

# Application Note



## UTIA EdkDSP Platform Firmware Programming Interface v2 PBBCELIB

Roman Bartosinski

*bartosr@utia.cas.cz*

### Revision

Revision	Date	Author	Description
0	01.10.2011	Bartosinski	document creation

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Content of the programming interface</b>	<b>1</b>
<b>3</b>	<b>How to use the programming interface</b>	<b>2</b>
3.1	Include API header file . . . . .	2
3.2	Use functions from API . . . . .	2
3.3	Build firmware binary . . . . .	3
3.4	Example of firmware with the API . . . . .	3
<b>4</b>	<b>User's point of view to computation in DFU</b>	<b>4</b>
<b>5</b>	<b>Changes from the previous version</b>	<b>7</b>
<b>6</b>	<b>Application Programming Interface</b>	<b>8</b>
6.1	enum dfu_bce_operations . . . . .	8
6.2	enum dfu_ag_indices . . . . .	8
6.3	enum dfu_idxag_indices . . . . .	9
6.4	enum dfu_data_memories . . . . .	10
6.5	enum dfu_ag_flags . . . . .	11
6.6	function mbpb_exchange_data . . . . .	11
6.7	function pb2mb_report_running . . . . .	12
6.8	function write_bce_id_to_cmem . . . . .	12
6.9	function write_dfu_caps_to_cmem . . . . .	13
6.10	function read_bce_cmem_u8 . . . . .	13
6.11	function read_bce_cmem_u16 . . . . .	14
6.12	function write_bce_cmem_u16 . . . . .	14
6.13	function pcnt_get_dfutime . . . . .	14
6.14	function pcnt_get_prptime_lo . . . . .	15
6.15	function pcnt_get_prptime_hi . . . . .	15
6.16	function pcnt_reset_prptime . . . . .	16
6.17	function get_dfulic . . . . .	16
6.18	function pb2dfu_wait4hw . . . . .	16
6.19	function pb2dfu_start_op . . . . .	17
6.20	function pb2dfu_restart_op . . . . .	17
6.21	function pb2dfu_set_cnt . . . . .	18
6.22	function pb2dfu_set_addr . . . . .	18
6.23	function pb2dfu_set_bank . . . . .	19
6.24	function pb2dfu_set_fulladdr . . . . .	19
6.25	function pb2dfu_set_inc . . . . .	20
6.26	function pb2dfu_set_bound_addr . . . . .	20
6.27	function pb2dfu_set_agflags . . . . .	21
6.28	function pb2dfu_set_repetitions . . . . .	21

## 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

The application programming interface (API), described in this document, is one part of a software development kit for hardware accelerators based on EdkDSP platform (EdkDSP SDK). The API version 2.0 (APIv2) for accelerator's microcontroller is described in this document. In this case, a PicoBlaze (PB) is used as a microcontroller and a 32bit processor MicroBlaze (MB) is used as the host CPU.

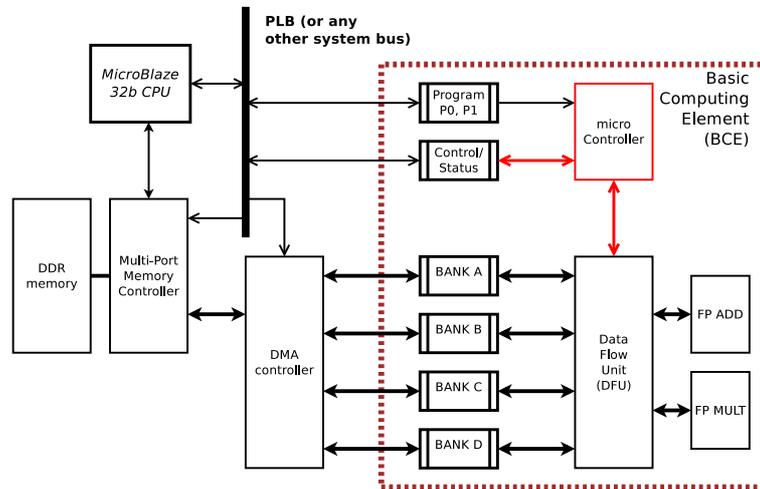


Figure 1: Basic Computing Element (BCE) - User's View. Data paths from outside memory to BCE and in BCE are represented by bold lines.

The API defines an interfaces between the microcontroller and data flow unit (PB2DFU) and between the accelerator and the host CPU from the side of the accelerator (PB2MB). It offers functions for communication with the host CPU and functions to parameterize and control basic computing operations in hardware.

The general concept of the EdkDSP platform is described in the deliverable D2.1 in SMECY project [1].

The document is organized as follows: The first part contains general information about the PB API and the second part contains description of all functions and constants in the APIv2 which is generated from the source code.

## 2 Content of the programming interface

The firmware programming interface is distributed as one header file for C compiler `pbbcelib.h` and library as one object file `pbbcelib.psmo`.

The programming interface is prepared for using with tools from EdkDSP SDK which are distributed in package `pb-toolchain`. The toolchain compiles source codes in limited C to binary firmware for the PicoBlaze microcontroller.

The PB2 APIv2 is distributed in the following directory structure

```
.
|-- api
    |-- 10-pb-firmware
        |-- doc
        |   |-- pbbcelib.pdf
        |-- pbbcelib.h
```

```
'-- pbbcelib.psmo
```

Because both versions of accelerators and also their PB APIs can be used at the same time in the same examples, this version of the API is distributed in subdirectory which begins with a digit one 10-pb-firmware (The previous version has been distributed in subdirectory 00-pb-firmware).

The following directory structure is considered in all examples in this document. All source codes for accelerator's firmware are in subdirectories 00-pb-firmware.

```
.
|-- api
|   |-- 10-pb-firmware
|   |-- 11-mb-standalone
|   '-- 12-mb-petalinux
|-- example1
|   |-- 10-pb-firmware
|   |-- 11-mb-standalone
|   '-- 12-mb-petalinux
```

### 3 How to use the programming interface

Usage of the API can be described in the following steps

- include API header file to the C source code,
- use functions from API in the source code,
- build firmware binary from source codes and the API library.

#### 3.1 Include API header file

Functions in the programming interface are declared in the `pbbcelib.h` header file. It must be included in the source code with the preprocessor directive `#include`.

```
#include "../..//api/10-pb-firmware/pbbcelib.h"
```

#### 3.2 Use functions from API

All functions and constants from the API can be used after including API header file. They can be divided into three groups.

The first group performs communication between accelerator and host CPU (which should use WAL API for communication with BCE accelerators). Against the previous version, only one function `mbpb_exchange_data` (section 6.6) is designed for this purpose. This function can receive and send 8bit value in one transaction. The second function in this group is `pb2mb_report_running` (section 6.7) which must be used at the beginning of the firmware to inform host CPU that accelerator is programmed and alive. Next functions (`read_bce_cmем_u8`, `read_bce_cmем_u16`, `write_bce_cmем_u16` - sections 6.10, 6.11, 6.12) read and write data to control memory which is accessible to host CPU.

The second group contains functions for collecting information about BCE and DFU. It contains the following functions: `write_bce_id_to_cmем` (6.8), `write_dfu_caps_to_cmем` (6.9), `pcnt_get_dfutime` (6.13), `pcnt_get_prptime_lo` (6.14), `pcnt_get_prptime_hi` (6.15), `get_dfulic` (6.17).

The third group contains functions for controlling DFU, i.e. settings of a following operation (`pb2dfu_set_`), operation execution (`pb2dfu_start_op`, `pb2dfu_restart_op`) and waiting to its finishing (`pb2dfu_wait4hw`). These functions begin with prefix `'pb2dfu_'`. Parameters of the next operation can be set before the

previous operation is done, but it cannot be launched before the previous operation is done. This feature allows to launch operation with minimal delay caused by microcontroller. The data flow unit remember the last settings and therefore only changed parameters can be set for the next operation.

### 3.3 Build firmware binary

Firmwares in C source codes must be compiled with the EdkDSP SDK toolchain distributed in the package `pb-toolchain`. The simplest way how to compile and build firmware binary is in the following listing. In the listing, we consider that the toolchain is installed and we compile the `example1` in directory `./example1/10-pb-firmware` with directory structure as shown in section 2.

```
pbcc ../../api/10-pb-firmware/pbbcelib.psmo fw_ex1.c -o fw_ex1.h
```

In this example, the output binary has form of C header file with array of binary codes of firmware. Such file will be used to compile application for the host CPU as described in the documentation of the WAL API [2].

### 3.4 Example of firmware with the API

This part shows simple example of firmware, which waits for operation required by the host CPU and if the operation is `wal_get_capability` the operation is starts in the accelerator. When the operation is finished the controller send return code to the host CPU.

```
/* include PBBCE library for BCE accelerator */
#include "../../api/10-pb-firmware/pbbcelib.h"

#define MY_MULT_OP 0x10

void main()
{
    unsigned char op;
    unsigned char rc;

    pb2mb_report_running();

    do {
        /* waiting for data from MB (the first byte is a required operation) */
        op = mbpb_exchange_data(0);
        rc = 0xff;

        if (op==WAL_OP_GETCAP) { /* FW support only this one operation */
            /* read DFU capability and write it into the control/status memory (8 words) */
            write_dfu_caps_to_cmem();
            rc = 0;
        } else if (op==MY_MULT_OP) {
            int l = mbpb_exchange_data(0);
            pb2dfu_set_bank(DFUAG_0, MBANK_A);
            pb2dfu_set_addr(DFUAG_0, 0x100);
            pb2dfu_set_fulladdr(DFUAG_1, MBANK_B, 0x08);
            pb2dfu_set_inc(DFUAG_0, 1);
            pb2dfu_set_inc(DFUAG_1, 2);
        }
    } while (rc != 0);
}
```

```

    pb2dfu_start_op(DFU_VMUL, 1);
    rc = pb2dfu_wait_hw();
}

/* send result/return code to MB */
mbpb_exchange_data(rc);
} while(1);
return 0;
}

```

## 4 User's point of view to computation in DFU

All functions from the third group affect settings and processing of the next operation computed in DFU. The DFU basically performs the following code

---

### Algorithm 1 User's view to process operation in DFU

---

```

loop
  if start operation op then
    prepare processing
      { set memory crossbar and address generators from DFU control registers
        - assign physical data memories (A,B,C,D) to virtual vectors  $M_{0..7}$ 
        - set initial indices of elements in vectors  $M$   $i_k(1) = O_k, k \in 0..7$  }
    for  $r=1$  to  $REP$  do
      for  $n=1$  to  $N$  do
        perform operation op for element  $n$ 
          ( $M_0(i_0) = op(M_k(i_k); k \in 1..3)$ )
        update address generators
      end for
    end for
    update DFU status register
  end if
end loop

```

---

Algorithm of updating address of the next vector element in physical data memory is the same for all address generators. This version of the accelerator supports several modes of updating address. The mode is selected with function `pb2dfu_set_agflags` (6.27). Each address generator contains two counters: the main counter (marked `DFUAG_0`, `DFUAG_1`, `DFUAG_2`, `DFUAG_3` in API functions) and slave counter (marked `DFUAG_4`, `DFUAG_5`, `DFUAG_6`, `DFUAG_7` in API functions).

The following pseudo-algorithms (algorithm 2-5) show updating address in an address generator. The main generator is marked with subscript 0 and the slave generator is marked with subscript 4. Table 1 shows how functions from the API affect settings. In algorithms and the table, meaning of parameters are:

- the previous value of the main generator  $i_0(k-1)$  and the slave generator  $i_4(k-1)$
- data from virtual vector  $M_4[i_4(k)]$
- updating mode  $MODE$
- increments of both generators  $I_0, I_4$
- lower and upper boundaries of both generators  $L_0, U_0, L_4, U_4$

---

**Algorithm 2** Update address of the next element  $i_0(k)$  in a data memory.

$MODE = 0$

---

$$i_0(k) = i_0(k-1) + I_0$$

**if**  $I_0 < 0 \wedge i_0(k) < L_0$  **then**

$$i_0(k) = U_0$$

**else if**  $I_0 > 0 \wedge i_0(k) > U_0$  **then**

$$i_0(k) = L_0$$

**end if**

---

---

**Algorithm 3** Update address of the next element  $i_0(k)$  in a data memory.

$MODE = USE\_IDX$

---

$$i_4(k) = i_4(k-1) + I_4$$

**if**  $I_4 < 0 \wedge i_4(k) < L_4$  **then**

$$i_4(k) = U_4$$

**else if**  $I_4 > 0 \wedge i_4(k) > U_4$  **then**

$$i_4(k) = L_4$$

**end if**

$$i_0(k) = i_0(k-1) + I_0$$

**if**  $I_0 < 0 \wedge i_0(k) < L_0$  **then**

$$i_0(k) = U_0$$

**else if**  $I_0 > 0 \wedge i_0(k) > U_0$  **then**

$$i_0(k) = L_0$$

**end if**

$$i_0(k) = i_0(k) + M_4[i_4(k)]$$

---

---

**Algorithm 4** Update address of the next element  $i_0(k)$  in a data memory.

$MODE = STEP\_IDX\ BND$

---

$$i_4(k) = i_4(k-1) + I_4$$

**if**  $I_4 < 0 \wedge i_4(k) < L_4$  **then**

$$i_4(k) = U_4$$

$$i_0(k) = i_0(k-1) + I_0$$

**else if**  $I_4 > 0 \wedge i_4(k) > U_4$  **then**

$$i_4(k) = L_4$$

$$i_0(k) = i_0(k-1) + I_0$$

**end if**

**if**  $I_0 < 0 \wedge i_0(k) < L_0$  **then**

$$i_0(k) = U_0$$

**else if**  $I_0 > 0 \wedge i_0(k) > U_0$  **then**

$$i_0(k) = L_0$$

**end if**

---

**Algorithm 5** Update address of the next element  $i_0(k)$  in a data memory.

$MODE = (USE\_IDX|STEP\_IDX|BND)$

```

 $i_4(k) = i_4(k - 1) + I_4$ 
if  $I_4 < 0 \wedge i_4(k) < L_4$  then
     $i_4(k) = U_4$ 
     $i_0(k) = i_0(k - 1) + I_0$ 
else if  $I_4 > 0 \wedge i_4(k) > U_4$  then
     $i_4(k) = L_4$ 
     $i_0(k) = i_0(k - 1) + I_0$ 
end if
if  $I_0 < 0 \wedge i_0(k) < L_0$  then
     $i_0(k) = U_0$ 
else if  $I_0 > 0 \wedge i_0(k) > U_0$  then
     $i_0(k) = L_0$ 
end if
 $i_0(k) = i_0(k) + M_4[i_4(k)]$ 

```

Table 1: Overview of DFU operation parameters

Parameter	Description	Default Value	API function
$op$	DFU operation	-	pb2dfu_start_op, pb2dfu_restart_op
$N$	length of the input vectors	0	pb2dfu_set_cnt, pb2dfu_start_op
$REP$	the number of times the following DFU operation will be restarted	1	pb2dfu_set_repetitions
$M_i$	mapping of virtual vector $M_i$ to physical data memory $A, B, C, D$	$M_i = A$	pb2dfu_set_bank, pb2dfu_set_fulladdr
$O_i$	address of the first element in virtual vector $M_i$	0	pb2dfu_set_addr, pb2dfu_set_fulladdr
$I_i$	increment of address generator $i$	0	pb2dfu_set_inc
$L_i$	lower boundary of address generator $i$	0	pb2dfu_set_bound_addr
$U_i$	upper boundary of address generator $i$	0xffff	pb2dfu_set_bound_addr
$MODE_i$	mode of address updating in address generator $i$	0	pb2dfu_set_agflags

## 5 Changes from the previous version

The previous version of firmware API (PB2 API) has been prepared for the previous version of BCE accelerator (called *DFU\_FP01*). A new version of BCE accelerators has more features which can help to compute more complex functions easier.

Main changes are mentioned in the following list. The previous version is on the first line and the current version is on the second line.

- source and destination data memories
    1. They are specified by operation (function `pb2dfu_start_op`, `pb2dfu_restart_op`)
    2. They are assigned to virtual vectors with functions `pb2dfu_set_bank` and `pb2dfu_set_fulladdr`
  - address in data memory
    1. is consist of memory bank and 8bit address (`pb2dfu_set_bank`, `pb2dfu_set_addr`, `pb2dfu_set_fulladdr`)
    2. is one 16bit address (`pb2dfu_set_addr`, `pb2dfu_set_fulladdr`)
- NOTE: Behaviour of the function `pb2dfu_set_bank` is changed.
- Mode of addressing vectors
    1. only basic linear addressing of elements in all vectors (`pb2dfu_set_bank`, `pb2dfu_set_addr`, `pb2dfu_set_fulladdr`, `pb2dfu_set_inc`, `pb2dfu_set_restart_addr`)
    2. several addressing modes - linear and indexed (`pb2dfu_set_addr`, `pb2dfu_set_fulladdr`, `pb2dfu_set_inc`, `pb2dfu_set_bound_addr`, `pb2dfu_set_agflags`)
  - Communication protocol between accelerator and the host CPU
    1. is more complicated - more functions `pb2mb_read_data`, `pb2mb_write_data`, `pb2mb_eoc`, `pb2mb_req_reset`, `pb2mb_reset` must be used
    2. is simpler. There is also another way how to transfer data. Functions `mbpb_exchange_data`, `read_bce_cmem_u8`, `read_bce_cmem_u16`, `write_bce_cmem_u16`.

## 6 Application Programming Interface

### 6.1 enum dfu\_bce\_operations

#### Purpose

enum dfu\_bce\_operations - codes of operations supported by accelerator's data flow unit

#### Synopsis

```
enum dfu_bce_operations {  
    DFU_VCOPY,  
    DFU_VADD,  
    DFU_VMUL,  
    DFU_VMAC,  
    DFU_DPROD,  
    DFU_VSUB,  
    DFU_VMSUBAC  
};
```

#### Constants

<i>DFU_VCOPY</i>	vector copying $M0[i] = M1[j]$
<i>DFU_VADD</i>	vector addition $M0[i] = M1[j] + M2[k]$
<i>DFU_VMUL</i>	vector multiplication $M0[i] = M1[j] * M2[k]$
<i>DFU_VMAC</i>	vector multiply-accumulation $M0[i] = M3[l] + (M1[j] * M2[k])$
<i>DFU_DPROD</i>	vector dot product $M0[i] = \text{sum}(M1[j] * M2[k])$
<i>DFU_VSUB</i>	vector subtraction $M0[i] = M1[j] - M2[k]$
<i>DFU_VMSUBAC</i>	vector multiply-accumulation $M0[i] = M3[l] - (M1[j] * M2[k])$

#### Description

These codes are used with functions `pb2dfu_start_op` and `pd2dfu_restart_op` to select operation performed in the accelerator.

---

### 6.2 enum dfu\_ag\_indices

#### Purpose

enum dfu\_ag\_indices - Indices of DFU Arguments / Address Generators.

## Synopsis

```
enum dfu_ag_indices {  
    DFUAG_0,  
    DFUAG_1,  
    DFUAG_2,  
    DFUAG_3  
};
```

## Constants

<i>DFUAG_0</i>	address generator 0 - virtual vector M0 - result of an operation
<i>DFUAG_1</i>	address generator 1 - virtual vector M1 - the first argument of an operation
<i>DFUAG_2</i>	address generator 2 - virtual vector M2 - the second argument of an operation
<i>DFUAG_3</i>	address generator 3 - virtual vector M3 - the third argument of an operation

## Description

These codes are used for selection which address generator will be adjusted with functions `pb2dfu_set...` (`_addr`, `_bank`, `_fulladdr`, `_inc`, `_bound_addr`, `_agflags`).

---

## 6.3 enum dfu\_idxag\_indices

### Purpose

enum `dfu_idxag_indices` - Indices of slave (auxiliary) DFU Arguments / Address Generators.

### Synopsis

```
enum dfu_idxag_indices {  
    DFUAG_IDX_0,  
    DFUAG_IDX_1,  
    DFUAG_IDX_2,  
    DFUAG_IDX_3  
};
```

## Constants

<i>DFUAG_IDX_0</i>	slave (index) address generator 0
<i>DFUAG_IDX_1</i>	slave (index) address generator 1
<i>DFUAG_IDX_2</i>	slave (index) address generator 2
<i>DFUAG_IDX_3</i>	slave (index) address generator 3

## Description

These codes are used for selection which auxiliary address generator will be adjusted with functions `pb2dfu_set...` (`_addr`, `_bank`, `_fulladdr`, `_inc`, `_bound_addr`, `_agflags`).

---

## 6.4 enum dfu\_data\_memories

### Purpose

enum `dfu_data_memories` - Physical Local memory banks.

### Synopsis

```
enum dfu_data_memories {  
    MBANK_A,  
    MBANK_B,  
    MBANK_C,  
    MBANK_D  
};
```

## Constants

<i>MBANK_A</i>	Use data memory A
<i>MBANK_B</i>	Use data memory B
<i>MBANK_C</i>	Use data memory C
<i>MBANK_D</i>	Use data memory D

## Description

These constants are used with function `pb2dfu_set_bank` to assign a physical memory to an address generator.

---

## 6.5 enum dfu\_ag\_flags

### Purpose

enum dfu\_ag\_flags - Address generator flags/modes.

### Synopsis

```
enum dfu_ag_flags {  
    AGFL_USE_IDX,  
    AGFL_STEP_IDXBND  
};
```

### Constants

*AGFL\_USE\_IDX*

Offset each address produced in the main AG by an index delivered from the slave AG.

*AGFL\_STEP\_IDXBND*

Increment address of the main AG only when the slave AG is reaching boundary.

### Description

This is used in **pb2dfu\_set\_agflags**.

### NOTE

The AGFL\_USE\_IDX and AGFL\_STEP\_IDXBND flags can be used independently. That is, it is possible to step the main AG by the boundary condition of the slave AG, while not using the indices received from it. However, the slave AG will still read data from the BRAM even if they are not used in the main AG.

---

## 6.6 function mbpb\_exchange\_data

### Purpose

mbpb\_exchange\_data - Exchange data with the host CPU.

### Synopsis

```
unsigned char mbpb_exchange_data (unsigned char data)
```

### Arguments

*data*

8bit data sending to the host CPU

### Description

Exchange byte with microblaze using the barrier synchronization. This function blocks.

## Return Value

The function returns 8bit data received from the host CPU.

---

## 6.7 function pb2mb\_report\_running

### Purpose

pb2mb\_report\_running - Inform the host CPU that firmware is running

### Synopsis

```
void pb2mb_report_running ()
```

### Description

Sets the R and B bits in the CFG Status register to report that the firmware has successfully started and is busy. This function does NOT block. It should be called immediately when picoblaze starts up.

### NOTE

This could be integrated into picoblaze C library to be called automatically upon startup.

## Return Value

The function doesn't return any value.

---

## 6.8 function write\_bce\_id\_to\_cmem

### Purpose

write\_bce\_id\_to\_cmem - Write the BCE ID into the cfg output memory.

### Synopsis

```
void write_bce_id_to_cmem (unsigned char fam1)
```

### Arguments

*fam1*

BCE ID which can be read with function wal\_get\_id in the host CPU application.

### Description

This function should be used as reaction to WAL\_OP\_GETID operation.

## Return Value

The function doesn't return any value.

---

## 6.9 function write\_dfu\_caps\_to\_cmem

### Purpose

write\_dfu\_caps\_to\_cmem - copy DFU bitmap of capabilities from DFU to control memory

### Synopsis

```
void write_dfu_caps_to_cmem ()
```

### Description

Write the whole DFU capabilities bitmap (256bits = 32 B = 8 words) into the cfg output memory, starting at address 0x81 of the ctrl/status memory. This function should be used as reaction to WAL\_OP\_GETCAP operation.

## Return Value

The function doesn't return any value.

---

## 6.10 function read\_bce\_cmem\_u8

### Purpose

read\_bce\_cmem\_u8 - Read u8 value from the BCE input cfg region.

### Synopsis

```
unsigned char read_bce_cmem_u8 (unsigned char cfgaddr, unsigned char byteidx)
```

### Arguments

### Description

*cfgaddr* = word address in the config mem. *byteidx* = [0; 3] Byte index, 0 = LL, 3 = HH

## Return Value

The function doesn't return any value.

---

## 6.11 function read\_bce\_cmem\_u16

### Purpose

read\_bce\_cmem\_u16 - Read u16 value from the BCE input cfg region.

### Synopsis

```
unsigned int read_bce_cmem_u16 (unsigned char cfgaddr, unsigned char widx)
```

### Arguments

### Description

*cfgaddr* = word address in the config mem. *widx* = 0; 2 Word index, 0 = Lo, 2 = Hi

### Return Value

The function doesn't return any value.

---

## 6.12 function write\_bce\_cmem\_u16

### Purpose

write\_bce\_cmem\_u16 - Write u16 value into BCE cfg memory.

### Synopsis

```
void write_bce_cmem_u16 (unsigned char cfgaddr, unsigned char widx, unsigned int dt)
```

### Arguments

### Description

The *cfgaddr* shall be in the output status region, ie. 0x80 - 0xff. *widx* = 0; 2 Word index, 0 = Lo, 2 = Hi

### Return Value

The function doesn't return any value.

---

## 6.13 function pcnt\_get\_dfutime

### Purpose

pcnt\_get\_dfutime - Get time of the last DFU operation.

## Synopsis

```
unsigned int pcnt_get_dfutime ()
```

## Description

Get the running time in clock-cycles of the last DFU operation executed.

## Return Value

The function returns value of 16bit DFU operation-time counter.

---

## 6.14 function **pcnt\_get\_prptime\_lo**

### Purpose

**pcnt\_get\_prptime\_lo** - Get value of the programm running-time counter (lower 16bit).

### Synopsis

```
unsigned int pcnt_get_prptime_lo ()
```

### Description

Get the program running time in clock-cycles. The counter is 32b in total, thus we provide two functions to access its lo/hi part.

### Return Value

The function returns lower 16bit of the program running-time counter.

---

## 6.15 function **pcnt\_get\_prptime\_hi**

### Purpose

**pcnt\_get\_prptime\_hi** - Get value of the programm running-time counter (higher 16bit).

### Synopsis

```
unsigned int pcnt_get_prptime_hi ()
```

### Description

Get the program running time in clock-cycles. The counter is 32b in total, thus we provide two functions to access its lo/hi part.

## Return Value

The function returns higher 16bit of the program running-time counter.

---

## 6.16 function `pcnt_reset_prgtime`

### Purpose

`pcnt_reset_prgtime` - Reset the program running-time counter.

### Synopsis

```
void pcnt_reset_prgtime ()
```

### Description

#### Return Value

The function doesn't return any value.

---

## 6.17 function `get_dfulic`

### Purpose

`get_dfulic` - Read licence flag from DFU.

### Synopsis

```
unsigned int get_dfulic ()
```

### Description

The function reads and returns DFU license counter.

#### Return Value

The function returns flag which indicates that the DFU licence is run out.

---

## 6.18 function `pb2dfu_wait4hw`

### Purpose

`pb2dfu_wait4hw` - PB will wait for end of computation

### Synopsis

```
unsigned char pb2dfu_wait4hw ()
```

## Description

The function waits for finishing computation in the accelerator. The function should be called before subsequent run of the next operation. The next operation can be prepared before the waiting to speed up the entire computation.

## Return Value

Zero if ok, Non-zero on DFU error

---

## 6.19 function pb2dfu\_start\_op

### Purpose

pb2dfu\_start\_op - start operation in DFU with specified length of data vectors

### Synopsis

```
void pb2dfu_start_op (unsigned char op, unsigned int cnt)
```

### Arguments

<i>op</i>	DFU operation (constants DFU_XXX)
<i>cnt</i>	length of input data vectors

## Description

The function covers two functions (pb2dfu\_set\_cnt and pb2dfu\_restart\_op).

## Return Value

The function doesn't return any value.

---

## 6.20 function pb2dfu\_restart\_op

### Purpose

pb2dfu\_restart\_op - start operation in DFU

### Synopsis

```
void pb2dfu_restart_op (unsigned char op)
```

### Arguments

<i>op</i>	DFU operation (constants DFU_XXX)
-----------	-----------------------------------

## Description

All parameters of the operation must be set before this function. All parameters are registered and so only changed parameters from previous operations must be set. On the other hand, the operation must be always set because the function starts a required operation in the DFU.

## Return Value

The function doesn't return any value.

---

## 6.21 function pb2dfu\_set\_cnt

### Purpose

pb2dfu\_set\_cnt - set length of input data vectors for the next operation

### Synopsis

```
void pb2dfu_set_cnt (unsigned int cnt)
```

### Arguments

<i>cnt</i>	length of input data vectors
------------	------------------------------

### Description

The function sets length of the input data vectors. The simple operations (as VADD, VMULT) will be performed *cnt*-times as one pipelined operation.

### Return Value

The function doesn't return any value.

---

## 6.22 function pb2dfu\_set\_addr

### Purpose

pb2dfu\_set\_addr - set the base address of vector for the given DFU argument

### Synopsis

```
void pb2dfu_set_addr (unsigned char dfuag, unsigned int addr)
```

## Arguments

*dfuag* select the DFU argument number (constant DFUAG\_x)  
*addr* the initial address of the vector

## Return Value

The function doesn't return any value.

---

## 6.23 function pb2dfu\_set\_bank

### Purpose

pb2dfu\_set\_bank - select bank for specified memory

### Synopsis

```
void pb2dfu_set_bank (unsigned char dfuag, unsigned char mbank)
```

## Arguments

*dfuag* select the DFU argument number (constant DFUAG\_x)

## Return Value

The function doesn't return any value.

---

## 6.24 function pb2dfu\_set\_fulladdr

### Purpose

pb2dfu\_set\_fulladdr - set full address (bank and offset) of the first element in the vector

### Synopsis

```
void pb2dfu_set_fulladdr (unsigned char dfuag, unsigned char mbank, unsigned int addr)
```

## Arguments

*dfuag* select the DFU argument number (constant DFUAG\_x)  
*mbank* the memory bank which will be used for the next operation (constant MBANK\_x)  
*addr* the initial address of the vector

## Return Value

The function doesn't return any value.

---

## 6.25 function `pb2dfu_set_inc`

### Purpose

`pb2dfu_set_inc` - set the stride of the vector for the DFU argument

### Synopsis

```
void pb2dfu_set_inc (unsigned char dfuag, int inc)
```

### Arguments

<i>dfuag</i>	select the DFU argument number (constant <code>DFUAG_x</code> )
<i>inc</i>	increment between two elements of vector

## Return Value

The function doesn't return any value.

---

## 6.26 function `pb2dfu_set_bound_addr`

### Purpose

`pb2dfu_set_bound_addr` - set boundary addresses for vector accesses

### Synopsis

```
void pb2dfu_set_bound_addr (unsigned char dfuag, unsigned int lo_bound, unsigned int hi_bound)
```

### Arguments

<i>dfuag</i>	select the DFU argument number (constant <code>DFUAG_x</code> )
<i>lo_bound</i>	lower address boundary
<i>hi_bound</i>	higher address boundary

## Return Value

The function doesn't return any value.

---

## 6.27 function pb2dfu\_set\_agflags

### Purpose

pb2dfu\_set\_agflags - set operation flags/mode of the specified address generator (DFU argument)

### Synopsis

```
void pb2dfu_set_agflags (unsigned char dfuag, unsigned char agflags)
```

### Arguments

<i>dfuag</i>	select the DFU argument number (constant DFUAG_x)
<i>agflags</i>	bitmap of flags to set (constants AGFL_x)

### Return Value

The function doesn't return any value.

---

## 6.28 function pb2dfu\_set\_repetitions

### Purpose

pb2dfu\_set\_repetitions - set the number of repetitions of a DFU operation.

### Synopsis

```
void pb2dfu_set_repetitions (unsigned char nrep)
```

### Arguments

<i>nrep</i>	the number of times the following DFU operation will be restarted.
-------------	--

### Return Value

The function doesn't return any value.

---

## References

- [1] J. Kadlec and all, "D2.1 - preliminary report on platform dependent parameters and optimizations," August 2010. SMECY project deliverable D2.1.
- [2] R. Bartosinski, "UTIA EdkDSP Platform, WAL - Worker Abstraction Layer," March 2011. SMECY project deliverable D2.2.