Application Note

ÚTIA

Akademie věd České republiky
Ústav teorie informace a automatizace AV ČR, v.v.i.

# UTIA EdkDSP Platform
# WAL - Worker Abstraction Layer

Roman Bartosinski

*bartosr@utia.cas.cz*

## Revision

| Revision | Date | Author | Description |
|---|---|---|---|
| 0 | 11.3.2011 | Bartosinski | document created from DocBook version |
| | | | |
| | | | |

## Contents

## Acknowledgement

The research leading to these result has received funding from the ARTEMIS Joint Undertaking under grant agreement n° 100230 and from the MSMT 7H10001.

# 1 Introduction

This document describes *Worker Abstraction Layer* (WAL) which is an application programming interface for accessing hardware accelerators based on UTIA EdkDSP platform [1]. The interface is designed to simplify and generalize access from user applications running on the host CPU. Working with accelerators is partially the same for all accelerator and partially special for each type of accelerator. Therefore the interface is divided into common part and a specific parts.

The document describes functions and constants in the API, it also describes how to use the API in user applications. Version 1.0 of the API is described in the document.

# 2 How to use the programming interface in an application

## 2.1 Introduction

The WAL library offers generalised API for using in an application. The library is too close to hardware and therefore it is divide into two parts - common API for using in applications and low level functions specific for each worker. The common API hides differences between workers and it is very easy to use in an application. The following steps must be done:

- Add library to compilation process
- Include header files of the library
- Define worker structure or use the WAL_REGISTER_WORKER macro
- Initiate worker
- Use worker (set worker firmware, run operations, ...)

## 2.2 Include header files of the programming interface

The main `wal.h` header file must be included in the application. It contains the common part of the API. The next header is a specific part for the used worker and the name of the specific header depends on its implementation. In this example, the specific part of the interface for *Basic Computing Element* (BCE) workers designed in UTIA is in the files `wal_bce_jk.h`, `wal_bce_jk.c`.

Other header file included to the application is from a hardware accelerator low-level driver generated by the Xilinx tools. In the example, it is file 'bce_fp01_1x1_0_plbw.h'. Configuration structure of the hardware accelerator is defined in the driver and declared in the included header files. Example of an include part in the application for including WAL is in the following listing

```
#include <wal.h>
#include <wal_bce_jk.h>
#include <bce_fp01_1x1_0_plbw.h>
```

## 2.3 Define worker structure

Worker structure can be defined and prepared by the `WAL_REGISTER_WORKER` macro. The macro defines pointers to shared data/control memories, define worker structure and interconnect the worker structure with configuration table defined in the driver of the hardware IP core and with a part of the WAL library specific for the IP core. The macro is designed for static definition of the worker and it defines all items in the scope of code where it is used, i.e. the worker can be defined globally for entire application or locally in one function.

The `WAL_REGISTER_WORKER` macro has these arguments:

1. A name of the worker and a name of a pointer to the worker structure. The pointer is used as the first argument of all other functions from the WAL API.
2. A name of the worker family specific for the hardware IP core. (In this example it is defined in the `wal_bce_jk.h`.)
3. A name of the configuration structure of the hardware IP core defined in the generated low-level driver. (In this example it is declared in the `bce_fp01_1x1_0_plbw.h`.)
4. An index to the configuration structure of the hardware IP core. Normally it is equal to zero but if more instances are used it is different for each instance of the same type of IP core.
5. Number of SIMD units enabled in the hardware accelerator. Maximal number is defined in the worker family description structure in the specific part of the library (In this example it is defined in the `wal_bce_jk.h`.).
6. Number of supported memories enabled in the worker. Maximal number is defined in the worker family description structure in the specific part of the library (In this example it is defined in the `wal_bce_jk.h`.).

Example of registering

```
WAL_REGISTER_WORKER(worker, BCE_JK_FP32M24,
                    BCE_FP01_1X1_0_PLBW_ConfigTable, 0, 1, 0);
```

## 2.4 Add interface to compilation process

There are several ways how to add the WAL programming interface to compilation process. It depends if the interface is as source codes or static library and if the compilation is under Xilinx XPS or from command line.

If the interface is in source codes, they can be simply added next to a source code of the application and build together. In the XPS environment it is done by adding files of the interface as sources and headers to a software project, low-level driver is automatically accessible and therefore it isn't in the project.

```
Software Projects
 '- Project: Application
     |- Processor: microblaze_0
     |- Executable: application.elf
     |- Compiler Options
     |- Sources
     |    |- appl.c
     |    |- wal.c
     |    '- wal_bce_jk.c
     '- Headers
          |- wal.h
          '- wal_bce_jk.h
```

From command line, it can be done by compiling sources of the API to object files and then adding them to link together with application object files. For our example it can be done with the following commands (for standalone MicroBlaze system, all source codes are in the current directory).

```
# compile library
mb-gcc -c -o wal.o wal.c
mb-gcc -c -o wal_bce_jk.o wal_bce_jk.c
# compile low-level driver from EDK
mb-gcc -c -o bce_fp01_1x1_0_plbw.o bce_fp01_1x1_0_plbw.c
mb-gcc -c -o bce_fp01_1x1_0_plbw_g.o bce_fp01_1x1_0_plbw_g.c
# compile application
mb_gcc -c -o appl.o appl.c
# build application (link all objects together)
mb-gcc -o application.elf wal.o wal_bce_jk.o bce_fp01_1x1_0_plbw.o
                      bce_fp01_1x1_0_plbw_g.o appl.o
```

In case, when the API is distributed as a static library (as a file `libwal.a` and header files) the header files must be accessible during compilation and the library (libwal.a) must be added to linker. Under Xilinx XPS studio it is done by setting compilation options. Search paths for libraries and header files must be set on the 'Path and Options' pane. And name of the library 'wal' must be added into the 'Libraries to Link against' item.

From command line, the static library is added to compilation with the argument '-l' and the directory, where the library is, is added with the argument '-L'. The low-level driver genereted with XPS tools must be also compiled and link together.

```
# compile low-level driver from EDK
mb-gcc -c -o bce_fp01_1x1_0_plbw.o bce_fp01_1x1_0_plbw.c
mb-gcc -c -o bce_fp01_1x1_0_plbw_g.o bce_fp01_1x1_0_plbw_g.c
# compile application
mb_gcc -c -I../libwal -o appl.o appl.c
# build application (link all objects and the library together)
mb-gcc -o application.elf -lwal -L../libwal appl.o
                      bce_fp01_1x1_0_plbw.o bce_fp01_1x1_0_plbw_g.o
```

## 2.5 Initiate worker

The worker initialization is done by calling the function `wal_init_worker` with pointer to the worker structure, which must have filed at least pointer to worker family description structure; arrays of pointers to shared memories and their quantities; and pointer to an IPcore configuration structure defined in the IP core low-level driver generated by XPS. The function get the IP core interface structure and then obtain pointers to shared memories according to their names.

Example of worker initialization in the application:

```
if (wal_init_worker(worker)!=WAL_RES_OK) {
    printf("Init worker failed\n");
    return -1;
}
```

## 2.6 How to use worker

The worker can be used immediately after its initialization by calling functions from WAL API. Because the worker mostly doesn't contain any firmware at beginning and it needs a firmware for working, the first called function is for setting firmware. Then the application can obtain worker capabilities, writes data to worker data memories, runs operations on the worker and reads results from worker data memories.

A simple application for sum operation in the hardware accelerator is in the following listing

```
#include <stdio.h>

#include <wal.h>;
#include <wal_bce_jk.h>;
#include <bce_fp01_1x1_0_plbw.h>;
#include <worker_firmware.h>;
         /* data array with worker firmware prepared with pb-toolchain */

         /* define worker - BCE_JK_FP32M24 family, first instance of the
            BCE_FP01_1X1_0_PLBW HW core, with one SIMD, no support memories */
WAL_REGISTER_WORKER(worker, BCE_JK_FP32M24,
                    BCE_FP01_1X1_0_PLBW_ConfigTable, 0, 1, 0);


int main(void) {
   unsigned int caps;
   float input1[5] = {123.456, 23.45, 3.4, 0.123, 1.23};
   float output = 0;

         /* initiate worker */
   if (wal_init_worker(worker)!=WAL_RES_OK) {
      printf("Init WAL failed");
      return -1;
   }
         /* set firmware (array of uint with PB program) to the worker */
   if (wal_set_firmware(worker, WAL_PBID_P0, worker_firmware, -1)
      printf("Couldn't load fw to PB0\n");
      return -1;
   }
         /* get worker capabilities - they are defined in the 'wal_bce_jk.h' */
   if (wal_get_capabilities(worker, WAL_PBID_P0, &caps)) {
      printf("Couldn't read hw capabilities\n");
      return -1;
   }
   printf("Worker capabilities are %x\r\n", caps);

         /* write data to worker - copy 5 values from input1 to the accelerator
            first data memory B (SIMD=1 therefore we have only first set of
            memories) with offset 10 */
   wal_mb2dmem(worker, 0, WAL_BCE_JK_DMEM_B, 10, &input1[0], 5);
         /* run operation on the accelerator - it depends on the PB firmware */
   wal_start_operation(worker, WAL_PBID_P0);
         /* send 8bit parameters to PB firmware through control register -
            it depends on the PB firmware */
   wal_mb2pb(worker, SUM_OPERATION); /* do SUM of input data */
   wal_mb2pb(worker, 5); /* from 5 values */
   wal_mb2pb(worker, WAL_BCE_JK_DMEM_B); /* from data memory B */
   wal_mb2pb(worker, 10); /* with offset 10 */
   wal_mb2pb(worker, WAL_BCE_JK_DMEM_Z); /* result save into Z memory */
   wal_mb2pb(worker, 0); /* with offset 0 */
```

```
        /* wait for accelerator is done and stop operation -
           it depends on the PB firmware */
    result = wal_pb2mb(worker, NULL);
    wal_end_operation(worker);
        /* read result from accelerator data memory */
    wal_dmem2mb(worker, 0, WAL_BCE_JK_DMEM_Z, 0, &output, 1);
        /* print output */
    printf("Result of the SUM hw operation is %f\n", output);
}
```

## 2.7 Functions in the WAL API

The interface contains several functions for using in an application. These functions can be divided into groups according to their purpose (see the table below). All functions have a pointer to the worker as the first argument and therefore there can be more workers (and also more instancies of the same worker) in an application. All functions in the library have names which begin with the 'wal_' prefix. Detailed descriptions of all functions are in the next chapter.

Table 1: Functions in WAL API for using in applications

| **Function** | **Description** |
|---|---|
| *Init/Done functions* | |
| wal_init_worker | Initiate worker |
| wal_done_worker | Cleanup and release worker (not used) |
| *Basic control functions* | |
| wal_reset_worker | Send hard reset to the worker - set control part to the default state |
| wal_start_operation | Select and run preloaded firmware in the worker |
| wal_end_operation | Send request with reset to the worker (stop operating) |
| wal_mb2pb | Set control word of the worker (and also send data to accelerator's controller) |
| wal_pb2mb | Read status word of the worker (and also read data from accelerator's controller) |
| *Functions for working with control memories* | |
| wal_mb2cmem | Copy block of data to any shared control memory (worker firmwares, control registers, support memories) |
| wal_cmem2mb | Copy block of data from any shared control memory (status registers, support memories) |
| *Functions for working with data memories* | |
| wal_mb2dmem | Copy block of data to any shared data memory (specific for each worker) |
| wal_dmem2mb | Copy block of data from any shared data memory (specific for each worker) |
| *Common support functions* | |
| wal_set_firmware | Copy worker firmware to selected position |
| wal_get_id | Read worker ID (it can be read from the hardware or returned as a constant from software - it depends on implementation |
| wal_get_capabilities | Read worker capabilities (it depends on implementation of the worker) |
| wal_get_license | Read worker license (it depends on implementation of the worker) |

All constants in the WAL API have names which begins with the 'WAL_' prefix. In the following table there are prefixes for all common group of constants and for some of constants specific for implemented worker families. All constants are listed in detail in the next section.

Table 2: Constants declared in the WAL library

| **Group of constants** | **Description** |
|---|---|
| *Common groups of constants* | |
| `WAL_ID_` | IDs of worker classes |
| `WAL_BCE_ID_` | IDs of worker groups of families for the BCE class |
| `WAL_GCE_ID_` | IDs of worker groups of families for the GCE class |
| `WAL_OP_` | IDs of common operations which all new worker should use. There are three common operations: get_id, get_capabilities and get_license. |
| `WAL_RES_` | returned codes of all functions in the WAL API. See to the next table |
| `WAL_CMEM_` | IDs of all common control memories (probably they shouldn't be used directly in applications) |
| `WAL_PBID_` | IDs of firmware of the PicoBlaze which is run with in the function |
| `WAL_WRK_STATE_` | IDs of the current state of workers |
| *Constants specific for worker families* | |
| `WAL_BCE_JK_ID_` | IDs of worker families in the BCE_JK group of families |
| `WAL_BCE_JK_DMEM_` | IDs of data memories in the BCE_JK group of families |
| `WAL_BCE_JK_SMEM_` | IDs of support memories in the BCE_JK group of families |
| `WAL_BCE_JK_V` | codes of hardware accelerator operations in the BCE_JK group of families |
| `WAL_BCE_JK_CAP_V` | codes of accelerator capabilities of workers from the BCE_JK group of families |

All functions in the WAL library should return a return code. A positive value of the code means 'warning', i.e. the function hasn't been processed correctly but probably regarding to wrong input arguments. A negative value means 'error', and the returned zero means processing of the function was successful. All defined return codes are in the following table:

Table 3: Error codes returned by all functions in the WAL library

| **Name** | **Value** | **Description** |
|---|---|---|
| *Success* | | |
| `WAL_RES_OK` | 0 | Function finished with success |
| *Warnings* | | |
| `WAL_RES_WNULL` | 1 | An unimportant argument is NULL |
| *Errors* | | |
| `WAL_RES_ERR` | -1 | Generic unspecified error |
| `WAL_RES_ENOINIT` | -2 | Worker hasn't be initiated |
| `WAL_RES_ENULL` | -3 | A mandatory argument is equal to NULL |
| `WAL_RES_ERUNNING` | -4 | The function cannot be processed if the worker is working |
| `WAL_RES_ERANGE` | -5 | Value of any function argument is out of the allowed boundaries |

# 3 WAL Application Programming Interface

This section contains detailed description of all functions and constants in the API.

## 3.1 enum wal_worker_class_ids

### Purpose

`enum wal_worker_class_ids` - List of IDs of known worker classes

### Synopsis

```
enum wal_worker_class_ids {
                             WAL_ID_UNKNOWN,
                             WAL_ID_BCE,
                             WAL_ID_GCE,
                             WAL_ID_DEVEL
                         };
```

### Constants

| | |
|---|---|
| *WAL_ID_UNKNOWN* | unknown or unassigned class |
| *WAL_ID_BCE* | Basic Computing Elements class |
| *WAL_ID_GCE* | Graphic Computing Elements class |
| *WAL_ID_DEVEL* | Special ID for a new CE under development unfiled to WAL |

### Description

This IDs identify the worker class. The (8bit) class ID is the first byte (MSB) of a returned 32bit value from the **wal_get_id** function.

---

## 3.2 enum wal_bce_ids

### Purpose

`enum wal_bce_ids` - List of IDs of known groups of families under the BCE class

### Synopsis

```
enum wal_bce_ids {
                    WAL_BCE_ID_UNKNOWN,
                    WAL_BCE_ID_JK,
                    WAL_BCE_ID_HK,
                    WAL_BCE_ID_RB
                };
```

**Constants**

| | |
|---|---|
| *WAL_BCE_ID_UNKNOWN* | unknown or unassigned group of BCE worker families |
| *WAL_BCE_ID_JK* | worker group of families provided by J.Kadlec |
| *WAL_BCE_ID_HK* | worker group of families provided by H.Kloub |
| *WAL_BCE_ID_RB* | worker group of families provided by R.Bartosinski |

**Description**

This IDs identify worker group of families in the BCE class. In this version, a worker group of families implemented and provided by one author or maintainer. The (8bit) ID of group of families is the second byte (big endian) of a returned 32bit value from the **wal_get_id** function.

---

## 3.3 function wal_id_class

**Purpose**

`wal_id_class` - macro for extracting the class ID from full ID

**Synopsis**

**wal_id_class** ( *id* )

**Arguments**

| | |
|---|---|
| *id* | full worker ID |

**Description**

The macro extracts the 8bit ID of worker class from a full 32bit ID returned by the **wal_get_id** function.

---

## 3.4 function wal_id_group

**Purpose**

`wal_id_group` - macro for extracting ID of group of families from full ID

**Synopsis**

**wal_id_group** ( *id* )

#### Arguments

*id*
: full worker ID

#### Description

The macro extracts the 8bit ID of worker group of families from a full 32bit ID returned by the **wal_get_id** function. The family ID is dependent on the worker class.

---

## 3.5 function wal_id_family

#### Purpose

`wal_id_family` - macro for extracting worker family(user) ID from full ID

#### Synopsis

**wal_id_family** ( *id* )

#### Arguments

*id*
: full worker ID

#### Description

The macro extracts the 16bit worker family/user ID from a full 32bit ID returned by the **wal_get_id** function. The user ID is dependent on the worker class and group of families. It should contain number of SIMD for dynamically allocated resources.

---

## 3.6 enum wal_ctrl_memories_indicies

#### Purpose

`enum wal_ctrl_memories_indicies` - list of the control memories of the worker

#### Synopsis

```
enum wal_ctrl_memories_indicies {
        WAL_CMEM_MB2PB,
        WAL_CMEM_PB2MB,
        WAL_CMEM_P0,
        WAL_CMEM_P1,
        WAL_CMEM_NUM_MEMORIES
};
```

### Constants

| | |
|---|---|
| *WAL_CMEM_MB2PB* | index to MB2PB control memory (the control register of the worker) |
| *WAL_CMEM_PB2MB* | index to PB2MB control memory (the status register of the worker) |
| *WAL_CMEM_P0* | index to P0 control memory (PicoBlaze program memory 1) |
| *WAL_CMEM_P1* | index to P1 control memory (PicoBlaze program memory 2) |
| *WAL_CMEM_NUM_MEMORIES* | number of all defined control memories |

### Description

These indices correspond to an array of names of the control memories. Each family and its members can define their own control memories and their names in the family description structure.

---

## 3.7 enum wal_picoblaze_indices

### Purpose

`enum wal_picoblaze_indices` - list of indices of the PicoBlaze firmwares

### Synopsis

```
enum wal_picoblaze_indices {
                               WAL_PBID_P0,
                               WAL_PBID_P1,
                               WAL_PBID_NUM
                            };
```

### Constants

| | |
|---|---|
| *WAL_PBID_P0* | index of the PicoBlaze firmware 0 |
| *WAL_PBID_P1* | index of the PicoBlaze firmware 1 |
| *WAL_PBID_NUM* | number of all defined PB firmwares |

### Description

These indices are used as one parameter in some wal functions, where PB firmware must be select (e.g. **wal_set_firmware** ).

---

## 3.8 struct wal_worker

### Purpose

`wal_worker` - structure describes instance of a worker family

### Synopsis

```
struct wal_worker {
        int struct_ver;
        const char * name;
        const struct wal_family_desc * fm_desc;
        void * inst_cfg_tbl;
        xc_iface_t * inst_iface;
        xc_shram_t ** ctrl_mems;
        xc_shram_t ** data_mems;
        int num_simd;
        int num_smems;
        unsigned int op_state;
        int op_runmode;
        void * userdata;
};
```

**Members**

*struct_ver*
: version of the worker description structure - should be set when structure is created (registered) (the current version is WAL_DESC_STRUCT_VERSION_1)

*name*
: name of worker instance

*fm_desc*
: pointer to worker family description structure

*inst_cfg_tbl*
: pointer to worker IPcore instance config table - must be set before calling init function

*inst_iface*
: pointer to xc interface - it is obtained in the init function from config table

*ctrl_mems*
: pointer to an array of pointers to shared memories for control and support memories

*data_mems*
: pointer to on array of pointers to shared memories for data

*num_simd*
: number of used SIMD in the worker instance (the value cannot be greater than `fm_desc->nmax_simd`.

*num_smems*
: number of support memories used in the worker instance (the value cannot be greater than `fm_desc->nmax_supp_mems` ).

*op_state*
: the current state of operation in the instance

*op_runmode*
: HW runmode used in the current operation (automatically saved by the **start_op** function and cleared in **end_op** function)

*userdata*
: pointer to a user data or NULL

**Description**

The worker structure describes instance of a worker family. It interconnect worker family description structure (description of shared memories and control functions) with IPcore configuration table (description of hardware) and arrays of pointers to initiated shared memories.

---

## 3.9 function WAL_REGISTER_WORKER

**Purpose**

`WAL_REGISTER_WORKER` - macro for statical registering worker instance

**Synopsis**

**WAL_REGISTER_WORKER** ( *wname*, *wtype*, *wcfgtbl*, *winstidx*, *wsimd_num*, *wnum_smems*)

### Arguments

*wname*
: name of a new worker (and name of the worker structure

*wtype*
: type of worker - part of name used to identify worker family descriptions the following values and structure must be defined WAL_<wtype>_MAX_SIMD, WAL_<wtype>_DMEM_NUM_MEMORIES, WAL_<wtype>_CMEM_NUM_MEMORIES, WAL_<wtype>_SMEM_NUM_MEMORIES, wal_<wtype>_description_structure.

*wcfgtbl*
: name of IPcore config table (only name without index)

*winstidx*
: value of instance (used as index to IPcore config table)

*wsimd_num*
: number of SIMD used in the worker

*wnum_smems*
: number of support memories used in the worker

### Description

The macro declares all required shared memories and prepares structure of the worker instance and pointer to the structure with specified name `wname`.

---

## 3.10 enum wal_bce_jk_family_ids

### Purpose

`enum wal_bce_jk_family_ids` - family identifications in the BCE_JK group of families

### Synopsis

```
enum wal_bce_jk_family_ids {
        WAL_BCE_JK_ID_UNKNOWN,
        WAL_BCE_JK_ID_FP32M24
};
```

### Constants

*WAL_BCE_JK_ID_UNKNOWN*
: unknown family in the BCE_JK worker group of families

*WAL_BCE_JK_ID_FP32M24*
: the original family with 32bit floating point, 24bit mantisa

---

## 3.11 enum wal_bce_jk_data_memories

### Purpose

`enum wal_bce_jk_data_memories` - indices to BCE_JK data memories

### Synopsis

```
enum wal_bce_jk_data_memories {
                                    WAL_BCE_JK_DMEM_A,
                                    WAL_BCE_JK_DMEM_B,
                                    WAL_BCE_JK_DMEM_Z,
                                    WAL_BCE_JK_DMEM_NUM_MEMORIES
                                };
```

### Constants

| | |
|---|---|
| *WAL_BCE_JK_DMEM_A* | index of the A data memory |
| *WAL_BCE_JK_DMEM_B* | index of the B data memory |
| *WAL_BCE_JK_DMEM_Z* | index of the Z data memory |
| *WAL_BCE_JK_DMEM_NUM_MEMORIES* | number of data memories in the BCE_JK families |

## 3.12 enum wal_bce_jk_support_memories

### Purpose

`enum wal_bce_jk_support_memories` - indices to BCE_JK support memories

### Synopsis

```
enum wal_bce_jk_support_memories {
                                      WAL_BCE_JK_SMEM_MB2SM1,
                                      WAL_BCE_JK_SMEM_MB2SM2,
                                      WAL_BCE_JK_SMEM_MB2SM3,
                                      WAL_BCE_JK_SMEM_MB2SM4
                                  };
```

### Constants

*WAL_BCE_JK_SMEM_MB2SM1*
: index to the first support memory for the BCE_JK families family. It is the first index after indices to control memories (`WAL_BCE_JK_SMEM_MB2SM1` = `WAL_CMEM_NUM_MEMORIES`)

*WAL_BCE_JK_SMEM_MB2SM2*
: index to the second support memory for the BCE_JK families

*WAL_BCE_JK_SMEM_MB2SM3*
: index to the third support memory for the BCE_JK families

*WAL_BCE_JK_SMEM_MB2SM4*
: index to the fourth support memory for the BCE_JK families

### Description

All BCE_JK support memories are used for cosimulation only.

---

## 3.13 enum wal_bce_jk_operation_codes

### Purpose

`enum wal_bce_jk_operation_codes` - worker operations known to BCE_JK group of families

### Synopsis

```
enum wal_bce_jk_operation_codes {
        WAL_BCE_JK_VVER,
        WAL_BCE_JK_VZ2A,
        WAL_BCE_JK_VB2A,
        WAL_BCE_JK_VZ2B,
        WAL_BCE_JK_VA2B,
        WAL_BCE_JK_VADD,
        WAL_BCE_JK_VADD_BZ2A,
        WAL_BCE_JK_VADD_AZ2B,
        WAL_BCE_JK_VSUB,
        WAL_BCE_JK_VSUB_BZ2A,
        WAL_BCE_JK_VSUB_AZ2B,
        WAL_BCE_JK_VMULT,
        WAL_BCE_JK_VMULT_BZ2A,
        WAL_BCE_JK_VMULT_AZ2B,
        WAL_BCE_JK_VPROD,
        WAL_BCE_JK_VMAC,
        WAL_BCE_JK_VMSUBAC,
        WAL_BCE_JK_VPROD_S2,
        WAL_BCE_JK_VFP2SP,
        WAL_BCE_JK_VSP2FP,
        WAL_BCE_JK_VDIV
};
```

**Constants**

*WAL_BCE_JK_VVER*
: return version of HW (capabilities) in the Z data memory (simdID=0)

*WAL_BCE_JK_VZ2A*
: copy vector a[i] $\leq$ z[j]

*WAL_BCE_JK_VB2A*
: copy vector a[i] $\leq$ b[j]

*WAL_BCE_JK_VZ2B*
: copy vector b[i] $\leq$ z[j]

*WAL_BCE_JK_VA2B*
: copy vector b[i] $\leq$ a[j]

*WAL_BCE_JK_VADD*
: add vectors z[i] $\leq$ a[j] + b[k]

*WAL_BCE_JK_VADD_BZ2A*
: add vectors a[i] $\leq$ b[j] + z[k]

*WAL_BCE_JK_VADD_AZ2B*
: add vectors b[i] $\leq$ a[j] + z[k]

*WAL_BCE_JK_VSUB*
: sub vectors z[i] $\leq$ a[j] - b[k]

*WAL_BCE_JK_VSUB_BZ2A*
: sub vectors a[i] $\leq$ b[j] - z[k]

*WAL_BCE_JK_VSUB_AZ2B*
: sub vectors b[i] $\leq$ a[j] - z[k]

*WAL_BCE_JK_VMULT*
: mult vectors z[i] $\leq$ a[j] * b[k]

*WAL_BCE_JK_VMULT_BZ2A*
: mult vectors a[i] $\leq$ b[j] * z[k]

*WAL_BCE_JK_VMULT_AZ2B*
: mult vectors b[i] $\leq$ a[j] * z[k]

*WAL_BCE_JK_VPROD*
: vector product z $\leq$ a'[i..i+nn]*b[i..i+nn]

*WAL_BCE_JK_VMAC*
: vector MAC z[i] $\leq$ z[i] - a[j]*b[k] 1..13.

*WAL_BCE_JK_VMSUBAC*
: vector MSUBAC z[i] $\leq$ z[i] - a[j]*b[k] 1..13.

*WAL_BCE_JK_VPROD_S2*
: vector product extended z $\leq$ (a1'[i..i+nn]*b1[i..i+nn] + a2'[i..i+nn]*b2[i..i+nn]) the same code has the WAL_BCE_JK_VPROD_S4 operation z $\leq$ (a1'[i..i+nn]*b1[i..i+nn] + a2'[i..i+nn]*b2[i..i+nn])+ (a3'[i..i+nn]*b3[i..i+nn] + a4'[i..i+nn]*b4[i..i+nn]) and the WAL_BCE_JK_VPROD_S8 operation z $\leq$ ((a1'[i..i+nn]*b1[i..i+nn]+a2'[i..i+nn]*b2[i..i+nn])+ (a3'[i..i+nn]*b3[i..i+nn]+a4'[i..i+nn]*b4[i..i+nn])) + ((a5'[i..i+nn]*b5[i..i+nn]+a6'[i..i+nn]*b6[i..i+nn])+ (a7'[i..i+nn]*b7[i..i+nn]+a8'[i..i+nn]*b8[i..i+nn]))

*WAL_BCE_JK_VFP2SP*
: vector conversion from proprietary FP to 32m24 single precision FP

*WAL_BCE_JK_VSP2FP*
: vector conversion from 32m24 single precision FP to proprietary FP

*WAL_BCE_JK_VDIV*
: vector division

## 3.14 enum wal_bce_jk_capabilities

### Purpose

`enum wal_bce_jk_capabilities` - BCE_JK possible worker capabilities

### Synopsis

```
enum wal_bce_jk_capabilities {
                                WAL_BCE_JK_CAP_VVER,
                                WAL_BCE_JK_CAP_VZ2A,
                                WAL_BCE_JK_CAP_VB2A,
                                WAL_BCE_JK_CAP_VZ2B,
                                WAL_BCE_JK_CAP_VA2B,
                                WAL_BCE_JK_CAP_VADD,
                                WAL_BCE_JK_CAP_VADD_BZ2A,
                                WAL_BCE_JK_CAP_VADD_AZ2B,
                                WAL_BCE_JK_CAP_VSUB,
                                WAL_BCE_JK_CAP_VSUB_BZ2A,
                                WAL_BCE_JK_CAP_VSUB_AZ2B,
                                WAL_BCE_JK_CAP_VMULT,
                                WAL_BCE_JK_CAP_VMULT_BZ2A,
                                WAL_BCE_JK_CAP_VMULT_AZ2B,
                                WAL_BCE_JK_CAP_VPROD,
                                WAL_BCE_JK_CAP_VMAC,
                                WAL_BCE_JK_CAP_VMSUBAC,
                                WAL_BCE_JK_CAP_VPROD_S2,
                                WAL_BCE_JK_CAP_VFP2SP,
                                WAL_BCE_JK_CAP_VSP2FP,
                                WAL_BCE_JK_CAP_VDIV
                              };
```

**Constants**

*WAL_BCE_JK_CAP_VVER*
: worker supports getting version of HW (capabilities)

*WAL_BCE_JK_CAP_VZ2A*
: worker supports function copy vector a[i] $\leq$ z[j]

*WAL_BCE_JK_CAP_VB2A*
: worker supports function copy vector a[i] $\leq$ b[j]

*WAL_BCE_JK_CAP_VZ2B*
: worker supports function copy vector b[i] $\leq$ z[j]

*WAL_BCE_JK_CAP_VA2B*
: worker supports function copy vector b[i] $\leq$ a[j]

*WAL_BCE_JK_CAP_VADD*
: worker supports function add vectors z[i] $\leq$ a[j] + b[k]

*WAL_BCE_JK_CAP_VADD_BZ2A*
: worker supports function add vectors a[i] $\leq$ b[j] + z[k]

*WAL_BCE_JK_CAP_VADD_AZ2B*
: worker supports function add vectors b[i] $\leq$ a[j] + z[k]

*WAL_BCE_JK_CAP_VSUB*
: worker supports function sub vectors z[i] $\leq$ a[j] - b[k]

*WAL_BCE_JK_CAP_VSUB_BZ2A*
: worker supports function sub vectors a[i] $\leq$ b[j] - z[k]

*WAL_BCE_JK_CAP_VSUB_AZ2B*
: worker supports function sub vectors b[i] $\leq$ a[j] - z[k]

*WAL_BCE_JK_CAP_VMULT*
: worker supports function mult vectors z[i] $\leq$ a[j] * b[k]

*WAL_BCE_JK_CAP_VMULT_BZ2A*
: worker supports function mult vectors a[i] $\leq$ b[j] * z[k]

*WAL_BCE_JK_CAP_VMULT_AZ2B*
: worker supports function mult vectors b[i] $\leq$ a[j] * z[k]

*WAL_BCE_JK_CAP_VPROD*
: worker supports function vector product z $\leq$ a'[i..i+nn]*b[i..i+nn]

*WAL_BCE_JK_CAP_VMAC*
: worker supports function vector MAC z[i] $\leq$ z[i] - a[j]*b[k] 1..13.

*WAL_BCE_JK_CAP_VMSUBAC*
: worker supports function vector MSUBAC z[i] $\leq$ z[i] - a[j]*b[k] 1..13.

*WAL_BCE_JK_CAP_VPROD_S2*
: worker supports function vector product extended z $\leq$ (a1'[i..i+nn]*b1[i..i+nn] + a2'[i..i+nn]*b2[i..i+nn]) the same flag is for SIMD 4 (WAL_BCE_JK_CAP_VPROD_S4) and the same flag is for SIMD 8 (WAL_BCE_JK_CAP_VPROD_S8)

*WAL_BCE_JK_CAP_VFP2SP*
: worker supports function vector conversion from proprietary FP to 32m24 single precision FP

*WAL_BCE_JK_CAP_VSP2FP*
: worker supports function vector conversion from 32m24 single precision FP to proprietary FP

*WAL_BCE_JK_CAP_VDIV*
: worker supports function vector division

## 3.15 function wal_init_worker

### Purpose

`wal_init_worker` - generalised function for worker initialising

### Synopsis

`int` **wal_init_worker** (`struct wal_worker *` *wrk*)

### Arguments

*wrk*
: pointer to the worker structure

### Description

This function is designed for calling from user application. The function checks if the `wrk` structure is prepared to initiate worker (the family description structure must be set). Then the assigned family function (**init_wrk**) is called. In the called function all arrays of pointers to shared memories should be initiated.

### Return Value

The function returns return code `WAL_RES_OK` if successful and `WAL_RES_E`... if any error occurs.

---

## 3.16 function wal_done_worker

### Purpose

`wal_done_worker` - generalised function for worker cleanup

### Synopsis

`int` **wal_done_worker** (`struct wal_worker *` *wrk*)

### Arguments

*wrk*
: pointer to the worker structure

### Description

This function is designed for calling from user application. The function calls done function (**done_wrk**) assigned to family description structure. In the called function all dynamically allocated worker structures, memories and resources should be cleanup and released if they have been created in the worker init function.

### Return Value

The function returns `WAL_RES_...` codes.

---

## 3.17 function wal_reset_worker

### Purpose

`wal_reset_worker` - generalised function for worker hard reset

### Synopsis

`int` **wal_reset_worker** (`struct wal_worker *` *wrk*)

### Arguments

*wrk*
: pointer to the worker structure

### Description

This function is designed for calling from user application. The function calls reset function (**reset_wrk**) assigned to the family description structure. In the called function the worker control registers should be reset (by HARD RESET bit in the worker control register). The reset is not acknowledged by accelerator.

### Return Value

The function returns `WAL_RES_...` codes.

---

## 3.18 function wal_start_operation

### Purpose

`wal_start_operation` - generalised function for starting operation on the accelerator

### Synopsis

`int` **wal_start_operation** (`struct wal_worker *` *wrk*, `unsigned int` *pbid*)

### Arguments

*wrk*
: pointer to the worker structure

*pbid*
: index of used PB firmware (`WAL_PBID_...`)

**Description**

This function is designed for calling from user application. The function checks if the accelerator is in the idle state and then it calls function for starting operation (**start_op**) assigned to the family description structure. The called function should start a new accelerator operation by setting accelerator control register and checking status register. This function is blocking, i.e. it waits for acknowledgement from accelerator.

**Return Value**

The function returns `WAL_RES_`... codes.

---

## 3.19 function wal_end_operation

**Purpose**

`wal_end_operation` - generalised function for finishing operation on the accelerator

**Synopsis**

`int` **wal_end_operation** (`struct wal_worker *` *wrk*)

**Arguments**

*wrk*
pointer to the worker structure

**Description**

This function is designed for calling from user application. The function checks if the accelerator is in processing state and then it calls function for ending operation (**end_op**) assigned to the family description structure. The called function should stop processing operation on the accelerator. And it waits for synchronization with the accelerator, therefore the function is blocking.

**Return Value**

The function returns `WAL_RES_`... codes.

---

## 3.20 function wal_mb2pb

**Purpose**

`wal_mb2pb` - generalised function for setting worker control register

**Synopsis**

`int` **wal_mb2pb** (`struct wal_worker *` *wrk*, `const uint32_t` *data*)

### Arguments

*wrk*
: pointer to the worker structure

*data*
: user data sends to worker control register

### Description

This function is designed for calling from user application. The function calls function for setting worker control register (**mb2pb**) assigned to the family description structure. The called function should send user data through control register with controlling READ bit. It should also waits for synchronization with accelerator.

### Return Value

The function returns `WAL_RES_..` codes.

---

## 3.21 function wal_pb2mb

### Purpose

`wal_pb2mb` - generalised function for reading worker status register

### Synopsis

`int` **wal_pb2mb** (`struct wal_worker *` *wrk*, `uint32_t *` *data*)

### Arguments

*wrk*
: pointer to the worker structure

*data*
: pointer to an output buffer where read user data is written

### Description

This function is designed for calling from user application. The function calls function for reading worker status register (**pb2mb**) assigned to the family description structure. The called function should read user data through worker status register with waiting for synchronization with accelerator.

### Return Value

The function returns `WAL_RES_..` codes.

---

## 3.22 function wal_mb2cmem

### Purpose

`wal_mb2cmem` - generalised function for writing block of data to any worker control or support memory

### Synopsis

`int` **wal_mb2cmem** `(struct wal_worker *` *wrk*, `unsigned int` *memid*, `unsigned int` *memoffs*, `const uint32_t *` *outbuf*, `unsigned int` *len*)

### Arguments

| | |
|---|---|
| *wrk* | pointer to the worker structure |
| *memid* | index of control/support memory where data are written to (`WAL_CMEM_`... or `WAL_`..._SMEM_...) |
| *memoffs* | offset in selected memory (in words not in bytes) |
| *outbuf* | pointer to memory where data are read from |
| *len* | number of words to copy from `outbuf` to accelerator control memory |

### Description

This function is designed for calling from user application. The function checks index of the required memory and then it calls function for writing data to any control/support memory (**mb2cmem**) assigned to the family description structure. The called function should get pointer to the right memory according to the required index `memid`. For accessing support memories they have to defined indices greater then indices to control memories. Then the called function should copy block of data from CPU memory `outbuf` to an accelerator control/support memory selected by `memid` and offset in selected memory `memoffs`.

### Return Value

The function returns `WAL_RES_`... codes.

---

## 3.23 function wal_cmem2mb

### Purpose

`wal_cmem2mb` - generalised function for reading block of data from any worker control or support memory

### Synopsis

`int` **wal_cmem2mb** `(struct wal_worker *` *wrk*, `unsigned int` *memid*, `unsigned int` *memoffs*, `uint32_t *` *inbuf*, `unsigned int` *len*)

### Arguments

*wrk*
: pointer to the worker structure

*memid*
: index of control/support memory where data are read from (`WAL_CMEM_...` or `WAL_..._SMEM_...`)

*memoffs*
: offset in selected memory (in words not in bytes)

*inbuf*
: pointer to memory where data are written to

*len*
: number of words to copy from `outbuf` to accelerator control memory

### Description

This function is designed for calling from user application. The function checks index of the required memory and then it calls function for reading data from any control/support memory (**cmem2mb**) assigned to the family description structure. The called function should get pointer to the right memory according to the required index `memid`. For accessing support memories they have to defined indices greater then indices to control memories. Then the called function should copy block of data from the accelerator control/support memory selected by `memid` and offset in selected memory `memoffs`.

### Return Value

The function returns `WAL_RES_...` codes.

---

## 3.24 function wal_mb2dmem

### Purpose

`wal_mb2dmem` - generalised function for writing block of data to any worker data memory

### Synopsis

`int` **wal_mb2dmem** (`struct wal_worker *` *wrk*, `unsigned int` *simdid*, `unsigned int` *memid*, `unsigned int` *memoffs*, `const void *` *outbuf*, `unsigned int` *len*)

### Arguments

|  |  |
|---|---|
| *wrk* | pointer to the worker structure |
| *simdid* | index of SIMD which data memories are indexed |
| *memid* | index of control/support memory where data are written to (`WAL_CMEM_`... or `WAL_`..._SMEM_...) |
| *memoffs* | offset in selected memory (in words not in bytes) |
| *outbuf* | pointer to memory where data are read from |
| *len* | number of words to copy from `outbuf` to accelerator control memory |

### Description

This function is designed for calling from user application. The function checks index of the required memory and then it calls function for writing data to any data memory (**mb2dmem**) assigned to the family description structure. The called function should get pointer to the right memory according to the required SIMD `simdid` and memory index `memid`. Then the called function should copy block of data from CPU memory `outbuf` to the accelerator data memory with offset inside the selected memory `memoffs`.

### Return Value

The function returns `WAL_RES_`... codes.

---

## 3.25 function wal_dmem2mb

### Purpose

`wal_dmem2mb` - generalised function for writing block of data to any worker data memory

### Synopsis

`int` **wal_dmem2mb** (`struct wal_worker *` *wrk*, `unsigned int` *simdid*, `unsigned int` *memid*, `unsigned int` *memoffs*, `void *` *inbuf*, `unsigned int` *len*)

### Arguments

*wrk*
: pointer to the worker structure

*simdid*
: index of SIMD which data memories are indexed

*memid*
: index of control/support memory where data are read from (`WAL_CMEM_`... or `WAL_`..._SMEM_...)

*memoffs*
: offset in selected memory (in words not in bytes)

*inbuf*
: pointer to memory where data are written to

*len*
: number of words to copy from `outbuf` to accelerator control memory

### Description

This function is designed for calling from user application. The function checks index of the required memory and then it calls function for reading data from any data memory (**dmem2mb**) assigned to the family description structure. The called function should get pointer to the right memory according to the required SIMD `simdid` and memory index `memid`. Then the called function should copy block of data from the accelerator data memory with offset inside the selected memory `memoffs`.

### Return Value

The function returns `WAL_RES_`... codes.

---

## 3.26 function wal_set_firmware

### Purpose

`wal_set_firmware` - generalised function for writing PicoBlaze firmware

### Synopsis

`int` **wal_set_firmware** (`struct wal_worker *` *wrk*, `int` *pbid*, `const unsigned int *` *fwbuf*, `int` *fw-size*)

### Arguments

*wrk*
: pointer to the worker structure

*pbid*
: index of used PB firmware (`WAL_PBID_`...)

*fwbuf*
: pointer to a firmware in CPU memory

*fwsize*
: size of the firmware in words, it can be a negative value to set full firmware (1024 words)

### Description

This function is designed for calling from user application. The function checks if all arguments are correct and then it calls function for writing PB firmware (**set_fw**). The called function should copy firmware from CPU memory `fwbuf` to Picoblaze program memory in the accelerator. The PB program memory is selected by the argument `pbid`. The firmware needn't be full 1024 word long. The firmware length (in words) can be set by the argument `fwsize`. If the `fwsize` is a negative value (you can use defined value `WAL_FW_WHOLE`) the function assumes the FW length is 1024 words.

### Return Value

The function returns `WAL_RES_`... codes.

---

## 3.27 function wal_get_id

### Purpose

`wal_get_id` - generalised function for getting ID from the worker

### Synopsis

`int` **wal_get_id** (`struct wal_worker *` *wrk*, `int` *pbid*, `unsigned int *` *outval*)

### Arguments

| | |
|---|---|
| *wrk* | pointer to the worker structure |
| *pbid* | index of used PB firmware (`WAL_PBID_`...) |
| *outval* | pointer to an output buffer for read worker ID |

### Description

This function is designed for calling from user application. The function checks arguments and calls function for getting ID from hardware (**get_id**) assigned to the family description structure. The called function should start, process and end entire worker operation for getting ID. The 32bit ID of the worker `wrk` is returned in the output buffer `outbuf`. The ID is merged in this order (big endian - MSB first): the worker class ID (8bit), worker group of family ID (8bit) and worker family/user ID (16bit).

### Return Value

The function returns `WAL_RES_`... codes

---

## 3.28 function wal_get_capabilities

### Purpose

`wal_get_capabilities` - generalised function for getting capabilities of the worker

#### Synopsis

`int` **wal_get_capabilities** (`struct wal_worker *` *wrk*, `int` *pbid*, `unsigned int *` *outval*)

#### Arguments

*wrk*
: pointer to the worker structure

*pbid*
: index of selected PB firmware (`WAL_PBID_`...)

*outval*
: pointer to an output buffer for read worker capabilities

#### Description

This function is designed for calling from user application. The function checks arguments and calls function for getting capabilities from the hardware (**get_cap**) assigned to the family description structure. The called function should start, process and end entire worker operation for getting capabilities. The 32bit capabilities of the worker `wrk` is a family specific bitmap and it is returned in the output buffer `outbuf`.

#### Return Value

The function returns `WAL_RES_`... codes

---

## 3.29 function wal_get_license

#### Purpose

`wal_get_license` - generalised function for getting license of the worker

#### Synopsis

`int` **wal_get_license** (`struct wal_worker *` *wrk*, `int` *pbid*, `unsigned int *` *outval*)

#### Arguments

*wrk*
: pointer to the worker structure

*pbid*
: index of used PB firmware (`WAL_PBID_`...)

*outval*
: pointer to an output buffer for read license

#### Description

This function is designed for calling from user application. The function checks arguments and calls function for getting license code from the hardware (**get_lic**) assigned to the family description structure. The called function should start, process and end entire worker operation for getting license cap.

The license is dependent on implementation of the worker family, it can be a counter or information in bitmap, but the returned value = 0 should mean that the license is expired. The license is returned in the `outbuf` buffer. If the family function (**get_lic**) is set to NULL the function returns license=1.

### Return Value

The function returns `WAL_RES_`... codes.

---

## References

[1] J. Kadlec and all, “D2.1 - preliminary report on platform dependent parameters and optimizations,” August 2010. SMECY project deliverable D2.1.