", "debugging mode activated");
#endif
```

12 Compiler Messages

This section offers a short description of the compiler and virtual machine messages in combination with hints for eliminating the reason for these messages.

We have to differentiate between messages that appear at compile time and messages that appear during runtime:

- Compile time messages contain the filename and line number of the input that causes the message. Only a few messages are not associated to line numbers.
- Runtime messages are returned via housekeeping data.

12.1 Warnings

Warnings show potential errors. Since some of these warnings may be caused deliberately, it is possible to suppress some types of warnings to keep the message list clear (see). You should not suppress all warnings, since some of them could indicate runtime problems.

`Array size unknown - be sure not to exceed it!` This means, that you are accessing a fix element of an array of a variable size. This may cause runtime errors if the accessed element is outside the array limits.

`Assignment in condition`: This warning indicates possible accidental use of an assignment (**`=`**) instead of an equality comparison (**`==`**) as a condition.

`Assignment of different enumeration types`: This assignment may lead to a value of the destination that is out of range.

`Assignment to enum - take destination as int`: This assignment may lead to a value of the destination that is out of range.

`Cast may truncate significant bits`: The operation causes an implicit typecast. The new type is smaller than the old type, which leads to loss of information. This warning appears for example at assignments of long to char.

`Conversion causes bit loss` indicates that you are assigning elements with different sizes.

`Conversion of boolean to numeric`: You are using a boolean variable or value as a numeric.

`Conversion of const unsigned to signed`: You are using an unsigned constant value as a signed value. This warning only appears if the value uses full 32 bit (like 2147483648, which is 0x80000000).

`Conversion of negative const to unsigned`: This message appears when you are assigning a negative constant value to an unsigned variable.

`Conversion of unsigned to signed` means, that you are using an unsigned variable as a signed. This may cause large values to appear as negative. Remember that all string and character constants are used as (arrays of) unsigned characters.

`Different array sizes, taking minimum size`: An operation on arrays with different length occurred. To prevent memory access errors, the operation will stop after the smaller array is processed completely. Some elements of the larger array will not be processed.

`External buffer partly unreachable`: You have declared an extern variable that is smaller than the memory area provided by the hardware driver. That means that you will not be able to access the complete memory area.

`Float truncating to int`: The floating-point value is truncated to the lower next integer value.

`Ignoring declaration as unsigned`: Floating point variables cannot be declared as unsigned.

`Ignoring incrementation on float`: The increment or decrement operators are not defined on floating point values.

`Ineffective code`: You add/subtract a constant floating point zero or multiply/divide by a constant floating point one. This operation may imply a type cast without affecting the value of the second operand. Maybe this cast is not necessary. This operation primarily is thought to force the result of QLook Items to be floating point.

`Large parameter copying`: You are using a large data type as non-reference parameter. This may be ineffective, because the parameter has to be copied completely, and it may require much space on the limited local stack.

`Low remaining stack`: The local memory used by the function is very big. It may cause runtime errors with recursion or deep call stacks.

`Modulo on floats is not defined, truncating`: The operation modulo (%) cannot be used for floating point values. The value is truncated to integer before the operation.

`Name conflict on resolving external`: You declared a symbol with a name that is also used as a function in the runtime library. In this case the runtime library function is not accessible.

`Non-existing external`: The runtime library contains a function that is not declared in the source code. That means, the function is not accessible for OCLs.

`Parameter has no identifier`: The parameter of a function has no name and cannot be used inside the function.

`Redefining with different value`: You are defining a symbol that is already defined. You should use `#undef` before the second definition to suppress this warning.

`Result is always zero`: The calculation is redundant, since its result is a constant zero.

`Signed interpreted as unsigned`: Negative values may be interpreted as large positive values.

`Source array has a variable size - using size of destination`: This assignment may cause runtime errors if the source is smaller than the destination. You should make sure that the source array is always larger than the destination.

`Stack cannot hold local memory`: The currently set stack size is too small to hold all local memory needed by the compiled function. This will cause runtime errors as soon as the function is called. You should increase the stack size or decrease the hold-local-limit in the compiler control panel (the latter takes effect after a new compiler run).

`Too many warnings. Suppressing further messages`: The currently executed module has reached the maximum number of warnings. To keep the message file readable, all following warnings of this module will be suppressed.

`Types differ with same size, copying`: This warning may appear if you are assigning an array of unsigned values to an array of signed values. Although the type is changed, the assignment is done. Remember that there are many cases that result in strange values of the destination variable (for example: you might be assigning a complex structure to an array of floating point values).

`Types do not fix exactly`: You are assigning a type to a slightly different data type.

`Unknown preprocessor directive`: The preprocessor doesn't know the used directive. Currently you can use `#include`, `#define`, `#undef`, `#ifdef`, `#ifndef`, `#else` and `#endif`.

`Unresolved function`: You declared a function to be extern that is not part of the runtime library, you declared a function to be part of an DLL that is not found or you declared a prototype of a function without defining the body. Any call of this unresolved function will cause runtime errors. The name of the function is shown in the warning message.

`Unsigned interpreted as signed`: An unsigned value is used as signed. This may cause large values to be interpreted as negative.

`Useless operation`: You are adding/subtracting a constant zero or you are multiplying/dividing by a constant one. Since this operation does not affect the result, it will be ignored.

`Using enumeration value as integer`: This is non-critical, since enumeration values always are internally represented as long integers.

`Using numeric value as boolean`: You are using a numeric value instead of a boolean. This message may indicate accidental use of logical operations instead of bit operations on numeric values. It may also indicate an erroneous use of a numeric value inside a conditional expression. Since all non-zero values are interpreted as true, this may cause unexpected results.

`Using temporary value as referenced parameter`: The value of a parameter may change inside the function, but it will have no effect to outside.

12.2 Errors

In very rare cases (actually there is no known case at present) internal bugs of the compiler may cause the error messages. If you are sure that your program is correct although there are error messages, you should take a look inside the next section (Fatal Errors) to find tips to create workarounds on compiler bugs.

`Array limits exceeded`: You are trying to access an element with a too high or negative constant index. This would cause unexpected results at runtime.

`Array with unspecified size has to be reference`: If you want to use arrays with an unspecified number of elements as a function parameter, you have to specify it as reference using `&`.

`Assignment to constant`: You are trying to assign a value to a constant.

`Assignment of different types`: You are trying to assign different types. This is only possible for simple types (bool, char, short, long, double) or types of the same size. Since the type checking is not based on type names, the message does not show the type names but the top-level type (a simple type, `ARRAY`, `UNION` or `STRUCT`). Although the top-level type may be equal, the difference may appear in lower levels.

`Cannot assign two arrays of variable size`: The sizes of the source and the destination array are not specified. It is not possible to derive the number of bytes to copy.

`Cannot handle this type here`: You may be trying to use a simple-type operation on a complex data structure. The resulting type is set to 'unknown', which may cause following `Unknown Type` – errors.

`Cannot return a value here`: Decoders cannot return a value. Use an empty return instead.

`Cannot use references here`: References only may be declared as function parameters.

`Cannot use variable array size here`: You are declaring an array with an unspecified number of elements in an illegal context. Arrays with an unspecified number of elements are allowed only as (referenced) parameters of functions or in unions with at least one element of known size.

`Changing preprocessor symbols`: You are using `#define` or `#undef` directives inside a QLook expression. This is not allowed, since the compilation order of expressions at startup is not defined.

`Constant expected`: You cannot use a variable here.

`Declaration does not fit prototype`: The function header differs from the prototype header specified before.

`Division by zero`: Since the division by zero is not defined, this operation is ignored.

`External buffer too small`: A external variable is larger than the memory area with the same name provided by the hardware driver.

`File not found`: The specified file does not exist.

`Global vars not allowed`: OCL prohibits the declaration of global variables, since this would cause complex timing behavior in combination with multiple parallel accesses.

`Illegal character`: You are using a character that is not allowed here.

`Illegal implicit cast on reference`: The type of the parameter differs from the function declaration. Since a typecast is not possible on references, the parameter cannot be used. Please note that parameters (like all internal variables) are stored in Intel format.

`Illegal local use of keyword`: You are using a keyword in a forbidden context.

`Illegal use`: You are using a non-function symbol as a function call.

`Illegal use of global variable`: Global variables cannot be initialized.

`lValue expected`: You are trying to use a constant as a parameter that is a reference (declared with `&`) or you are trying to assign a value to a constant. Use a non-constant variable instead.

`Modules must return a value`: explicit triggered QLook-interface modules have to return a value.

`Modulo on float undefined`: The modulo-operation cannot be used on floating point variables.

`No fitting loop`: You are using a `cont` or `break` statement outside a loop.

`Not in global scope`: The use of some keywords (like `extern` and `dll`) is forbidden in local scopes. Please note that all QLook-items as well as batch files are completely embedded by a local scope.

`Not in single expression mode`: Probably you have not specified a function header before the braces `{ }`. Since this is the syntax of the automatically triggered QLook Items, it causes an error if used elsewhere.

`Only the first dimension of an array may be variable`: You are trying to declare an multidimensional array with an unspecified number of elements in any other than the first dimension.

`Out of memory`: This is a runtime error of the virtual machine. The execution of the code will be aborted immediately.

`Parse error`: The parser cannot find a fitting grammar rule.

`Positive constant value expected`: Probably you are using a negative value or a variable.

`Preprocessor stack exceeded`: You are including too deeply nested files, or you are using too deep nested symbols. Perhaps you are recursively including a file or defining a symbol.

`Pure virtual function called`: You are calling an external function that is not linked. Check for '`Unresolved external`'-warnings and eliminate it.

`Returning different types`: The type of the symbol that you want to return differs from the declared return type of the function.

`Size exceeded`: You are trying to specify a bit-size of a type that is larger than the internal size of this type.

`Stack overflow`: The stack is too small to hold all local data needed by the virtual machine. Increase the stack size that is shown in the compiler control panel (this takes effect immediately) or decrease the size limit of local hold symbols (this takes effect after a recompile).

`Symbol already declared`: There is a symbol with the same name already declared inside the same scope. Remember that it is not possible to use the same name for a variable and a type at once.

`Symbol is not a member`: The structure or union does not contain the specified element. Remember that the compiler is case sensitive.

`Syntax error`: See explanation in the message itself. If there is no explanation, see the following messages for further information.

`Too few parameters`: The function call has fewer parameters than the function declaration.

`Too many errors - aborting`: The maximum number of errors is reached and the current (compiler- or virtual machine-) execution is aborted. The code will not be executed any more. This number can be adjusted in the compiler control panel at runtime. If you increase the maximum number of errors at runtime, code that already had reached the former maximum number of errors can be revived – until the new number of errors is reached again.

`Too many parameters`: The function call has more parameters than the function declaration.

`Too many single expressions`: The source contains more than one QLook Item function.

`Undeclared symbol`: You are using a symbol that is not declared yet. Place the symbol declaration before the first use. Remember that the compiler is case sensitive. Maybe you are using a formerly known preprocessor symbol that is not longer defined (perhaps you deleted the symbol using the compiler control panel?).

`Unexpected end of file`: Probably there is a comment not closed.

`Unknown Type - ignoring operation`: This is an aftereffect in most cases.

`Unresolved external variable`: The specified variable name is not provided by any hardware driver for access to mapped memory.

`Use does not fit declaration`: You are using a parameter of a different type as specified in the declaration of the function.

`Use of 'extern' not allowed here`: Only runtime-library functions may be declared as external.

12.3 Fatal Errors

These messages indicate internal compiler errors that are not caused by the user. Nevertheless they may be caused by the error handling of the compiler (aftereffect of a user error). This means that the fatal errors should disappear in most cases, if the user program is corrected.

If you get a fatal runtime error by the machine, this possibly indicates that the machine itself or the compiler is defective.

Although bugs of the compiler or virtual machine cause fatal errors, it may be possible to find workarounds by changing the user program. For this reason, you will find some hints on effective changing your program. In some cases it also may be helpful to force the creation of internal help variables by adding useless calculations like (...**`+1-1`**). Sometimes it is possible to find a workaround by inserting a ‘senseless’ statement (like **`1;`**) before (or even inside) the part of your program that causes the error.

`Cast error`: The compiler cannot perform an implicit cast. This is definitively a bug inside the compiler and should not occur. Perhaps you can create a workaround by using local variables for provisional results. Possibly it would help to use explicit typecasts.

`JumpStack corrupted`: The internal table of jump-addresses for loops or conditional expressions (like if...else, switch...) is corrupted. As a workaround you may try to eliminate some complex loops or conditional expressions.

`Memory error`: The managing of local help variables detected a malfunction. As a workaround you may try to eliminate implicit help variables by using local user-defined variables (instead of `a[b*c]...` try `d = b*c; a[d]...`).

`Not implemented`: At compile-time, this message indicates that you are using a rule of the OCL grammar that is not implemented yet. Your source probably will be correct in a future version, but you have to reword the statement if you want it to get compiled now (for example: Use **`if`** ... **`else`** ... instead of ... **`?`** ... **`:`** ...). If this error occurs at runtime, it may indicate that your program contains illegal memory write commands that destroy the code. The compiler or the virtual machine should detect almost all cases of illegal memory accesses, but possibly you found a technique to avoid this detection.

`Stack corrupted`: Runtime error. The code generated by the compiler has incorrect stack handling. As a workaround you may try to eliminate some function calls.

`Strange`: This internal error is really strange.

`Symboltable inconsistency`: The symbol table does not contain the expected information.

`Type not implemented here`: If there are no preceding errors, this message indicates a bug inside the compiler. Otherwise this message may result from former 'unknown symbol'-errors.

`Value can't have an address`: Definitely a compiler bug – without any known workaround.

13 List of Keywords

The following list contains the keywords of OCL that may not be used as variable or function names:

abs, acos, asin, atan, at
bool, break
case, const, continue, cos
default, do, double
else, enum, exp, extern
false, fixed, for
if, int
ln, log, long
random, return
seed, signed, sin, sizeof, signal, sleep, static, struct, switch
tan, true, typedef
union, unsigned
void
wait, while

14 OCL Compiler & UDP Manager

The OCL Compiler contains the ground part of the UDP manager. It generates files that can be uploaded directly to DPU via service 6.

All functions (UDPs) that are declared once will remain declared in all later compiler runs as long as they are not removed explicitly from the symbol table. Opposite to that, declarations of data types are not stored for later compiler runs. That means, you should store all commonly used data type declarations in one file that can be included by all source files that make use of these types.

14.1 Configuration

Configuration of the OCL compiler is done using a special file. The default name of this configuration file is `OCL.ini` (inside the current directory). The name of this file can be altered by using the command line parameter `-cfg` ***filename*** as first parameter or by setting the environment variable `OCLCONFIG` to the desired name and path.

If OCL does not find the specified configuration file it will create it (filled with default values). If you already have this file with only some options missing, you can call OCL with the parameter `-write.ini` to add all missing options with their default values.

14.1.1 Search Path for Including Files

The directories to be searched for files that should be included (`#include`) can be specified with the keyword `SearchPath:`. Particular directories have to be separated by `;` (semicolon). The search path should (at least) contain the current directory `.` (dot).

The search path can also be specified by using the environment variable `OCLSEARCHPATH`. The value of this variable is appended to the string specified inside the configuration file.

14.1.2 Aborting After a Couple of Errors

The option `MaxErrors:` inside the OCL.ini file allows to specify the number of compile errors that may occur before aborting the compilation. Please note that the resulting number of error messages will be value+2 (since value+1 is the first message exceeding the limit and value+2 is the `too many errors` message). The default is `MaxErrors: 7`.

14.1.3 Hold Local up to Size

To allow the use of large local variables in combination with a strictly limited stack size, the compiler stores variables that exceed the size specified with the keyword `HoldLocalSize:` in dynamically allocated memory.

Variables that are smaller than the specified size are stored on the local stack, variables larger than the specified size are stored outside the local stack automatically. The syntax of accesses to non-local variables is exactly the same as of accesses to local variables.

Accesses to variables on the local stack are faster than accesses to dynamically stored variables. Setting the limit too low will cause loss of performance while setting the limit to high may result in stack overflow errors. The default value is `HoldLocalSize: 8`.

14.1.4 Warning Message Suppression

Generation of warning messages (see 12.1) can be suppressed via `OCL.ini` like follows:

`Warnings:` **(** `¿` **((** `+` **|** `-` **)** ***WarningText*** `¿` **)*** `)`

WarningText is the message text, `-` deactivates the warning. All entries of this list have to stand in a new line. After the first run of the OCL compiler, `OCL.ini` contains a complete list of all warnings.

14.1.5 Debug Information

The OCL compiler can store additional debug information into a file. Which information is to be stored can be specified in the `OCL.ini` file using the following keywords:

- `Debug:` **(** `yes` **|** `no` **)** generate file or do not generate debug file at all
  - `ShowCode:` **(** `yes` **|** `no` **)** show the generated token code
    - `ShowFirstPass:` **(** `no` **|** `yes` **)** show the code even of the first pass (`ShowCode:` has to be set to `yes` in this case)
  - `ShowSymbols:` **(** `no` **|** `yes` **)** show the symbol table
  - `ShowTemp:` **(** `no` **|** `yes` **)** show allocation of internal variables for temporary results

Normally, only the first two options should be set to yes. The other three options are more useful for internal compiler debugging. The name of the file can be specified inside the `OCL.ini` using the keyword `DebugFile:`. The default name is `OCLdebug.txt`. Normal compiler runs append their data to this file, only the initial compiler run (without an existing symbol table file) will replace an existing debug file.

14.1.6 Command File

The filename of the command sequence generated by the OCL compiler can be specified using the keyword `CommandFile:`.

The default filename is `command.bin`. You may use the wildcard `*` inside this specification. This causes the compiler to replace this character by a generated text depending on its current action and take the resulting string as output filename. Please note that some special characters may be replaced by underscore (`_`) for file name creation.

14.1.7 Symbol Table File

The name of the file containing the symbol table (see 14.6.2) is specified using the keyword `SymbolFile:`. If the OCL compiler does not find that file, it assumes to be called the first time and enters the initialization mode. The default file name is `OCLsym.txt`.

14.1.8 Log File

The keyword `LogFile:` specified the name of the file that contains all logging information. All compiler runs append their output to the existing file.

14.1.9 Compression Table

The compiler uses a user-defined compression table, if the filename is specified behind the keyword `CompressionTable:`. If this keyword is not specified, the built-in table is used to generate compressed token code.

For further information about the compression and the format of user defined compression tables see document IDA-OCL-0004.

14.1.10 Preserving UDP-IDs

You can make sure that certain UDPs get certain IDs:

`PreservedIDs:` `(` `¿` **(** ***ID UDP-Name*** `¿` **)*** `)`

The ID has to be larger than the number of built-in functions. It is not possible to preserve IDs that are already used by other UDPs.

14.2 Compilation and Handling of UDPs

The OCL compiler should be used to compile UDP source code to token code as well as to generate commands for upload and deletion of UDPs and for job control.

14.2.1 Initialization

If no existing symbol table file is found, the compiler enters the initialization mode automatically. Only in this mode it is allowed to specify "external" functions. Since these functions are built-in to DPU, external declarations only affect the ground based symbol table. It is important to use the correct declaration order at this point, because all calls to functions are handled using their IDs that depend on the declaration order.

If a symbol table file (see 14.6.2) is found, it is not allowed to declare any other function to be extern any more.

The compiler run for initialization looks like any other call for compilation of multiple UDPs, but the source file should only contain all declarations of external functions. The declared functions will remain declared in all later compiler runs, as long as they are not deleted explicitly from the symbol table file.

`ocl` `-mc` ***declarationsFile***

The ***declarationsFile*** could be named `init.ocl`, for example.

If you use the wildcard `*` inside the command file name, it will be replaced by the name of the ***declarationsFile*** (without the extension, e.g.: `init.ocl` will result in `init`).

14.2.2 Compiling a UDP for Execution on Startup

To generate a UDP which is executed automatically on startup of the OCL space segment, no special compiler option is needed. The UDP Manager initiates the execution of the POP with the lowest possible ID, if it is named `AutoStart` (case sensitive) and does neither expect parameters nor returns any value. So, to create this startup-UDP, the following procedure should be used:

1. Initialize the ground symbol table (see 14.2.1, `ocl` `-mc` ***declarationsFile***)
2. Optional (but recommended): determine the first unused ID from the generated ground symbol table file and preserve this ID for the UDP `AutoStart` (see 14.1.10). For the DPU Software version 7.03 the ID of `AutoStart` has to be 99.
3. Compile the file containing the UDP `AutoStart` (see 14.2.3). If you left out step 2, `AutoStart` must be declared as first UDP inside of this file (`void` `AutoStart` `(` `)` `;`). Please be aware that `AutoStart` cannot access any POP before it has been loaded explicitly by calling the appropriate RTL function (`LoadPOP`). Only RTL functions are available immediately.
4. Send the token code to the UDP Manager.
5. Save (at least) the UDP `AutoStart` into NVRAM (see 14.3.1).

That's all. Step 2 is recommended, because it allows easier deletion and replacement of the `AutoStart` UDP. Please be aware that saving POPs destroys existing POPs with higher IDs. That means, re-saving only an updated `AutoStart` to NVRAM may invalidate the following UDP library.

14.2.3 Compiling Multiple UDPs at Once

14.2.3.1 Input From File

Sometimes it may be useful to create multiple UDPs at once. To do this, you can create a file containing all desired UDPs (functions) and compile it using the option `-mc` (multiple compile). On success, the compiler generates the code for upload to S/C and appends it to the file `command.oct`.

Additionally, all defined functions will be added to the symbol table file `symbols.tab` for use in later compiler runs. Of course it is allowed to include other files and define own data types.

The compiler call is as follows:

`ocl` `-mc` ***sourceFile***

Normally, the ***sourceFile*** should have the filename extension `.ocl`.

Of course, this can be used even to compile files that contain only a single UDP.

If you use the wildcard `*` inside the command file name, it will be replaced by the name of the ***sourceFile*** (without the extension).

14.2.3.2 The generated UDP token code is compressed. If you want to generate uncompressed UDP token code, you should use `-mcu` instead of `–mc`.Input From Standard Input

It is even possible to compile input from the standard input device.

`ocl` `-stdin` can be used if the output file name specified in `ocl.ini` contains no wildcards, otherwise you have to specify the name of the output file explicitely:

`ocl` `-stdin` ***outputFileName***

The input will be piped into a temporary file. Compilation will start not before the standard input reaches EOF.

14.2.4 Compiling One UDP for Execution at Certain Time(s)

This is done by using the option `-tc` (time line compile).

The command `ocl` `-tc` ***"code"*** **(** ***time*** **)*** can be used to generate a UDP that should be started at the given time(s). Afterwards, the UDP will be removed automatically from DPU memory. The ***"code"*** must not have the normal function header (return type, title and parameters) and should not return any value. It is supposed to contain only one or a few calls to more complex UDPs.

time is specified in seconds and corresponds to the time of the DPU. You may specify fractions up to milliseconds.

The generated UDP will get a name consisting of the UDP ID and the first few characters of the source code. The name will be displayed on success. On upload, the UDP will be automatically inserted to the timeline for execution at all specified times. The UDP will be executed as often as times are listed. After that, the UDP is deleted automatically from DPU memory.

If you use the wildcard `*` inside the command file name, it will be replaced by the generated name of the UDP followed by the keyword `Time`.

14.2.4.1 Example

Assume that slot 9 is the first free UDP slot.

```
ocl –tc "sleep (10)" 123.456 234.567
```

This line will create a UDP named `009:sleep_(10)` that is stored in the file `009_sleep__10_Time.bin`. The UDP will be executed two times (at 123.456sec and 234.567sec) and deleted afterwards.

14.2.5 Compiling One UDP for Immediate Execution

Using the option `–xc`, (execution compile) the compiler will generate code for a UDP that will be executed directly after upload and it will be removed after execution. The calling syntax is the following:

`ocl` `–xc` ***"code"***

"code" must not have the normal function header (return type, title and parameters) and should not return any value. It is supposed to contain only one or a few calls to more complex UDPs. The generated UDP will get a name consisting of the UDP ID and the first few characters of ***code***. The name will be displayed on success.

If you specified a command file name with a wildcard, asterisk is replaced by the generated UDP name followed by `Exec`.

14.2.6 Replace an Existing UDP

If you want to replace the code of an existing UDP by another, you should use the option

`-rc` ***UDP-name UDP-file***

UDP-name specifies the UDP that has to be replaced, ***UDP-file*** is the name of the file containing the new source code of the UDP. The interface of the UDP (name, return type and parameters) has to remain exactly the same as before.

Uploading the replacement code does not change time line entries. Replacing UDPs is possible only if the affected UDP is not currently running, so the UDP manager will wait a couple of seconds for the UDP to finish (if necessary). During this time, calls to this UDP will fail. Afterwards, all calls to the replaced UDP are handled like before.

If you specified a command file name with a wildcard, asterisk is replaced by the generated UDP name followed by `Replace`.

Please note that the replacement command is generated for a specific timestamp. It cannot be used to replace any other version of the same UDP with a different timestamp. This should prevent mistakenly replacement of UDP versions.

14.2.7 Execute a Certain UDP

You may initiate the execution of an existing UDP that does not need any parameters by using the option `-run`.

`ocl` `-run` ***UDP-name*** generates a command that starts the execution of the named UDP directly after upload. Of course, the mentioned UDP has to be present at the DPU's memory already.

If you specified a command file name with a wildcard, asterisk is replaced by the UDP name followed by `Run`.

14.2.8 Remove a Certain UDP

Sometimes you may want to remove a UDP from DPU by using the option `-del` ***UDP-name***. This generates a command sequence that causes the DPU UDP-manager to delete the UDP specified by ***UDP-name*** and remove all associated entries in the time line. Please note that only UDPs that are not currently running can be deleted. Nevertheless, the UDP manager will wait a couple of seconds for the affected UDP to finish.

If you specified a command file name with a wildcard, asterisk is replaced by the UDP name followed by `Del`.

Please note that the generated command sequence can only be used to delete the UDP with the timestamp found in the symbol table at generation time. The deletion will fail if the timestamp differs. This should prevent mistakenly deletions caused by unintentional later repetition of the command sequence (which could affect a different UDP with coincidental the same ID). If deletion fails, you will get a message containing the on-board timestamp of the UDP. If you are sure about what you are doing, you can correct the ground based symbol table's UDP timestamp (see 14.6.2) and generate a new deletion command sequence.

14.3 POP Handling

POPs are UDPs that are stored in the DPU's NVRAM. You can easily create POPs from any UDP in DPU memory. POPs are useful to avoid wasting transmission time for uploading large amounts of UDP data to DPU from earth after a reset.

14.3.1 Save UDP to NVRAM

It is possible to save UDPs in sequential order to NVRAM. You can generate the corresponding command using the OCL compiler with the option `-save` followed by one or two parameters:

- `ocl` `-save` `all` saves all UDPs at once to NVRAM. Existing POPs will be overwritten in NVRAM. Behind the last written UDP an EOF marker will be written to indicate the end of the used NVRAM area (useful for later POP operations). POPs behind the EOF marker will not be accessible any more.
- `ocl` `-save` ***UDP-ID*** saves only the specified UDP to NVRAM. A possibly existing POP with the same ID will be lost, existing POPs with a higher ID may be overwritten. This command

does not write an EOF marker, so POPs with higher ID remain accessible (unless they have been overwritten by the new POP).

- `ocl -save` ***first-ID last-ID*** saves all UDPs from ***first-ID*** up to ***last-ID*** to NVRAM. All POPs with an ID from ***first-ID*** up to ***last-ID*** will be lost, POPs with a higher ID may be overwritten. An EOF marker is written only if last-ID exceeds the highest ID of currently available UDPs. In that case, all previously existing POPs from ***first-ID*** on will be made unusable.

Please note that the UDPs are stored with the current values of its static variables.

If you replace an existing POP with a larger version, following POPs may be partially overwritten. That means, you will not be able to restore the overwritten POP from NVRAM and probably you will receive error messages during POP state checking or POP loading. Nevertheless, all untouched POPs behind will remain usable, and the (partially) destroyed POPs cannot corrupt the on-board symbol table or virtual machines.

If you specified a command file name with a wildcard, asterisk is replaced by **`SavePOPs`**, **`SavePOP_`*****UDP-ID*** or **`SavePOP_`*****first-ID*****`-`*****Last-ID*** (depending on the used option).

14.3.2 Load POP from NVRAM

UDPs stored in NVRAM can be restored into DPU memory. The appropriate command can be generated with the OCL compiler:

- `ocl -load all` reads all POPs from NVRAM. POPs that are already present as UDPs in DPU RAM remain untouched. If you want to restore all saved POPs you should make sure that all IDs of the saved POPs are currently unused. This option may be especially useful after a UDP manager reset.
- `ocl -load` ***POP-ID*** reads the specified POP from NVRAM into DPU RAM. A possibly existing former UDP with the same ID will be lost. If the interface of the existing UDP differs from the POP's interface, the UDP remains untouched and the load fails.
- `ocl -load` ***UDP-name*** replaces the existing UDP with its pendant from NVRAM, if the interface matches.

Please note that restoring POPs from NVRAM may cause inconsistencies between the DPU-based symbol table and the ground-based table. Note also that loading from NVRAM will restore the states of all static variables of the restored POP to their values at saving time. If you want to restore the initial values you have to save the UDP before its first execution.

If you specified a command file name with a wildcard, asterisk is replaced by **`LoadPOPs`**, **`LoadPOP_`*****POP-ID*** or **`LoadPOP_`*****UDP-name*** (depending on the used option).

14.3.3 Check POPs in NVRAM

The current state of the NVRAM can be checked by using

```
ocl -POPstate
```

The resulting command causes the UDP manager to return a table of contents of the NVRAM. The integrity of each found POP is checked by a checksum.

14.4 Job Control for Debugging

The OCL compiler can be used to generate job controlling command sequences. Please note that all following job control commands affects all token interpreters currently executing the mentioned UDP as top-level UDP. You cannot control a certain token interpreter. (This should be not real problem, since in most cases all interpreters execute a unique top-level UDP.)

14.4.1 Pause a Certain UDP

For debugging purposes it is possible to pause the execution of a certain top level UDP by using the option `-stop` *UDP-name*. The generated command will stop the execution of the specified UDP. Please note that all currently running instances of ***UDP-name*** will be stopped. Please note also that only top level UDPs can be stopped explicitly, but UDPs that are called by a stopped UDP will stop also.

If you specified a command file name with a wildcard, asterisk is replaced by the UDP name followed by `Stop`.

14.4.2 Resume a Certain UDP

After stopping a UDP, you probably may want to resume the execution of this UDP. Therefor you can use the option `-cont` *UDP-name*. This command will cause the UDP to return to normal execution mode. Please note that all current instances of ***UDP-name*** will be resumed. Please note also that only top level UDPs can be resumed explicitly, but UDPs that are called by a stopped UDP will be continued also.

If you specified a command file name with a wildcard, asterisk is replaced by the UDP name followed by `Cont`.

14.4.3 Execute a Single Command of a Certain UDP

If you want to execute the UDP step by step, you can initiate the execution of the next OSIRIS command token by using the option `-step` ***UDP-name***. Please note that stepping a UDP that is currently executed by more than one interpreter will cause unpredictable effects.

If you specified a command file name with a wildcard, asterisk is replaced by the UDP name followed by `Step`.

14.4.4 Quit a Certain UDP

If you want to abort the execution of a certain top level UDP, you should use the option `-quit` ***UDP-name*** to generate the appropriate command. Please note that the generated command will abort the executions of all instances of ***UDP-name***.

Quit will do a soft abort of the UDP on the next token fetch. That means, called external or system functions (like `sleep`) will not be interrupted. Possibly called sub-UDPs will be left directly via the normal returning mechanism and clean up properly.

If you specified a command file name with a wildcard, asterisk is replaced by the UDP name followed by `Quit`.

14.5 Customize DPU-Based UDP Manager and Token Interpreter

Commands for customization of the DPU-based part of UDP manager and token interpreter can be generated easily using the OCL compiler.

14.5.1 Set UDP Interpreter Options

It is possible to change some internal settings of the OSIRIS command token interpreter using the option `-set` ***stackSize errorNumber tableSize timeLineSize timeTolerance***.

14.5.1.1 stackSize

This is the size (in long) that is used for the local stack of the token interpreter. Since each instance of the interpreter has its own local stack, the stack should not be too large. The stack is used to store user-defined variables, parameters, local data of functions and so on.

A small stack is enough if you do not need much local memory or a large function call stack. If you need large amounts of local memory and use deep recursions, you should increase the stack size.

All instances of the token interpreter use the same stack size.

14.5.1.2 errorNumber

This value specifies the number of runtimes errors that have to appear to abort the current code execution. The machine will continue execution although there might be runtime errors, as long as this number is not reached. The value 0 (zero) means, that there is no limit for runtime errors. If a interpreter has reached the maximum number of runtime errors, the interpreter will quit the execution of the current UDP.

A small group of runtime errors causes immediate aborts, independent of the value set here. A runtime error of this type is a stack overflow.

14.5.1.3 tableSize

This value specifies the number of UDPs that can be hold simultaneously in the DPU's symbol table. If you specify a value that is larger than the currently set size, the DPU's symbol table will be enlarged. If the number is smaller, the table will be shrinked. Please note that the number has to be greater or equal to the maximum UDP ID currently uploaded.

14.5.1.4 timeLineSize

This specifies the number of time line entries that can be hold at once. You cannot shrink the time line to a size smaller than the current number of used entries.

14.5.1.5 timeTolerance

This specifies the maximum allowed belatedness of time line started UDPs. Time line entries that lay further in the past than ***timeTolerance*** milliseconds are treated as outdated and removed from time line without execution.

14.5.2 Reset UDP Manager

The option `-reset UDP manager` The OCL compiler may be used to generate a command sequence that causes the UDP manager on the DPU to do a full reset by. That means, all UDPs are stopped and deleted from DPU memory, and all settings are set to their defaults. UDPs that cannot be quitted in a couple of seconds can not be deleted. In this case, the reset will not be done completely.

14.5.3 Compiler options

This section contains some options that affect the compiler behavior only.

14.5.3.1 Create list of all preserved IDs

The option `–list` ***filename*** causes the compiler to write a list of all preserved IDs with corresponding names and parameter sizes into the file ***filename***.

14.5.3.2 Complete .ini-file

The option `–write.ini` causes the compiler to replace the current .ini-file by a completed version.

14.5.3.3 Configuration file

The option `–cfg` ***filename*** as first option of the OCL compiler can be used to specify a certain .ini-file. It may be followed by any other option described in this section.

14.5.3.4 Include files

The option `–include` ***filename*** **(** `+` ***filename*** **)*** as first option or directly after the configuration file option (see 14.5.3.3) specifies files that are to be included for the compilation. This option can be used in combination with `–xc` and `–tc`, where preprocessor directives won't work.

14.6 Outputs

14.6.1 Command Sequence File

All command sequences generated by the OCL compiler are saved to file. This file can be uploaded to DPU. The name of this file can be specified inside the **`OCL.ini`** file using the keyword **`CommandFile:`** (see 14.1.6).

14.6.1.1 Examples

```
CommandFile: code.bin
```

This line will cause each compiler run to append the generated data to the file named **`code.bin`**. This file should be deleted after successful upload.

```
CommandFile: *.bin
```

The compiler will generate a filename consisting of a string depending on the call parameters and the extension **`.bin`**. Compilation of the file **`StandardUDPs.ocl`** would result in the file **`StandardUDPs.bin`**.
Please note that the generated code will replace an existing file with the same name, if you use the wildcard option for the command file name.

14.6.2 Symbol Table Logfile

The symbol table file contains the interface of all UDPs that are present on DPU. Data types are not stored and have to be defined separately. The UDP interfaces are stored in the following format:

UDP	**:=**	***ID*** **`F`** ***name timestamp externflag*** **`[`** ***return*** **`]`** **`(`** ***parameters*** **`)`**	
ID	**:=**	8 digits hex value	
name	**:=**	up to 64 characters	
timestamp	**:=**	8 digits hex value	
externflag	**:=**	**(`-` | `&`)**	(`&`: is extern, `-`: is not extern)
return	**:=**	***symbol***	
parameters	**:=**	**(** ***symbol*** **)***	
symbol	**:=**	**`A`** ***info elements*** **`(`** ***symbol*** **`)`**	(array of symbols)
	|	**`S`** ***info size*** **`(`** **(** ***symbol*** **)*** **`)`**	(structure of symbols)
	|	**`U`** ***info size*** **`(`** **(** ***symbol*** **)*** **`)`**	(union of symbols)

|	**(B|N|I|R)** *info*	(simple symbol)
|	**V**	(void symbol)

The characters **A**, **S**, **U**, **B**, **N**, **I**, **R** and **V** specify the type of the symbol: **A**rray, **S**tructure, **U**nion, **B**oolean, **N**atural (unsigned integer) value, **I**nteger value, **R**eal (floating point) value and **V**oid. The flag **&** specifies reference parameters.

info	**:=**	***offset reference externflag***
offset	**:=**	8 digits hex value (offset of the data inside the memory)
reference	**:=**	8 digits hex value (offset of the pointer inside memory)
elements	**:=**	8 digits hex value (number of elements of arrays)
size	**:=**	8 digits hex value (size in memory units)

The symbol table log file is read before each UDP manipulation. Its contents should correspond to the information held in the DPU part of the UDP manager. If there is a difference between the ground symbol table and the DPU table, this can have two reasons:

1. Not all generated command sequences have been uploaded to DPU (due to not sending or upload error)

2. A malfunction of the UDP manager or memory hazard

To prevent malfunctioning of UDPs you should check the consistency of both tables once in a while (dump the DPU's table and compare it with this file).

The name of the symbol table file can be specified inside the **OCL.ini** file using the keyword **SymbolFile:** (see 14.1.7). The default file name is **OCLsym.txt**.

14.6.3 Command Sequence Information Logging

For archiving purposes, all successful generated UDPs are logged. The logging information consists of the UDP's source code followed by the ID and timestamp of the generated UDP. The source code inside the log file is preprocessed, all **#include**s are inserted and uses of symbols that are defined by **#define** are replaced already. Source code taken from this log file can be used as input for the OCL-compiler without modifications. Each logging is finished by a final message containing the name of the generated file and the generation time.

The name of the log file is set to **OCLlog.txt** by default and can be changed using the keyword **LogFile:** inside the configuration file **OCL.ini**.

14.6.3.1 Example

Let's assume, you run the OCL compiler three times with the following commands:

```
ocl –mc Init.ocl
ocl –xc "return true"
ocl –rc NEGATE correction.ocl
```

After these calls, the log file may look like this (without the explanations, of course):

```
long NEGATE (long lIn)
{
  return ~lIn;
}
```

It starts with the source code of the generated UDP,

```
#ifdef SHOWGENERATEDCOMMENTS
//==============================================================================
// UDP information
// ID  timestamp  name
// 008 0x3b04db18 NEGATE
//==============================================================================
#endif
```

followed directly by some information to the generated UDP: The ID of UDP `NEGATE` is `008` and it's timestamp is `0x3b04db18`.

The `#ifdef` / `#endif` is used for eliminating this automatically generated information, if code from the log file is recompiled.

```
################################################################################
 in:  Init.ocl
 out: Init.bin (new)
 logging closed on Fri May 18 08:19:36 2001
################################################################################
```

These lines mark the end of a compilation run. The original input file was named `Init.ocl`, the generated UDPs has been stored inside the command file `Init.bin`. The `(new)` means, that no previous data is stored inside the command file. If you do not use a wildcard * inside the name of the output file given inside ocl.ini, all commands will be appended to this file. In this case, you would get the message `(add)` instead. At last, the generation time is logged.

```
exec{
return true
;}
```

This is the source code of the compiler run for direct execution. The `exec{` `}` is generated automatically due to compiler needs. It is followed by the UDP information.

```
#ifdef SHOWGENERATEDCOMMENTS
//==============================================================================
// UDP information
// ID  timestamp  name
// 009 0x3b04dc9c 009:return_true
//==============================================================================
#endif
```

Of course, even this OCL-compiler run generates a final message:

```
################################################################################
 out: 009_return_trueExec.bin (new)
 logging closed on Fri May 18 13:28:13 2001
################################################################################
```

The `in:` line is missing, because there has been no input file, but only direct input. The third call of the OCL compiler replaces the former version of UDP `NEGATE`. Of course, the former version will remain inside the log file in addition to the new one:

```
long NEGATE (long lIn)
{
  return -lIn;
}
#ifdef SHOWGENERATEDCOMMENTS
//==============================================================================
// UDP information
// ID  timestamp  name
// 008 0x3b04dcad NEGATE
//==============================================================================
#endif
```

(As described above)

```
################################################################################
 former timestamp of UDP: 0x3b04db18
 out: NEGATEReplace.bin (new)
 logging closed on Fri May 18 13:31:57 2001
################################################################################
```

The first line of the final message mentions the timestamp of the former version that will be removed from DPU memory.

15 Index