Advanced OEM Solutions

Software API

OEM-PA/OEM-MC/FMC

Version 1.2

REVISION HISTORY

Date	Version	Description
2014/12/12	1.0	General Edit
2015/11/01	1.0	Documentation Reorganization
2016/05/01	1.0	Small edits/Typo fixes
2016/09/01	1.1	FW Recovery Time information (§4.4.10, §6.4, §7.4) § 5 FIFO API added § 6 Multiple Channel (OEM-MC): reorganized with former application notes document and added explanations
2016/11/30	1.2	§ 4.4.4 Trigger: Paragraphs reordered (4.4.4.2 was missing) § 4.4.4.5 Temperature: Update for checking during acquisition § 4.4.8.3 Echo tracking (IF): corrections to OEMPA file example § 4.4.9 Data loss: addition of GetLostUSB3 counter § 4.5.2 Specific device: addition of 256/256 device § 7.2 Setup: added detail on FMCElementStep § 7.4 FW Recovery Time and FMC Timeslot Considerations: additional details about TimeSlot and SubCycle Time Other minor edits

CONTENTS

1. Introduction 7

1.1 In-Depth Presentation 7

1.1.1 OEM-PA Driver API 8

1.1.2 Folders and files...9

1.1.3 Custom Installer...11

1.1.4 File formats...12

1.1.5 Shared memory 13

1.2 APIs...13

1.2.1 High Level API (Wizard API) 13

1.2.2 Medium Level API (Customized API) 13

1.2.3 Driver API 14

1.3 Configuration Data...15

1.3.1 CFocalLaw 16

1.3.2 Memory size 16

1.4 DLL...16

1.4.1 Unmanaged code..16

1.4.2 Managed code (open source)..17

1.5 Version 18

2. High Level API (Wizard).. 19

2.1 Initialization 19

2.2 API 20

3. Medium Level API (Customized).. 20

3.1 Configuration file 21

3.2 Link with the driver 21

3.3 Initialization 22

3.4 Callback 22

3.4.1 Load Configuration File...22

3.4.2 Save Configuration File 23

3.5 API 24

3.6 Project configuration 24

4. Low Level API (Driver) 25

4.1 Project configuration 25

4.1.1 C++ 25

4.1.2 Headers 25

4.1.3 Libraries 25

4.1.4 Compilation options 26

4.2 C# Driver 26

4.3 Low Level API 26

4.3.1 General 26

4.3.2 Instantiation 28

4.3.3 Initialization 29

4.3.4 Communication 30

4.3.5 Read/write operations 34

4.3.6 Error reporting 43

4.3.7 HW Identify 45

4.4 Acquisition 46

4.4.1 Terminology 46

4.4.2 Callback function 47

4.4.3 Enable/disable 48

4.4.4 Trigger 49

4.4.5 A-scan 60

4.4.6 C-scan 71

4.4.7 Callback common inputs 78

4.4.8 HW Gates 80

4.4.9 Data loss 87

4.4.10 FW Recovery Time 88

4.5 Special features....89

4.5.1 Configuration file....89

4.5.2 Specific device....94

4.5.3 Calibration....94

4.5.4 Encoder management....99

5. FIFO API....102

5.1 Introduction....102

5.2 C++/C# API....102

5.3 Matlab example....105

6. Multiple Channel (OEM-MC)....106

6.1 OEM-MC Cycles....106

6.2 OEM-PA / OEM-MC Decimation....107

6.3 Data Acquisition Modes....107

6.3.1 Multiplexed Mode....107

6.3.2 Parallel Mode....110

6.3.3 Summary....113

6.3.4 Multiplexed vs. Parallel Example....114

6.4 FW Recovery Time....115

6.4.1 HW Acquisition Cycle....115

6.4.2 HW Replay Cycle....116

6.4.3 Minimum Time Slot....116

6.5 API....117

6.5.1 Basic....117

6.5.2 Cycle....118

6.5.3 Pulser....119

6.5.4 Receiver....120

6.6 OEMMC File Examples....122

6.6.1 Multiplexed Mode Example....122

6.6.2 Parallel Mode Examples....125

7. Full Matrix Capture (FMC) 129

7.1 Basic 133

7.2 Setup 133

7.2.1 Example 135

7.3 Acquisition 136

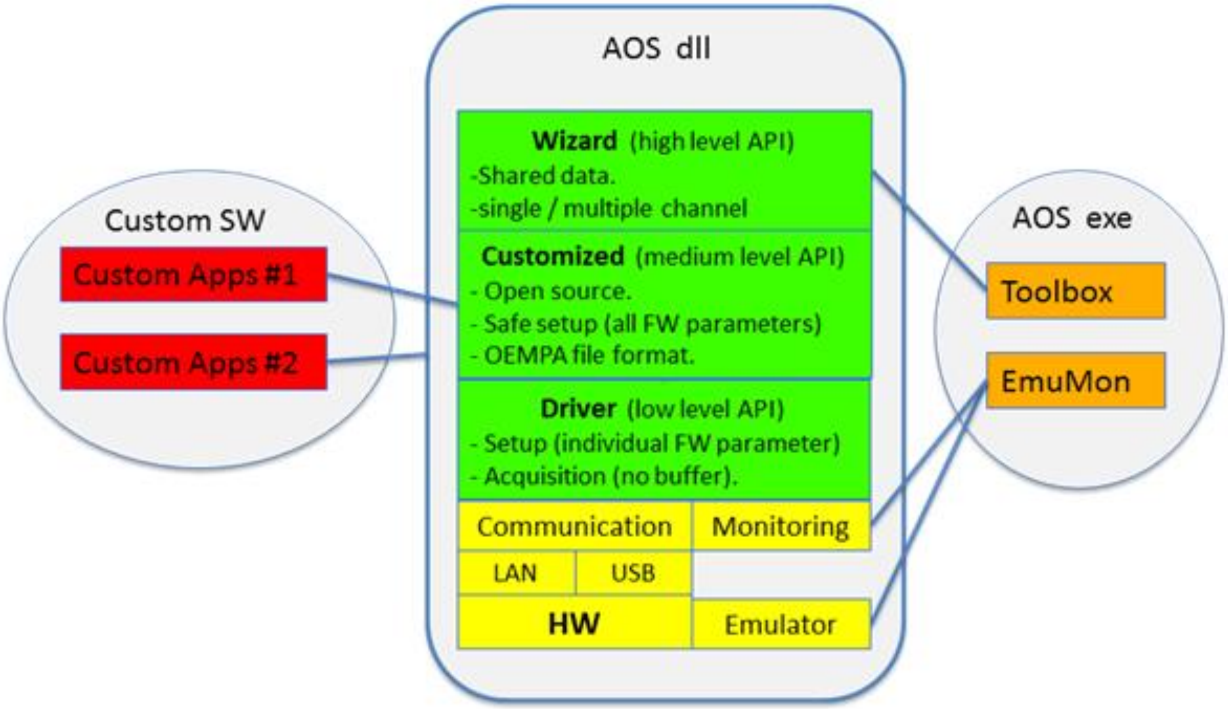
7.4 FW Recovery Time and FMC Timeslot Considerations 136

1. Introduction

This document details the APIs provided by AOS for the OEM Hardware units: OEM-PA, OEM-MC, and FMC.

The software APIs available to the customer are built to help them understand the hardware interface, provide a guideline on how to create custom software for their specific applications and also provide software which can be readily used.

A summary of the available APIs is presented below:



This document explains the **low level API**, **medium level API** and **high level API**. The **low level API** can be used to access each firmware (FW) individual parameter. The **medium level API** is a higher level of abstraction to access all FW parameters at the same time. To properly set up the device from scratch, please use it in order not to forget one of the required FW parameters. The **high level API** further simplifies the efforts by providing a simple tool that performs all the functions.

1.1 In-Depth Presentation

The following section contains detailed information of the installed software.

1.1.1 OEM-PA Driver API

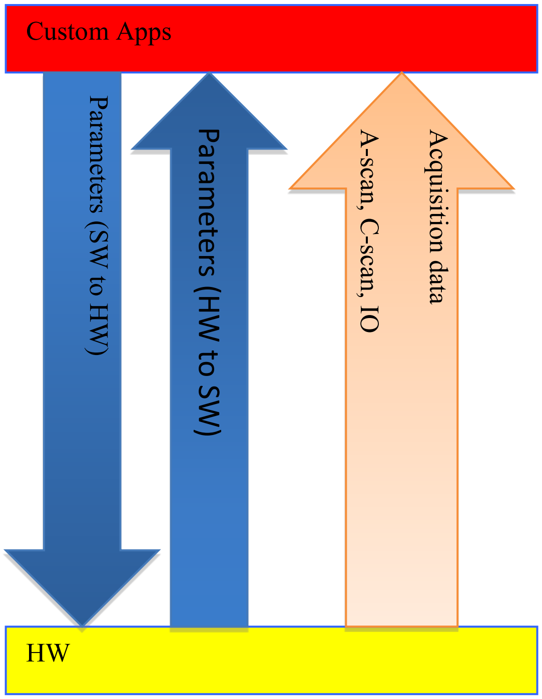
Section 4 of this document, Low Level API (Driver), explains the low level driver API. For the list of all FW parameters, see *Software_Functions_&_Parameter_List.pdf*.

The customized API is also included in this DLL, but the user cannot modify it.

1.1.1.1 Communication

The communication between the HW and the application SW is done through either Ethernet or USB. For any communication link (Ethernet, USB, socket, etc.) the communication relies on a peer-to-peer data stream exchange (between the SW and the HW).

In the case of Ethernet, the HW sends KeepAlive messages to the computer to ensure that the connection is maintained.



The Setup (FW parameters) streams are sent by the computer to the HW and acquisition data streams are sent by the HW to the computer. There are 4 different stream types: A-scan, C-scan, IO stream (encoder + digital inputs) and notification (errors and warning).

Read and write operations are packed together and sent to the device in the same stream in order to optimize communication performance. Then the device returns answers for many read operations at the same time. Read and write operations are performed asynchronously.

Communication between different applications is performed with sockets (example link between “EmuMon” and “OEMPATool” software).

The EmuMon SW can be used to monitor the communication flow.

1.1.1.2 Function groups

To configure the HW, the SW has to send all required FW parameters. Connection with the hardware must be completed before these functions can be called. Different steps are required depending on the function group:

- General functions:
  - Instantiation of an object with the class API.
  - Registration of an acquisition path: an acquisition callback function should be defined to get acquisition data.
- Connection management:
  - Communication link selection (Ethernet, USB).
  - Connection with the HW device.
  - Status management (to be notified of disconnection).
- FW parameters management:
  - Lock/unlock the communication link.
    - Only one thread can access read/write function at the same time.
  - To read/write FW UT parameters (gain, delays, etc.):
    - Before processing the input FW parameter, any write function checks the range of the input parameter.
    - Any check function can also be called directly.
    - Each write function has its own check process, fully independent of other functions. There is no global check with the low level API.

1.1.1.3 FW parameters

The communication link should be locked before calling read/write functions that rely on the communication layer, and unlocked afterwards. Neglecting to unlock will result in errors.

Caution: Communication is not immediately complete when the read/write function returns. The communication is finished only after returning from functions “CHWDevice::UnlockDevice” and “CHWDevice::Flush”. This is an important step when reading FW parameters.

1.1.2 Folders and files

After installation you will find the following folder (evaluation kit can be different):

- “Program Files“ folder
  - “C:\Program Files\AOS\OEMPA [version]”
    - DLL
      - “UTKernelDriverOEMPA.dll”: OEM-PA driver DLL.

- “UTCom.dll”: Ethernet and USB management.
- “UTKernelAPI.dll”: kernel DLL.
- “UTCustomizedDriverAPI.dll” and “UTCustomizedWizardAPI.dll”: customized API.
- EXE
  - “OEMPAApplicationExample.exe”: “OEMPAApplicationExample” SW.
  - “OEMPAFormExample.exe”: “OEMPAFormExample” SW.
  - “OEMPACustomizedExample.exe”: “OEMPACustomizedExample” SW.
  - “OEMPAWizardExample.exe”: “OEMPAWizardExample” SW.
  - “OEMPATool.exe”: “OEMPATool” SW.
  - “OEMPASector.exe”: “OEMPASector” SW.
  - “DefaultConfiguration.exe”: “DefaultConfiguration” SW.
  - “UTToolbox.exe”: “Toolbox” SW.
  - “Manager.exe”: this SW is used to kill all others (for frozen or unresponsive apps) and to access the software configuration file.
  - “UTKernelError.exe”: this SW is used to manage the display of errors and warnings.
- “C:\Program Files\AOS\OEMPA [version]\EMuMon”
  - “EmuMon.exe”: “EmuMon” SW.
- “C:\Program Files\AOS\OEMPA [version]\Documents”: documentation.
- “C:\Program Files\AOS\OEMPA [version]\StringTable” used by the high level API to manage the language.
- Common application data folder (“C:\ProgramData“ for Windows 7)
  - “C:\ProgramData\AOS\OEMPA [version]\Cfg”
    - Configuration files
      - “CfgKernel.txt”: SW configuration and preferences of the kernel DLL.
      - “CfgOEMPASector.txt”: SW configuration of the “OEMPASector” SW.
      - “OEMPAInit.txt”: default custom initialization data of the HW.
      - “OEMPAEmuMonInit.txt”: default custom initialization of “EmuMon” SW.
  - “C:\ProgramData\AOS\OEMPA [version]\Files” user files (some folders are created during installation and others at SW application run time)

        - Subfolder “EmuMon”: storage of EmuMon format (acquisition data) files and default flash file (hardware flash like that contains the device definition - for example 16/16, 32/32, 32/128, etc.) called “EmuMonFlashSWSection.txt” (EmuMon needs to know which kind of device is emulated).
        - Subfolder “OEMPA”: storage of OEMPA format (customized API) files.
        - Subfolder “Kernel”: storage of “Kernel” format files.
        - Subfolder “OEMPASector”: storage of “OEMPASector” format files.
        - Subfolder “Palette”: default palette for “OEMPATool” and “OEMPASector”.
- “C:\ProgramData\AOS\OEMPA [version]\Sources”
    - “UTKernel” folder in which you can find the following solutions:
        - “OEMPAApplicationExample.sln”
        - “OEMPAFormExample.sln” (C#)
        - “OEMPACustomizedExample.sln”
        - “OEMPAWizardExample.sln”
        - “UTKernelMatlab.sln”
        - “UTCustomizedAPI.sln”
    - “OEMPATool\OEMPATool.sln” solution for “OEMPATool”
    - “OEMPASector\OEMPASector.sln” solution for “OEMPASector”
    - “UserHeaders” is the folder in which user headers are stored

For Windows XP, the common application data folder is “C:\Documents and Settings\All Users\Application Data” (instead of “C:\ProgramData” for Windows 7).

The Evaluation Kit has a slightly different folder structure:

- “C:\Program Files\AOSEvaluation\OEMPA [version]” instead of “C:\Program Files\AOS\OEMPA [version]”
- “C:\ProgramData\AOSEvaluation\OEMPA [version]” instead of “C:\ProgramData\AOS\OEMPA [version]”

1.1.3 Custom Installer

If you want to build your installer for your own application, you must include the following file:

"C:\ProgramData\AOS\OEMPA [version]\Cfg\CfgKernel.txt" (Windows 7)
"C:\Documents and Settings\All Users\Application Data\AOS\OEMPA [version]\Cfg\CfgKernel.txt" (Windows XP)

This file is required by the AOS DLLs. You also have to include “UTKernelError.exe” and all required DLLs (see paragraph 1.4 “DLL” in this document for low-level and medium-level driver

DLLs). Notice that if “UTKernelAPI.dll” is used, then the folder “StringTable” and all its sub folders and files should be also packed in your installer. “StringTable” can be found in the following folder:

“C:\Program Files\AOS\OEMPA [version]\StringTable”

“C:\Program Files (x86)\AOS\OEMPA [version]\StringTable” (only for the 32 bits kit on a 64 bits computer)

1.1.4 File formats

The following file formats are available:

- **OEMPA File** or **configuration file** (text and binary). These files are used by the medium level API to configure all FW parameters for the device to work properly.
  - Distinction should be made between the default configuration loaded by the driver at connection time, and setup files loaded after connection by the user (see paragraph 4.5.1 “Configuration file” in this document).
  - Text format is slower to access than binary format.
- Kernel files (“Kernel” format), text, and binary.
  - All data (including the beam of each focal law) can be saved or only input data of the wizard.
  - These are high level parameters (Steering angle, aperture size, etc.)
- FW-update text file.
  - These special files are used to update the firmware.
  - These files are generated by AOS only.
  - “EmuMon” is the only SW that should use these files.

Since the same extension is used for different file formats, it would be good practice to store these files in different folders depending on their function. The path to the most recent folder to store these files is saved in the kernel configuration file. By default it is:

- “C:\ProgramData\AOS\OEMPA [version]\Files”
  - Subfolder “EmuMon”: storage of EmuMon format (acquisition data) files and default flash file (hardware flash like that contains the device definition, for example 16/16, 32/32, 32/128, etc.) that is “EmuMonFlashSWSection.txt” (EmuMon needs to know which kind of device is emulated).
  - Subfolder “OEMPA”: storage of OEMPA format (customized API) files.

  - Subfolder “Kernel”: storage of “Kernel” format files.
  - Subfolder “OEMPASector”: storage of “OEMPASector” format files.

1.1.5 Shared memory

A single folder should be used for all applications in order for them to share some of the same RAM memory (not required for all shared memory).

The kernel for all software is packed into a single DLL (“UTKernelAPI.dll”). The driver for OEM-PA is the DLL named “UTKernelDriverOEMPA.dll”, and all OEM-PA applications are linked with this DLL.

The kernel is built with object-oriented programming and all their data members are shared between all applications. This means, for example, that you can run any application SW and the “Toolbox” SW (or the “DefaultConfiguration” SW) simultaneously, provided they are installed in the same folder. In such a case, the Wizard objects can be edited from the toolbox, and the result will be directly available for the application SW. Acquisition data managed by the kernel will be shared as well in the future.

With a 64-bit computer, the shared memory is only available with the 64-bit kit (not with the 32-bit kit).

1.2 APIs

1.2.1 High Level API (Wizard API)

The Wizard consists of:

- Input parameters (specimen, probes, scan)
  - A special input parameter is used to update output parameters from inputs.
- Output parameters (delays, etc.)

The Wizard is useful to build configuration files (“OEMPA files”) with the proper focal laws required for your application.

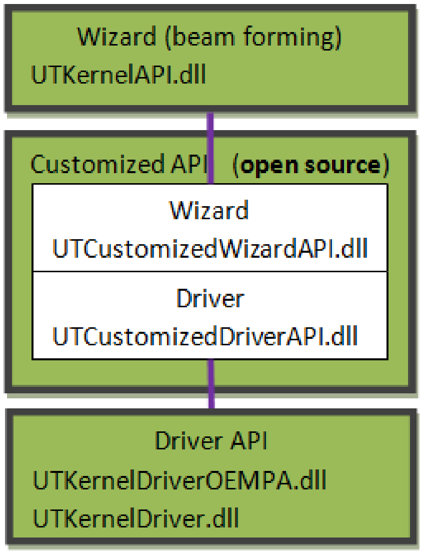
It is possible to use the wizard without being connected to the device, but in this case it is not possible to check that your scanning is compatible with the device (32/128 or 16/16 etc.).

The “Toolbox” is only a graphical user interface to display objects in the kernel database. Keep in mind that you can run the Kernel (i.e. access the Kernel via the API) without the Toolbox GUI (the Wizard is included in the kernel).

1.2.2 Medium Level API (Customized API)

The customized API is made of two parts:

- The driver customized API
    - Definition of main features (configuration data, main functions to exchange configuration data with the computer memory, the hard disk, or the device).
- The wizard customized API
    - To exchange configuration data or simpler data with the Wizard (toolbox) for beam forming.



The customized API is **open source**. It can be modified by the user (for example to optimize the memory size), but would then need to be managed and preserved when there are new SW updates and releases.

The driver ("UTKernelDriverOEMPA.dll") calls the customized API for two tasks for which the user may not alter the functions:

- To load the default configuration file ("OEMPA_ReadFileWriteHW").
- To build the configuration file used for calibration ("OEMPA_CreateDefaultSetupFile").

1.2.3 Driver API

The following lists out the details of the Driver API. These are discussed further in detail in this document.

- The list of functions to access FW individual parameters.
    - See the document *Software_Functions_&_Parameter_List.pdf* for the list of all FW parameters.

- Acquisition: delivery of the acquisition data stream (A-scan and C-scan data) coming from the FW.
    - FW parameters used to acquire A-scan.
    - FW parameters used to acquire C-scan.
    - HW gates: wedge delay and echo tracking.
    - Data loss.
- Configuration file: those files that are used to pack together all the FW parameters. The configuration data is managed by the customized API, and is explained in this document and also in the "OEMCustomizedAPI" document:
    - The data structure to store **all FW Setup Parameters**, also named **configuration data**.
    - Many functions delivered to exchange configuration data with:
        - The hardware device.
        - The computer memory**.**
        - The computer hard disk. For example: the function to setup the device serves as a wrapper of many smaller low level functions of the driver to write parameters to the hardware.
- Specific device: the 128/128 uses a special connection.
- Calibration: some devices can require an electronic calibration to work properly.

Specific feature for each HW platform is specified in the document "HW Driver API".

1.3 Configuration Data

These are configuration data structures and classes used to properly setup an OEM-PA device. There are 3 main items:

Structures	Comment	SizeOf (bytes)
structRoot	Main FW registers.	4424
structCycle	FW parameters that are cycle dependent.	1168
CFocalLaw	FW parameters to setup focal laws (cycle dependent).	18124

Because it is open source, all data structures are very simple. Each structure (except "CFocalLaw") uses only static arrays. There are no restrictions and any possible setup to configure the device is possible: for example, you can manage 4096 cycles with 64 focal laws (DDF) with aperture of 128 elements and 4 gates. This is why the memory size is not optimized (even if you don't use some features the corresponding memory is allocated).

1.3.1 CFocalLaw

The class “CFocalLaw” is the most complex class. The following are some useful comments about its members:

- “iElementCount” is the element count in the aperture “adwElement”.
  - This value should be the same as “iPrmCount”.
  - This is also the first dimension of the array “afDelay”.
- “iPrmCount” is the element count in the array “afPrm” (width for emission and gain for reception).
  - This value should be the same as “iElementCount”.
- “iFocalCount” is the focal law count (greater than 1 in case of DDF).
  - This is the element count in the array “adFocalTimeOfFlight”.
  - This is the 2d dimension of the array “afDelay”.
- “iDelayCount” is the total delay count, and is the product of “iElementCount” and “iFocalCount”.

1.3.2 Memory size

The size of one cycle is about 38 Kbytes. You can compare the allocation size of 64 cycles:

Data structure	**Allocated memory size (Kbytes)**
Customized API	2338
Wizard API (“Toolbox”)	147

And for 4096 cycles => 149 Mbytes. But remember that this memory is allocated only when the setup file is loaded.

1.4 DLL

1.4.1 Unmanaged code

Only the **Customized API** is open source (2 DLLs):

- “UTCustomizedWizardAPI.dll”: required functions to exchange **configuration data** with the Wizard.
- “UTCustomizedDriverAPI.dll”: format of the **configuration data** and the required functions to exchange it with:
  - The hardware device.

- The computer memory.
- The computer hard disk.

All other DLLs used by the driver are proprietary source. Here are the two main DLLs:

- “UTKernelDriverOEMPA.dll”: low level functions to access the hardware device.
- “UTKernelDriver.dll”: high level communication functions and functions that are not hardware device dependent.

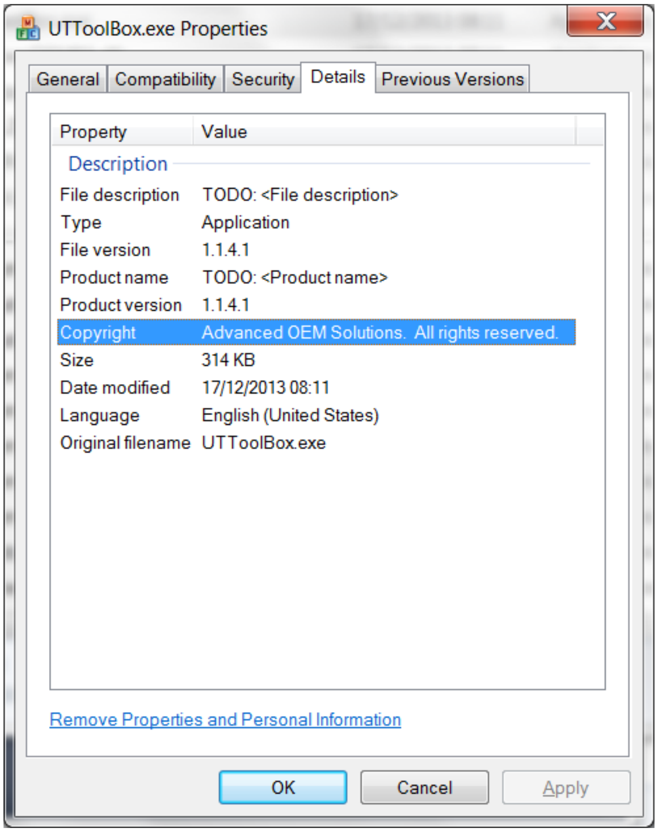
As a side note, “UTKernelDriverOEMPA.dll” and “UTKernelDriver.dll” are also internally linked with “UTCom.dll” (communication) and “win_stub.dll”.

1.4.2 Managed code (open source)

- “csDriverOEMPA”: low level functions.
- “csWizardTemplateLinear”: example of using the wizard.

1.5 Version

For any MFC item (exe or dll), the version can be found from windows explorer, here is the example of the Toolbox:



Property	Value
Description	
File description	TODO: <File description>
Type	Application
File version	1.1.4.1
Product name	TODO: <Product name>
Product version	1.1.4.1
Copyright	Advanced OEM Solutions. All rights reserved.
Size	314 KB
Date modified	17/12/2013 08:11
Language	English (United States)
Original filename	UTToolBox.exe

For other unmanaged DLLs the version is exported through a "`VS_FIXEDFILEINFO`" data:

```
VS_FIXEDFILEINFO g_VS_FIXEDFILEINFO;
```

Here is one way to read it:

```
VS_FIXEDFILEINFO *pVS_FIXEDFILEINF;

pVS_FIXEDFILEINF = (VS_FIXEDFILEINFO*)GetProcAddress(pModule,"g_VS_FIXEDFILEINFO");
if(pVS_FIXEDFILEINF)
{
	sprintf(pFileVersion,"%d.%d.%d.%d",
		(int)HIWORD(pVS_FIXEDFILEINF->dwFileVersionMS),
		(int)LOWORD(pVS_FIXEDFILEINF->dwFileVersionMS),
		(int)HIWORD(pVS_FIXEDFILEINF->dwFileVersionLS),
		(int)LOWORD(pVS_FIXEDFILEINF->dwFileVersionLS));
}
```

For managed DLL "AssemblyVersionAttribute" is used to specify the version of the assembly.

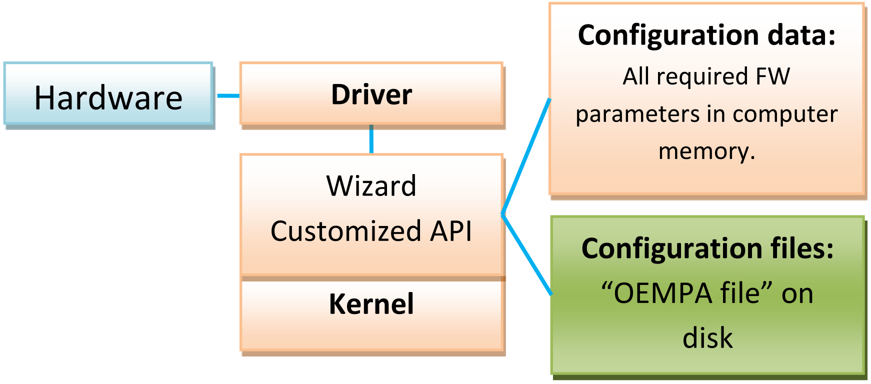
In the case of the “UTKernelDriverOEMPA.dll” the following functions are exported:

```
DRIVEROEMPA_API const wchar_t* WINAPI OEMPA_GetVersion();
DRIVEROEMPA_API DWORD WINAPI OEMPA_GetVersionMajor();
DRIVEROEMPA_API DWORD WINAPI OEMPA_GetVersionMinor();
DRIVEROEMPA_API const char OEMPA_GetVersionLetter();
```

The last function is exported from the version 1.1.5.1b.

2. High Level API (Wizard)

The wizard customized API is included in the Kernel to exchange configuration data between the Kernel, the computer memory, and the disk. An example application using the High Level API (“OEMPAWizardExample”) is described in *Software_Examples.pdf*.



2.1 Initialization

In the same way that you register the driver customized API callback function (described in 3.3 Initialization), you can also register the wizard customized API callback function (see “COEMPAWizardExampleView::COEMPAWizardExampleView” found in the source code of “OEMPAWizardExample”):

```
OEMPA_SetCallbackCustomizedWizardAPI(CallbackCustomizedWizardAPI);
```

The process to build configuration data from the Wizard calls this callback two times (see functions “OEMPA_ReadWizardSingleChannel” and “OEMPA_ReadWizardMultiChannel”):

- At the beginning of the process.
    - To get the correction table input parameter.
- At the exit of the load process.

2.2 API

General single and multiple acquisition channel API exchanges data between the Kernel and configuration data (data that are compatible with OEM-PA):

Function name	Comment
OEMPA_ReadWizardWriteFile	Save configuration file from the Wizard.
OEMPA_ReadWizard	Update configuration data in computer memory from the Wizard.
OEMPA_SetCallbackCustomizedWizardAPI	To register the callback function.

Other functions are also available to take into account additional input parameters for single acquisition channel API only:

Function name	Comment
OEMPA_ReadWizardWriteFile	Save configuration file from the Kernel Wizard.
OEMPA_ReadWizard	Update configuration data in computer memory from the Kernel Wizard.
OEMPA_WriteWizard	Update the wizard with input parameters.

You have to include a header. Here is an example (see “stdafx.h”):

```
#include"UTKernelDriver.h"//low level API used by the customized API
#include"UTKernelDriverOEMPA.h"//low level API used by the customized API
#include"OEMPACustomizedAPI.h"//to use the driver customized API
#include"OEMPACustomizedWizardAPI.h"//to use the kernel customized API
#include"UTKernelAPI.h"//to use high level API (Kernel Wizard)
```

Link the project with libraries: “UTKernelAPI.lib”, “UTKernelDriver.lib”, “UTKernelDriverOEMPA.lib”.

3. Medium Level API (Customized)

An example application using the Medium Level API (“OEMPACustomizedExample”) is described in *Software_Examples.pdf*. The Medium Level API has the following features:

- The definition of structures and classes to store the configuration data.
- A list of functions to manage configuration data and configuration files.
- Two callback functions to register at initialization time (at the same time acquisition callback functions are registered).
    - One callback for the driver customized API.

- One callback for the wizard customized API.

3.1 Configuration file

Configuration files can be either text or binary files, and are required to properly setup an OEM-PA device; these files are also named OEMPA files. The binary format on the disk is the same as the configuration data structure in the computer memory. It is quicker to load.

The text format provides a better compatibility when the file format is updated. Here are some features about the text file:

- Some parameters are optional:
  - Parameters that could be saved in the default configuration file of your device instead of the OEMPA File.
    - Normally these types of parameters don’t really change per setup and it is more convenient to save them in a default configuration file. For example, often these parameters could be considered more device-dependent, versus setup-dependent.
    - If you save them as an OEMPA file and later want to change one of those parameters, you would have to go back and change every setup file.
  - The point count is a single cycle parameter.
    - By default the point count is not printed in the OEMPA file, so you can change the range easily and the point count will be updated automatically.
- You can select default options with the button “file option”:
  - “CustomFilter” defines options for default configuration file.
  - “Manager” defines options for setup configuration files.
- You can also change those options with the callback for each OEMPA file.

It is also possible to add a display correction section in an OEMPA file to store information required for a corrected view display. This section is optional. It is not possible (right now) to include a correction section in the binary format (so “OEMPASector”, described in *Software _Utilities.pdf*, will not work properly with a binary format).

3.2 Link with the driver

To function properly, the driver requests the customized API for three tasks:

- Load a configuration file.
- Creation of the default configuration file.
- Load the calibration configuration (version 1.1.5.0 and above).

The driver must call those functions of the higher level customized API, so three callbacks are registered by the customized API at initialization time (in the main function “DllMain” of the customized API). If you remove those registrations, or if you link your project only with the driver, a warning will be displayed and those 3 features of the driver will not function properly. It is better to link your application with the driver and the customized API; to avoid the warning, you can call the following function:

```
CHWDeviceOEMPA::DisableDisplayFatalErrorCustomizedAPI(true);
//call this function before any device connection.
```

3.3 Initialization

You can register the callback function for the customized API at the same time you register an acquisition callback function for the low level API (see Section 4, Low Level API (Driver)). This callback function (see “COEMPACustomizedExampleDlg::CallbackCustomizedAPI” found in the source code of “OEMPACustomizedExample”) is called each time an OEMPA file is loaded. The following is an example of callback registration (see the function “COEMPACustomizedExampleDlg::OnInitDialog”):

```
m_HWDeviceOEMPA.SetAcquisitionParameter(this);//registration of the user data
OEMPA_SetCallbackCustomizedDriverAPI(COEMPACustomizedExampleDlg::CallbackCustomizedAPI);
```

3.4 Callback

If you use the OEMPA file format, then the callback is useful to load/save OEMPA file.

3.4.1 Load Configuration File

The load function calls the callback function (“OEMPA_ReadFileWriteHW”) at different execution points:

- At the beginning of the load process.
    - Correction parameter is not an input parameter of the load function.
    - The callback function is used to get the correction table parameter.
- Just before updating the hardware device.
    - It is possible to browse the configuration data loaded in the computer memory.
    - It is also possible to modify it.
- Just after updating the hardware device.
    - It is possible to browse the configuration data that has been used to update the hardware device. This data could be different because the driver can adjust limits.

- At the exit of the load process.
- During connection with the device.

The callback function is also called at connection time with the device, when the default configuration file is loaded. Be careful with the configuration file:

- No cycles are defined: “iCycleCount” should be -1.
- No emission and reception are defined: all pointers are null.

3.4.2 Save Configuration File

The save functions call the callback functions (“OEMPA_WriteFileText”, “OEMPA_WriteFileBinary”) at different execution points:

- At the beginning of the save process.
    - You can change options used to save parameters in the text file.
    - You can change some parameters with your default values.
- At the exit of the save process.

Example: if you want to add the feature “1/n ascan” in the OEMPA saved file, you can refer to the following example in “OEMPATool”:

```
structCorrectionOEMPA* WINAPI CDlgHW::CallbackCustomizedAPI(void
*pAcquisitionParameter,const wchar_t *pFileName,enumStepCustomizedAPI eStep,structRoot
*pRoot,structCycle *pCycle,CFocalLaw *pEmission,CFocalLaw *pReception)
{
       int iChannelProbe,iChannelScan,iChannelCycle;
       CHWDeviceOEMPA *pOEMPA=(CHWDeviceOEMPA*)pAcquisitionParameter;
       bool bCscan=false;
       bool bSaveRequestAscanExample=false;//example - to save the "1/n ascan" feature in
OEMPA saved file.

       if(bSaveRequestAscanExample)
       {
              if(eStep==eWriteFile_Enter)
              {
                     pRoot->ullSavedParameters |= OEMPA_ASCAN_REQUEST;//to save the
feature "1/n ascan" in the OEMPA saved file.
                     pRoot->eAscanRequest = eAscanSampled;
                     pRoot->dAscanRequestFrequency = 100.0;
              }
       }
```

To enable the process you need to set “bSaveRequestAscanExample” to true.

3.5 API

The following is the API list:

Function name	Comment
OEMPA_WriteHW	Update the device with configuration data.
OEMPA_ReadHW	Read back configuration data from the device.
OEMPA_ReadFileText	Load configuration data from a text file.
OEMPA_ReadFileBinary	Load configuration data from a binary file.
OEMPA_WriteFileText	Save configuration data in a text file.
OEMPA_WriteFileBinary	Save configuration data in a binary file.
OEMPA_ReadFileWriteHW	Load configuration file and update the device from it.
OEMPA_ReadHWWriteFile	Read back the device to save its configuration in a file.
OEMPA_New / OEMPA_Free	To manage correction data allocation (*).
OEMPA_SetCallbackCustomizedDriverAPI	To register the callback function.

(*) correction data are useful data that are not sent to the device but required by the software to display corrected view.

In “OEMPACustomizedExample" there is only an example of loading an OEMPA file. You will find examples of how to save it in “OEMPAApplicationExample”. These applications are described in *Software_Examples.pdf*.

3.6 Project configuration

A special solution “UTCustomizedAPI.sln” is included with the SDK if you want to recompile the customized API to debug or to modify it:

“C:\ProgramData\AOS\OEMPA [version]\Sources\UTKernel\UTCustomizedAPI.sln”

To use the customized API in another project you will need to use the following headers (see “stdafx.h”):

```
#include"UTKernelDriver.h"//low level API used by the customized API
#include"UTKernelDriverOEMPA.h"//low level API used by the customized API
#include"CustomizedDriverAPI.h"//to use the driver customized API
```

The “CustomizedDriverAPI.h” should be included after MFC headers, if they are used.

Link your project with libraries: “UTKernelDriver.lib”, “UTKernelDriverOEMPA.lib” (link with “CustomizedDriverAPI.lib” is done by the header).

4. Low Level API (Driver)

4.1 Project configuration

There are two versions of the Low Level API: C++ and C#. There is an example application using each ("OEMPAAplicationExample" and "OEMPAFormExample") described in *Software_Examples.pdf*. The example code referred to comes from these examples. Some paragraphs refer to the utility "OEMPATool" which is described in *Software_Utilities.pdf*.

4.1.1 C++

Headers, libraries and compilation options are explained.

4.1.2 Headers

You need to include the following headers:

```
#include "Common.h"//not required (see below comments).

#include"UTKernelDriver.h"//required to use low level API
#include"UTKernelDriverOEMPA.h"// required to use low level API
#include"CustomizedDriverAPI.h"// required to use the driver customized API
#include"CustomizedWizardAPI.h"// required to use the wizard customized API
```

The "Common.h" header defines main options with which the kit has been compiled:

- Version letter (example 'a' for the kit "1.1.5.2a").
- The compilation option ("Debug" or "Release").
- Preprocessor definition of the symbol "_EVALUATION_":
    - If defined then the kit is an evaluation kit.
    - If not defined then this is a standard kit.

The "CustomizedDriverAPI.h" and "CustomizedWizardAPI.h" should be included after MFC headers if they are used.

In the case of standard kits, those different headers can be found in following folders:

"C:\ProgramData\AOS\OEMPA [version]\Sources\Common"
"C:\ProgramData\AOS\OEMPA [version]\Sources\UserHeaders\x64" for 64 bit
"C:\ProgramData\AOS\OEMPA [version]\Sources\UserHeaders\win32" for 32 bit

4.1.3 Libraries

You also have to link your application with following libraries:

- "UTKernelDriverOEMPA.lib"
- "UTKernelDriver.lib".

The link with “UTCustomizedDriverAPI.lib” and “UTCustomizedWizardAPI.lib” are defined in the headers.

4.1.4 Compilation options

You should compile in “Release” mode (you can find main options in the header “Common.h” for example the variable “KIT_CONFIGURATION”). “Win32” is used for 32-bit computers, and “x64” for 64-bit computers.

4.2 C# Driver

Use the reference “csDriverOEMPA” (the C# driver with main customized API features). This is an open source class library linked with “UTKernelDriverOEMPA.dll” and “UTKernelDriver.dll”.

You can also use the reference “csWizardTemplateLinear” as an example of how to use the wizard customized API.

4.3 Low Level API

The functions that are open to the user can be grouped as follows:

- General Functions:
  - Creation/deletion of the device.
  - Acquisition management: an acquisition callback function should be defined to get acquisition data.
- Communication Management:
  - Communication link selection (Ethernet, USB).
  - Connection with the HW device.
  - Status management (to be notified of disconnection, communication errors, etc.)
- Parameter Management:
  - Lock/Unlock the communication link.
  - Read/Write all FW UT parameters: the gain, the delays etc.:
    - Before processing the input FW parameter, any kind of “write” function checks the range of the input parameter for consistency.
    - Check functions can also be called directly.

NOTE: See the header “UTKernelDriverOEMPA.h”.

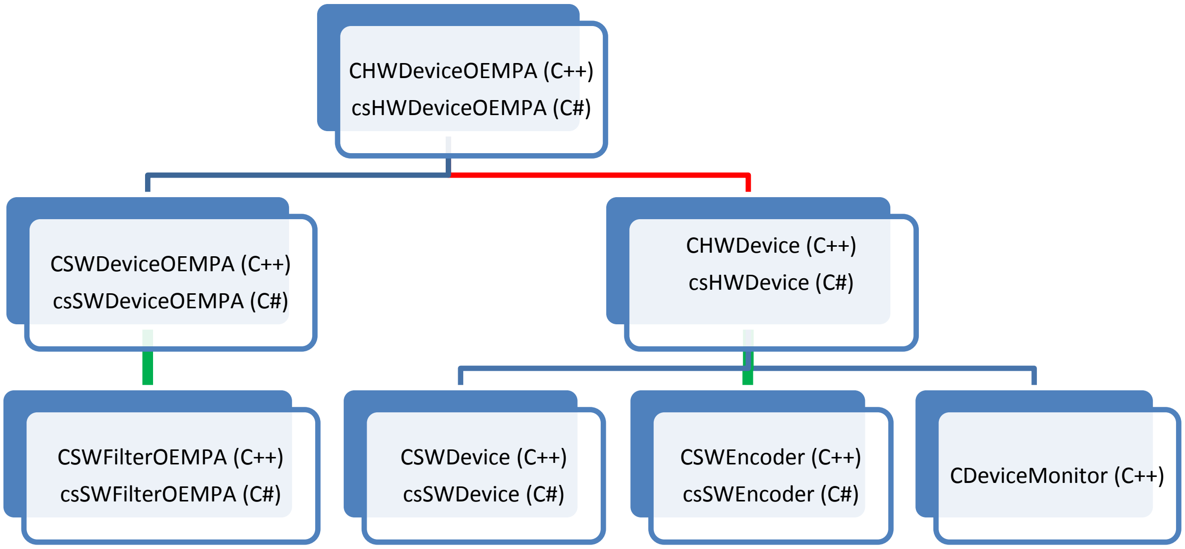
4.3.1 General

Most of parameters of the driver are stored in the device. Some of them are also stored in the computer memory. The device is managed by many different classes of two types:

- HW classes:

  - These classes are used to access directly the device (FW parameters).
  - Normally, you should stop the acquisition to access the device; if not, you can experience problems (it depends on what you want to do).
  - These classes have the prefix “CHW” for C++ and “csHW” for C# (example “CHWDeviceOEMPA”/”csHWDeviceOEMPA”).
- SW classes:
  - These classes are used to access parameters that are stored in the device and in the computer memory.
  - You can read these parameters without stopping the acquisition, and is quicker than accessing the device.
  - To perform special tasks for which no FW parameter is associated; for example to manage the connection state and the encoder resolution.
  - These classes have the prefix “CSW” for C++ and “csSW” for C# (example “CSWEncoder”/”csSWEncoder”).

Here is the flowchart (red line means inheritance, green lines means array):



- “CHWDevice” (C++) / “csHWDevice” (C#):
  - To lock/unlock the communication link.
  - To flush communication link (after locking it) in order to perform all pending communication operations.

    - To register/unregister acquisition callback function.
- "CSWDevice" (C++) / "csSWDevice" (C#)
    - To get the board name.
    - To manage main parameters of the communication link:
        - Type (Ethernet/USB).
        - Connection
    - To enable/disable pulser.
    - To get communication status:
        - Error count.
        - Stream count.
        - Data lost count (communication overflow).
- "CSWEncoder" (C++) / "csSWEncoder" (C#)
    - To enable/disable encoder.
    - To change the resolution, wire connections, encoder type.
- "CHWDeviceOEMPA" (C++) / "csHWDeviceOEMPA" (C#)
    - To read/write any FW parameter.
    - "LockDevice"/"UnlockDevice" should be used with these functions.
- "CSWDeviceOEMPA" (C++) / "csSWDeviceOEMPA" (C#)
    - To manage device-dependent communication link:
        - IP address
        - Port
    - To read HW type (serial number, system type, RX board count, IO board presence, main clock frequency, FW identifier).
    - To access main FW parameters that are cycle-independent (KeepAlive, AscanEnable, CscanTofEnable, AscanBitSize, trigger, etc.).
- "CSWFilterOEMPA" (C++) / "csSWFilterOEMPA" (C#): to manage device filters.
- "CDeviceMonitor" (C++): used to manage link with "EmuMon".

Generally function members of these classes return two different types of data:

- Boolean: true means no error and false means error.
- Integer: 0 means no error, otherwise it means error.

4.3.2 Instantiation

You can directly use the default constructor (see "OEMPAApplicationExampleDlg.h" for C++ and "OEMPAFormExample.cs" for C#):

`//C++`
`CHWDeviceOEMPA m_HWDeviceOEMPA;`
	`//C#`
	`public csHWDeviceOEMPA hwDeviceOEMPA;`

4.3.3 Initialization

Two steps:

- Registration of acquisition callback functions.
- Connection with the device.

You can register acquisition callback functions before managing the communication link - see the function "COEMPAApplicationExampleDlg::OnInitDialog" for C++ and "OEMPAFormExample.InitDialog" for C#:

```
//C++
m_HWDeviceOEMPA.SetAcquisitionParameter(this);//user parameter
m_HWDeviceOEMPA.SetAcquisitionAscan_0x00010103(COEMPAApplicationExampleDlg::CallbackAcqui
sitionAscan);
m_HWDeviceOEMPA.SetAcquisitionCscan_0x00010X02(COEMPAApplicationExampleDlg::CallbackAcqui
sitionCscan);
```

```
//C#
hwDeviceOEMPA.SetAcquisitionParameter(this);//user parameter
hwDevice.SetAcquisitionAscan(AcquisitionAscan_0x00010103);
hwDevice.SetAcquisitionCscan(AcquisitionCscan_0x00010X02);
```

To connect, two functions are called:

- One to set the IP address.
- The second to connect/disconnect.

Example: to connect (see "COEMPAApplicationExampleDlg::OnBnClickedCheckConnect" for C++ and "OEMPAFormExample.checkBoxConnect_CheckedChanged" for C#):

```
//C++
swprintf(pValue,40,L"%d.%d.%d.%d",m_cIP[0],m_cIP[1],m_cIP[2],m_cIP[3]);
if(!m_HWDeviceOEMPA.GetSWDeviceOEMPA()->SetIPAddress(pValue) ||
       !m_HWDeviceOEMPA.GetSWDevice()->SetConnectionState(eConnected,true))
{
       //connection ERROR
}else{
       //connection OK
}
```

```
//C#
bRet = swDeviceOEMPA.SetIPAddress(strIPAddress);
bRet = bRet && swDevice.SetConnectionState(csEnumConnectionState.csConnected,true);
if (!bRet)
       MessageBox.Show("Communication error!");//connection ERROR
```

Example: to disconnect (see “COEMPAApplicationExampleDlg::OnBnClickedCheckConnect” for C++ and “OEMPAFormExample.checkBoxConnect_CheckedChanged” for C#):

```
//C++
if(!m_HWDeviceOEMPA.GetSWDevice()->SetConnectionState(eDisconnected,true))
{
        //disconnection ERROR
}else{
        //disconnection OK
}

//C#
bRet = swDevice.SetConnectionState(csEnumConnectionState.csDisconnected, true);
if (!bRet)
        MessageBox.Show("Communication error!");//connection ERROR
```

4.3.4 Communication

Asynchronous communication will result in the highest performance (speed).

Before accessing FW parameters (“CHWDeviceOEMPA” for C++, “csHWDeviceOEMPA” for C#) with read/write functions, you must first lock the communication link with the function “CHWDevice::LockDevice” for C++ and “csHWDevice::LockDevice” for C#. ***Only one thread can access the communication link at the same time.***

4.3.4.1 Asynchronous

Read operations are not real time - it is not necessary to wait until the current operation (read or write) has been completed to request a new one in order to get the best communication performance (speed).

Read and write operations are packed together and sent to the device in the same stream. Then the device returns an answer for many read operations at the same time. Read and write operations are performed asynchronously.

Read functions use data pointers that are used asynchronously. “UnlockDevice” is used to complete all pending read (and write) operations. These data pointers are used before this function call, but you don’t know at which time they are used after the read operation call (and before the “UnlockDevice”).

4.3.4.2 Lock

All the following functions return “true” in the case of no errors, or “false” in the case of a communication error.

Function name	**Definition**
CHWDevice::LockDevice (C++) csHWDevice.LockDevice (C#)	To lock the communication link with the current thread.
CHWDevice::Flush (C++) csHWDevice.Flush (C#)	This function is useful for completing all pending communication operations, particularly read operations. This function can be called after “LockDevice” but before “UnlockDevice”.
CHWDevice::UnlockDevice (C++) csHWDevice.UnlockDevice (C#)	To release the communication link.

You can call any parameter function after the communication has been locked. **Afterwards, you must call** “UnlockDevice” to release the communication link. After returning from “UnlockDevice” all pending communication operations are completed (same feature as “Flush” that could be called between “LockDevice” and “UnlockDevice”).

The functions “LockDevice” and “UnlockDevice” have a special input parameter used to manage the pulser. If you want to access the device to perform read/write operations, it is better to disable the pulser; otherwise you can disturb the acquisition.

4.3.4.3 Lock C++

Here is the format of the input parameter to lock/unlock the communication link:

```
//C++
typedefenum enumAcquisitionState{   eAcqOff,    //pulse shot is disabled.
                                    eAcqDefault,//the  configuration  file  is
used to stop or start the pulse shot.
                                    eAcqOn};    //pulse shot is enabled.
```

One default input value is provided for the function (“eAcqDefault”) in order to automatically stop the pulser with “CHWDevice::LockDevice” and enable the pulser with “CHWDevice::UnlockDevice”.

Depending on the function called, here is the final value for the default input parameter (“eAcqDefault”):

Function	The final value with default input parameter “eAcqDefault” is stored in the software configuration file
“CHWDevice::**LockDevice**”	“eAcqOn” (to enable pulser) or “eAcqOff” (to disable pulser). To access it, use “CSWDevice::**GetLockDefaultDisablePulser**” and “CSWDevice::**SetLockDefaultDisablePulser**”.

"CHWDevice::**UnlockDevice**"	"eAcqOn" (to enable pulser) or "eAcqOff" (to disable pulser). To access it use "CSWDevice::**GetUnlockDefaultEnablePulser**" and "CSWDevice::**SetUnlockDefaultEnablePulser**".

Internal functions of **OEMPA customized API** use these default values, for example when you load an OEMPA file, the pulser is automatically disabled (via "CHWDevice::LockDevice") and automatically enabled (via "CHWDevice::UnlockDevice").

4.3.4.4 Lock C#

Here is the type of the input parameter to lock/unlock the communication link:

```
//C#
public enum class csEnumAcquisitionState{ csAcqOff=eAcqOff,//pulse   shot   is   disabled.
        csAcqDefault=eAcqDefault,//the configuration file is used to stop or start the pulse
shot.
        csAcqOn=eAcqOn        //pulse shot is enabled.
        };
```

For C# two types of functions are delivered, one with input parameter and another one without. This last function calls the first function with a default input parameter. For example:

```
bool csHWDevice::LockDevice()
{
        return m_pHWDevice->LockDevice(csAcqDefault);
}
```

The same rules apply for C#, you can read C++ from previous paragraph by replacing:

C++	C#
CHWDevice	csHWDevice
CSWDevice	csSWDevice
eAcqDefault	csAcqDefault
eAcqOn	csAcqOn
eAcqOff	csAcqOff

4.3.4.5 Pulser management

You can use "CSWDevice::EnablePulser" for C++ and "csSWDevice::EnablePulser" for C# to manage the pulser. It is also possible to use "CHWDevice::LockDevice" and "CHWDevice::UnlockDevice", in cases where you don't use the default value "eAcqDefault"/"csAcqDefault" (see "COEMPAApplicationExampleDlg::OnBnClickedCheckEnable"):

```
//C++
enumAcquisitionState eAcqState;
if(m_bCheckPulserEnable)
        eAcqState = eAcqOn;//enable pulser
else
        eAcqState = eAcqOff;//disable pulse
if(m_HWDeviceOEMPA.LockDevice())//here default value is used
{
        if(!m_HWDeviceOEMPA.UnlockDevice(eAcqState))//default value is not used
                bRet = false;//communication error
}else
        bRet = false;//communication error

//C#
csEnumAcquisitionState eAcqState;
if(m_bCheckPulserEnable)
        eAcqState = csAcqOn;//enable pulser
else
        eAcqState = csAcqOff;//disable pulse
if(hwDeviceOEMPA.LockDevice())//here default value is used
{
        if(!m_HWDeviceOEMPA.UnlockDevice(csAcqState))//default value is not used
                bRet = false;//communication error
}else
        bRet = false;//communication error
```

4.3.4.6 Read/write operations (C++)

"CHWDevice::LockDevice" and "CHWDevice::UnlockDevice" must be used to perform read/write operations with the device. You can use the default value (see "COEMPAApplicationExampleDlg::OnBnClickedButtonWrGain"):

```
//C++
bool bRet=true;
UpdateData();
if(m_HWDeviceOEMPA.LockDevice())//default input parameter
{
        if(!m_HWDeviceOEMPA.SetGainDigital(0,m_dEditWriteGain))
                bRet = false;//communication error
        if(!m_HWDeviceOEMPA.UnlockDevice())//default input parameter
                bRet = false;//communication error
}else
        bRet = false;//communication error
```

4.3.4.7 Read/write operations (C#)

“csHWDevice::LockDevice” and “csHWDevice::UnlockDevice” must be used to perform read/write operations with the device. You can use the default value (see “OEMPAFormExample::buttonWrite_Click”):

```
//C#
double dGain;
bool bRet = true;
dGain = Convert.ToDouble(textBoxWrite.Text);
if (hwDeviceOEMPA.LockDevice())
{
       if (!hwDeviceOEMPA.SetGainDigital(0, ref dGain))
              bRet = false;
       if (!hwDeviceOEMPA.UnlockDevice())
              bRet = false;
}
else
       bRet = false;
if (!bRet)
      MessageBox.Show("Communication error!");
```

4.3.5 Read/write operations

To read/write FW UT parameters, some functions are delivered; but before using them, you need to call “LockDevice” and “UnlockDevice”.

All functions are listed in the header “UTKernelDriverOEMPA” for C++ and “csDriverOEMPA.cpp” for C# between these two lines:

```
//<<PARAMETERS MANAGEMENT FUNCTIONS : BEGIN>>
…
//<<PARAMETERS MANAGEMENT FUNCTIONS : END>>
```

For simple FW parameters you have a single write function and a single read function, but for more complex FW parameters you have more than one function. The difference is in the type of input SW parameter used to reference the same FW UT parameter. See paragraph 4.3.5.7 “Input callback function parameter (C++)”.

A list of FW parameters and their corresponding functions are explained in the document “OEMParametersList”.

Example of how to read/write the gain is given in paragraphs 4.3.5.2 “Write operation (C++)” and Read operation (C++) “Read operation (C++)”.

```
//C++
bool CHWDeviceOEMPA::SetGainDigital(int iCycle,double &dGain);
bool CHWDeviceOEMPA::GetGainDigital(int iCycle,double *pdGain);

//C#
public bool SetGainDigital(int iCycle,ref double dGain);
public unsafe bool GetGainDigital(int iCycle,double *pdGain);
```

When you write a parameter, the value could be out of range. There is a special check function to check whether the value is in or out of range. This check function is called from the write function. Check functions can also be called directly from the class “CHWDeviceOEMPA” for C++ and “csHWDeviceOEMPA” for C#. For example “CHWDeviceOEMPA::SetGainDigital” calls internally “CHWDeviceOEMPA::CheckGainDigital”, the code is the following:

```
//C++
bool CHWDriverOEMPA::SetGainDigital(int iCycle,double&dGain)
{
bool bRet = CHWDriverOEMPA::CheckGainDigital(dGain);
if(!bRet)
      return false;
…//write the value to the HW.
}
```

4.3.5.1 Units

International System of Units is used for all inputs parameters, for example: for gain the unit is in decibels, and for delay the unit is in seconds… Be careful in the OEMPA text file, where the unit may be different. When this occurs, the units are specified in the file. For example:

[Cycle:0]
Gain=10.000000 dB   unit is in decibels
Start=1.000000 µs   (A-scanStart) unit is in microseconds (in the file only - in central memory the unit is in seconds)

```
//C++
bool CHWDriverOEMPA::SetGainDigital(int iCycle,double &dGain);//dGain unit dB
(USI).
bool CHWDriverOEMPA::SetAscanStart(int iCycle,double &dAscanStart);
//dAscanStart unit is second (USI).

//C#
public ref bool SetGainDigital(int iCycle,ref double dGain);//dGain unit dB
(USI).
public ref bool SetAscanStart(int iCycle,ref double dAscanStart);
//dAscanStart unit is second (USI).
```

If you want to use different units in your code or different types than the functions uses, please refer to paragraph 4.3.5.7 “Input callback function parameter (C++)”. This paragraph demonstrates how you can change units or types for write and read operations.

4.3.5.2 Write operation (C++)

Example of how to write the digital gain (see “COEMPAApplicationExampleDlg::OnBnClickedButtonWrGain”):

```
//C++
bool bRet=true;
UpdateData();
if(m_HWDeviceOEMPA.LockDevice())//default input parameter
{
        if(!m_HWDeviceOEMPA.SetGainDigital(0,m_dEditWriteGain))
                bRet = false;//communication error
        if(!m_HWDeviceOEMPA.UnlockDevice())//default input parameter
                bRet = false;//communication error
}else
        bRet = false;//communication error
```

Write operations are performed with functions that use referenced input/output parameters. These functions return “true” in case of no error, but for example, if your input parameter is out of range, then the function will return false and your input parameter value will be different.

4.3.5.3 Write operation (C#)

Example: to write the digital gain (see “buttonWrite_Click”):

```
//C#
double dGain;
bool bRet = true;
dGain = Convert.ToDouble(textBoxWrite.Text);
if (hwDeviceOEMPA.LockDevice())
{
        if (!hwDeviceOEMPA.SetGainDigital(0, ref dGain))
                bRet = false;
        if (!hwDeviceOEMPA.UnlockDevice())
                bRet = false;
}
else
        bRet = false;
if (!bRet)
        MessageBox.Show("Communication error!");
```

Write operations are performed with functions that use referenced input/output parameters. These functions return “true” in case of no error, but for example, if your input parameter is out of range, then the function will return false and your input parameter value will be different.

4.3.5.4 Check operation

It is also possible to check the range of any parameter without using the write operation. Here is an example of the digital gain parameter and the following code:

```
//C++
double dGain=12.2;
bRet = CHWDeviceOEMPA::CheckGainDigital(dGain);

//C#
double dGain=12.2;
bRet = CheckGainDigital(ref dGain);
```

Here are some call examples:

dGain (input value dB)	bRet (Boolean)	dGain (output value dB)	Comment
-40.0	False	0.0	Minimum 0.0 dB
0.22	True	0.2	Step 0.1 dB
90.0	False	80.0	Maximum 80.0 dB

Maximum, minimum and step are given in the document “OEMParametersList”.

4.3.5.5 Read operation (C++)

Remember that communication is asynchronous. You have to take into account a special rule to know when the returned output parameter value is available. A pointer is used to reference each read parameter. Just after returning from the read function, the pointer has not been used by the communication layer yet. The communication layer has finished its processing just after returning from “CHWDevice::UnlockDevice” or from “CHWDevice::Flush”.

Example (see “COEMPAApplicationExampleDlg::OnBnClickedButtonRdGain”):

```
//C++
m_dEditReadGain = -1.0;
if(m_HWDeviceOEMPA.LockDevice())
{
        if(!m_HWDeviceOEMPA.GetGainDigital(0,&m_dEditReadGain))
                bRet = false;//communication error
        if(!m_HWDeviceOEMPA.UnlockDevice())
                bRet = false;//communication error
}else
        bRet = false;//communication error
```

An example of improper use of the result of the gain read function:

```
//C++
double dGain,dAux;
int iCycle=0;
bool bRet=true;
if(m_HWDeviceOEMPA.LockDevice())
{
      bRet = m_HWDeviceOEMPA.GetGainDigital(iCycle,&dGain);
      dAux = dGain + 20.0; //ERROR the returned value is unsafe!!!!!
      if(!m_HWDeviceOEMPA.UnlockDevice())
            bRet =false;//communication error
      dAux = dGain + 20.0;//OK the returned value is safe here!!!!!
}else
      bRet =false;//communication error
```

Correct way to use the result of the gain read function (with flush function):

```
//C++
double dGain,dAux;
int iCycle=0;
bool bRet=true;
if(m_HWDeviceOEMPA.LockDevice())
{
      bRet = m_HWDeviceOEMPA.GetGainDigital(iCycle,&dGain);
      if(!m_HWDeviceOEMPA.Flush())
            bRet =false;//communication error
      dAux = dGain + 20.0;//the returned value is safe after “Flush”!!
      if(!m_HWDeviceOEMPA.UnlockDevice())
            bRet =false;//communication error
}else
      bRet =false;//communication error
```

4.3.5.6 Read operation (C#)

Remember that communication is asynchronous. You have to take into account a special rule to know when the returned output parameter value is available. A pointer is used to reference each read parameter. Just after returning from the read function, the pointer has not been used by the communication layer yet. The communication layer has finished its processing just after returning from “csHWDevice.UnlockDevice” or from “csHWDevice.Flush”.

In C#, read operation of the device is “unsafe”. Example (see “buttonRead_Click” for C#):

```
//C#
double dGain=-1.0;
```

```
int iCycleCount=-1;
bool bRet = true;
unsafe
{
        if (hwDeviceOEMPA.LockDevice())
        {
                if (!hwDeviceOEMPA.GetCycleCount(&iCycleCount))
                        bRet = false;
                if (!hwDeviceOEMPA.GetGainDigital(0, &dGain))
                        bRet = false;
                if (!hwDeviceOEMPA.UnlockDevice())
                            bRet = false;
        }
        else
                bRet = false;
}
if (!bRet)
        MessageBox.Show("Communication error!"); //error
```

An example of improper use of the result of the gain read function:

```
//C#
double dGain,dAux;
int iCycle=0;
bool bRet=true;
unsafe
{
        if(hwDeviceOEMPA.LockDevice())
        {
                bRet = hwDeviceOEMPA.GetGainDigital(iCycle,&dGain);
                dAux = dGain + 20.0; //ERROR the returned value is unsafe!!!!!
                if(!m_HWDeviceOEMPA.UnlockDevice())
                        bRet =false;//communication error
                dAux = dGain + 20.0;//OK the returned value is safe here!!!!!
        }else
                bRet =false;//communication error
}
```

Correct way to use the result of the gain read function (with flush function):

```
//C#
double dGain,dAux;
int iCycle=0;
bool bRet=true;
unsafe
{
        if(hwDeviceOEMPA.LockDevice())
        {
```

```
        bRet = m_HWDeviceOEMPA.GetGainDigital(iCycle,&dGain);
        if(!m_HWDeviceOEMPA.Flush())
            bRet =false;//communication error
        dAux = dGain + 20.0;//OK the returned value is safe after “Flush”!!
        if(!hwDeviceOEMPA.UnlockDevice())
            bRet =false;//communication error
    }else
        bRet =false;//communication error
}
```

4.3.5.7 Input callback function parameter (C++)

If the function of the driver doesn’t meet all your requirements: For example, if the input parameters type or unit is not what you need, then you need to use the additional function that uses the input callback function parameter. It is more difficult to use but it should meet all of your requirements.

This is the case for “csDriverOEMPA”, the C++/CLI stub to access the driver via C# interface. This stub uses callback functions to convert with C# data type.

For example - suppose that you want to store delays in memory with type “integer” and with unit nanosecond. The type of input parameter for read/write operation on emission delays (functions “CHWDeviceOEMPA::GetEmissionDelays”/“CHWDeviceOEMPA::SetEmissionDelays”) with the device is:

- Type is “float”.
- Unit is second.

So you have to convert the input parameter for both read/write operations. For write operation it is easy to change the variable unit just before calling the function. Here is an example:

```
//C++
//example to write delays with conversion unit
“CHWDeviceOEMPA::SetEmissionDelays”
bool SetEmissionDelay(int iCycle,int iDelay[16])//iDelay input delays in
nanosecond
{// “LockDevice” and “UnlockDevice” should be called outside this function.
DWORD dwHWAperture[g_iApertureDWordCount]={0x0000ffff,0};//aperture
definition : the first 16 elements of the probe are used.
float fDelay[16];//delays for 16 elements.
for(iElement=0;iElement<iApertureElementCount;iElement++)
{
    fDelay[iElement] = (float)(1.0e-9f*iDelay[iElement]);//format
conversion.
}
return HWDeviceOEMPA.SetEmissionDelay(iCycle,
                  adwHWAperture,pfDelay[iElement]);
}
```

It is possible to do the same with read functions, but in this case you need to call “CHWDevice::Flush” (see the rule explained in the previous paragraph):

```
//C++
//BAD example of how to read delays with conversion unit
"CHWDeviceOEMPA::GetEmissionDelays"
bool GetEmissionDelay(int iCycle,int iDelay[16])//iDelay input delays in
nanosecond
{// "LockDevice" and "UnlockDevice" should be called outside this function.
DWORD dwHWAperture[g_iApertureDWordCount]={0x0000ffff,0};//aperture
definition : the first 16 elements of the probe are used.
float fDelay[16];//delays for 16 elements.
if(!HWDeviceOEMPA.GetEmissionDelays(iCycle,
                        adwHWAperture,pfDelay[iElement]))
      return false;
if(!HWDeviceOEMPA.Flush())
      return false;
for(iElement=0;iElement<iApertureElementCount;iElement++)
{
iDelay[iElement] = (int)(1.0e9f*fDelay[iElement]);//BAD format conversion.
}
return true;
}
```

A single “CHWDevice::Flush” call is normal, however, multiple ‘Flush’ calls (for example many cycles to read delays, DAC etc.) will decrease the communication performance (the “Flush” function takes time processing). **To get the best communication performance, you should use special callback functions** that are automatically called when the HW reading operation has just completed.

These special callback parameters type have the suffix “structCallbackArray”, for example you can find:

- “structCallbackArrayFloat1D” to read/write emission delays and pulse widths.
- “structCallbackArrayFloat1D” to read/write reception element gains.
- “structCallbackArrayFloat2D” to read/write reception delays.
- “structCallbackArrayFloatDac” to read/write the DAC.

Example of high-performance read-back of emission delays (without calling “Flush” function):

```
//C++
bool WINAPI CallbackSetSize(struct structCallbackArrayFloat1D *pCallbackArray,intiSize)
```

```
{//set the size of the array
      if(iSize>16)
            return false;
      return true;
}
Bool WINAPI CallbackSetData(struct structCallbackArrayFloat1D
*pCallbackArray,intiIndex,floatfData)
{//set the size of the array
      int *piDelay=(int *)pCallbackArray->apParameter[0];
      if(iIndex>16)
            return false;
      piDelay[iIndex] = (int)(1.0e9f*fData);//RIGHT format conversion.
      return true;
}

//C++
//RIGHT example to read delay with unit conversion
bool GetEmissionDelay(int iCycle,int iDelay[16])//iDelay input delays in
nanosecond
{// "LockDevice" and "UnlockDevice" should be called outside this function.
structCallbackArrayFloat1D callbackArrayFloat1D;
callbackArrayFloat1D.pSetSize = CallbackSetSize;
callbackArrayFloat1D.pSetData = CallbackSetData;
pCallbackArray->apParameter[0] = (void*)&iDelay[0];//of course you have to be
sure that iDelay should still be allocated when the callback function will be
called (before "UnlockDevice" is called.
return HWDeviceOEMPA.GetEmissionDelays(iCycle,16, callbackArrayFloat1D);
}
```

4.3.5.8 Input callback examples (C++)

Examples of how to use input callback function parameters are shown in “OEMPATool”. You can also see the paragraph 4.4.5.2 “DAC” as an example in this document.

In “OEMPATool” click the button “Samples”. Then you can select one item in the samples list. By default none are selected.

```
[APISamples]
ReceptionFocusing=0
EmissionFocusing=0
EmissionWedgeDelay=0
ReceptionWedgeDelay=0
ReceptionBeam=0     #Gain, A-scanRange, GateMode.
DAC_standard=0      #set the DAC with two standard floating arrays.
DAC_callback=0      #set the DAC with callback functions and any variable of your choice.
ReceptionGate=0
```

You have to set the value 1 for one of those 8 items. Items “DAC_standard” and “DAC_callback” show how to set the same FW parameter with an array and with an input callback function. Refer to the file “APISamples.cpp”.

Here is the link between the item and the function:

Item	**Function name**
ReceptionFocusing	`APISamplesReceptionFocusing`
EmissionFocusing	`APISamplesEmissionFocusing`
EmissionWedgeDelay	`APISamplesEmissionWedgeDelay`
ReceptionWedgeDelay	`APISamplesReceptionWedgeDelay`
ReceptionBeam	`APISamplesReceptionBeam`
DAC_standard	`APISamplesDACStandard`
DAC_callback	`APISamplesDACCallback`
ReceptionGate	`APISamplesGate`

Each of those functions follows the same process:

- Lock the device
- Write the SW parameters.
- Read back the SW parameters.
- Unlock the device
- Check if there is any difference between written value and read back value
  - Notify the user of any discrepancy

4.3.5.9 Input callback function parameter (C#)

In the case of C#, only one data type is available. As explained “csDriverOEMPA”, the C++/CLI stub to access the driver via C# interface uses C++ callback functions to convert with C# data types. But from the C# interface, no input delegate are available.

4.3.6 Error reporting

Errors are reported in the dump file (for example on windows 7 “C:\ProgramData\AOS\OEMPA [version]\DumpTraceGlobal.txt”). Some errors are also displayed in the special window named “UTKernelError”. You can avoid the display of this window if you registered your own callback function to process those error notifications.

To add data in the dump file, you can use the function “UTKernel_Trace”. To add data in the “UTKernelError” window, you can use the function “UTKernel_SystemMessageBoxList”. To display the first message, “UTKernelError” is launched automatically; this is a heavy process for the computer.

Some popups are internally displayed by the driver with the 4 following functions that the user can also call:

Function (C++ / C# only for version 1.1.5.0)	Comment
UTKernel_SystemMessageBox / csDriverOEMPA.csMsgBox.SystemMessageBox	Blocking popup with single "OK" button.
UTKernel_SystemMessageBoxList / csMsgBox.SystemMessageBoxList	Non-blocking popup, no need for the user to press button "OK". A list of many calls could be displayed in the same popup.
UTKernel_SystemMessageBoxButtons / csMsgBox.SystemMessageBoxButtons	Blocking popup that requires specific answer from the user (YES, NO ...).
OempaApiMessageBox / csMsgBox.OempaApiMessageBox	Non-blocking popup called by the

If you want to avoid the display of these popups, you can overwrite their process by using a callback function. For each of these functions you can register one callback function:

Function (C++ / C# only for version 1.1.5.0)	Callback register function (C++ / C#)
UTKernel_SystemMessageBox / csDriverOEMPA.csMsgBox.SystemMessageBox	SetCallbackSystemMessageBox / csMsgBox.SetCallbackSystemMessageBox
UTKernel_SystemMessageBoxList / csMsgBox.SystemMessageBoxList	SetCallbackSystemMessageBoxList / csMsgBox.SetCallbackSystemMessageBoxList
UTKernel_SystemMessageBoxButtons / csMsgBox.SystemMessageBoxButtons	SetCallbackSystemMessageBoxButtons / csMsgBox.SetCallbackSystemMessageBoxButtons
OempaApiMessageBox / csMsgBox.OempaApiMessageBox	SetCallbackMessageBox / csMsgBox.SetCallbackOemmpaApiMessageBox

Examples of how to use these callbacks can be found in “OEMPATool” for C++ (menu “Edit/Kernel Message Box”) and “OEMPAFormExample” for C# (button “Msg Box”).

4.3.6.1 Acquisition error

Be careful - if the throughput is too high you can disturb the acquisition by adding data in the dump file or in the “UTKernelError” windows. This can add more processing burden on the computer, and is enough to add delay in the hardware socket; causing some acquisition data to be lost.

If the throughput is too high, the device will send a notification error. It is not possible to print this error in the dump file or to display it in the “UTKernelError” because it adds a little more heavy processing that can further disturb the device and cause more acquisition data loss (snowball effect). It is better to process this error by the application software itself (for example by displaying it) and then no more to avoid further acquisition loss.

4.3.7 HW Identify

Some functions can help you to identify your hardware equipment. The button “Status” of “OEMPATool” dialog can be used to print some of them in the status file. Most of these functions are members of the class “CSWDeviceOEMPA”:

Function “CSWDeviceOEMPA” (C++) / “csSWDeviceOEMPA” (C#)	Comment
“GetSerialNumber”	Return the serial number of your equipment.
“GetSystemType”	Return the system type as a string, for example “16:16”, “32:32”, etc…
“GetApertureCountMax”	Return the maximum element count of the aperture.
“GetElementCountMax”	Return the maximum element count of the probe connector.
“dGetClockPeriod” / “fGetClockPeriod”	Return the digitizing clock period.
“lGetClockFrequency”	Return the digitizing clock frequency.
“fFocussingDelayStepEmission”	Return the emission focusing delay step.
“fFocussingDelayStepReception”	Return the reception focusing delay step.
“GetFirmwareId”	Return the firmware identifier (com board).
“GetCycleCountMax”	Version 1.1.5.0: return the maximum cycle count of your HW. Version 1.1.4.1: supports only 2047 cycles.
“IsFullMatrixCapture”	Version 1.1.5.0: The FMC bit could be enabled or disabled. Version 1.1.4.1: FMC is not supported.
“IsFullMatrixCaptureReadWrite”	Version 1.1.5.0: the FMC bit could be Read/Write or Read only (for old FMC FW: FMC bit is not read only). Version 1.1.4.1: FMC is not supported.
“IsMatrixAvailable”	Version 1.1.5.0: is your equipment configured to use the matrix wizard? Version 1.1.4.1: matrix is not supported.
“IsLabviewAvailable”	Version 1.1.5.0: is your equipment configured to use the Labview stub?

	Version 1.1.4.1: LabView is not supported.
“GetTemperatureCount”	Return the RX board count and the maximum temperature sensor count.
“GetTemperatureSensorCount”	Return the temperature sensor count for each board.
“IsIOBoardEnabled”	Is an IO board integrated in your equipment?
“GetEmbeddedVersion”	Return the version of the embedded software.
“GetOptionsCom”	Return the speed of the Ethernet link.
“GetOptionsTCP”	Return TCP parameters.
“GetOptionsFlash”	Return identity of the flash.

4.4 Acquisition

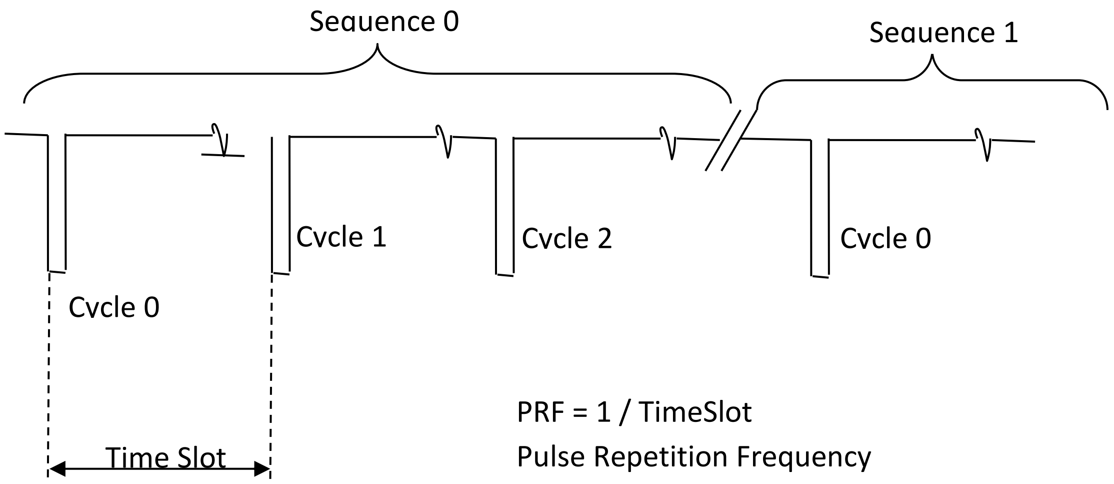
Before collecting acquisition data, you should set all required parameters to trig ultrasound pulser. Most of them are documented in this paragraph. For each FW parameter, the corresponding API is documented by:

- The C++ member function name of the class “CHWDeviceOEMPA” to write the FW parameter (“LockDevice” and “UnlockDevice” should be used).
- The C# member function name of the class “csHWDeviceOEMPA” to write the FW parameter (“LockDevice” and “UnlockDevice” should be used).
- The key name in the OEMPA file.
    - Even if you don’t use OEMPA files in your final SW application, this file format is useful for testing during the development process.

For classes “CHWDeviceOEMPA” and “csHWDeviceOEMPA”, the write function name has the prefix “Set”. In such cases, the read function name is the same except that the prefix is “Get”.

4.4.1 Terminology

Each cycle is defined by one emission pulse and reception. A set of cycles is a sequence.



In a sectorial scan, one cycle will correspond to an angle. In a linear scan, one cycle will correspond to an aperture.

4.4.2 Callback function

All A-scan, C-scan, and IO that are coming from the FW are delivered to a callback function written by the user. The IO stream is sent by the HW to the computer (as A-scan and C-scan streams). These IO streams inform the computer of the current encoder and digital inputs.

The trigger is mainly managed by two parameters: the trigger mode and the request IO mode. If you don't request an IO stream, the A-scan and C-scan acquisition callback functions can't receive encoder and digital inputs. The driver integrates a default process to link A-scans and C-scans with IO streams.

There are 4 acquisition callback functions:

- The A-scan acquisition callback that delivers A-scan with following inputs:
  - One user parameter that you can specify as you want (see "CHWDevice::SetAcquisitionParameter" for C++ and "csHWDevice.SetAcquisitionParameter" for C#).
  - General acquisition information
    - Encoder values
    - Digital inputs
    - Device identifier
    - Acquisition identifiers used to manage multiple channels

  - Use function “CHWDevice::SetAcquisitionAscan_0x00010103” for C++ and “csHWDevice.SetAcquisitionAscan_0x00010103” for C# to register your A-scan callback function.
- The C-scan acquisition callback that delivers C-scans with following inputs:
  - One user parameter that you can specify as you want.
  - General acquisition information
    - Encoder values
    - Digital inputs
    - Device identifier
    - Acquisition identifiers used to manage multiple channels.
  - Use function “CHWDevice::SetAcquisitionCscan_0x00010X02” for C++ and “csHWDevice.SetAcquisitionCscan_0x00010X02” for C# to register your C-scan callback function.
- The IO acquisition callback function that delivers IO streams.
  - Normally you don’t need to use it because the driver process already delivers encoder and digital inputs to A-scan and C-scan callback functions.
  - Use function “CHWDevice::SetAcquisitionIO_0x00010101” for C++ and “csHWDevice.SetAcquisitionIO_0x00010101” for C# to register your IO stream callback function.
- The reporting callback function “CHWDevice::SetAcquisitionInfo” for C++ and “csHWDevice.SetAcquisitionInfo” for C#
  - Instead of displaying HW errors with “UTKernelError” windows, you can choose to process HW errors by yourself.

Every time the Setup is updated, an identifier could be updated (the Setup identifier) so that the new acquisition data can be linked with this number. When acquisition data is delivered, you can determine from which settings they came from. This is an optional parameter - it is not required.

It is absolutely forbidden to access the device from callback functions (that is to call “LockDevice”/”UnlockDevice” and other read/write functions to the device).

4.4.3 Enable/disable

Before initiating acquisition (see paragraph 4.3.4.5 “Pulser management” for information on how to enable the pulser), you need to set some parameters: the cycle count, the A-scan enable, C-scan enable and IO stream enable, depending on your needs.

A-scan can be enabled/disabled. C-scan (amplitude) can be enabled/disabled via the HW gate. A special function is used to enable C-scan time of flight.

Feature	**Function C++ /** **Function C# /** **OEMPA file key**
Cycle count	CHWDeviceOEMPA::SetCycleCount / csHWDeviceOEMPA.SetCycleCount / “CycleCount” in section **“Root”**
A-scan	CHWDeviceOEMPA::EnableAscan / csHWDeviceOEMPA.EnableAscan / “AscanEnable” in section **“Root”**
HW C-scan gate (required for C-scan amplitude and time of flight)	CHWDeviceOEMPA::SetGateModeThreshold / csHWDeviceOEMPA.SetGateModeThreshold / “GateCount” in section **“Cycle:X”** and “Enable” in section **“Cycle:X\Gate:Y”**
C-scan time of flight (required only for the time of flight)	CHWDeviceOEMPA::EnableCscanTof / csHWDeviceOEMPA.EnableCscanTof / “EnableCscanTof” in section **“Root”**

If you want encoder data, you must enable the IO stream (encoder and digital inputs) - see the following paragraph.

4.4.4 Trigger

The **trigger mode** (FW parameter) specifies which signal is used to trig the ultrasound pulser. The **request IO** mode (FW parameter) specifies how the HW sends the IO stream (encoder and digital inputs) to the computer.

Three basic triggers are explained below for “OEMPAApplicationExample” (C++) dialog and “OEMPAFormExample” (C#). Here are the three different choices in the combo box “Trigger”:

Trigger	API “SetTriggerMode” (see array below)	API “SetRequestIO” (see array below)
Internal	The trigger mode is “Internal”: pulser is triggered by an internal timer. Another FW parameter, “TimeSlot”, is used to define this timer.	No IO streams are requested from the HW to the computer. A-scan and C-scans are delivered without position.
Encoder	The trigger mode is “Encoder”: pulser is triggered by a displacement of encoder, you must also specify the encoder acquisition step. If you don’t move the encoder, no pulses are generated and no A-	IO streams are required from the HW. They are sent on each cycle. It is also possible to send digital inputs on each cycle. A-scans and C-scans are delivered with position.

	scan is delivered to the callback function. The encoder is defined by many other FW parameters (wires, type, etc…).	
Mixed	The trigger mode is "Internal": pulser is triggered by an internal timer. Another FW parameter, "TimeSlot", is used to define this timer. If you don't move the encoder, A-scans are still delivered to the callback function.	IO streams are required from the HW. They are sent on each cycle. It is also possible to send digital inputs on each cycle. A-scans and C-scans are delivered with position.

API: "CHWDeviceOEMPA::**SetTriggerMode**(**enumOEMPATrigger** &eTrig)" (C++) /
"csHWDeviceOEMPA.**SetTriggerMode**(**csEnumOEMPATrigger** eTrig)" (C#) /
Section "**Root**" the key "**TriggerMode**" in OEMPA file

enumOEMPATrigger (C++) / csEnumOEMPATrigger (C#) / OEMPA file string	**Comment**
eOEMPAInternal / csOEMPAInternal / "Internal"	Internal mode. The cycles are generated from the PRF parameter and a new sequence starts after the last cycle. You need also to define the time slot (API "**SetTimeSlot**").
eOEMPAEncoder / eOEMPAEncoder / "Encoder"	Encoder mode. You must also define the acquisition step via the function "**SetTriggerEncoderStep**", and define your encoder (wires, type, resolution, etc.): see the next paragraph 4.4.4.1 "Encoders".
eOEMPAExternalCycle / eOEMPAExternalCycle / "ExternalCycle"	A new cycle is triggered by the digital input that you need to specify via "**SetExternalTriggerCycle**". The next cycle after the last cycle is for the next sequence.
eOEMPAExternalSequence / csOEMPAExternalSequence / "ExternalSequence"	A new sequence is triggered by the digital input that you need to specify via "**SetExternalTriggerSequence**". Inside a sequence, the cycle is generated via "**SetTimeSlot**".
eOEMPAExternalCycleSequence / csOEMPAExternalCycleSequence / "ExternalCycleSequence"	A new sequence is triggered by the digital input that you need to specify via "**SetExternalTriggerSequence**". Inside a sequence, a new cycle is triggered by the digital input that you need to specify via "**SetExternalTriggerCycle**".

API: "CHWDeviceOEMPA::**SetRequestIO**(**enumOEMPARequestIO** eRequest)" (C++) /
"csHWDeviceOEMPA.**SetRequestIO**(**csEnumOEMPARequestIO** eRequest)" (C#) /
Section "**Root**" the key "**RequestIO**" in OEMPA file

enumOEMPARequestIO (C++) / **csEnumOEMPARequestIO (C#) /** **OEMPA file string**	**Comment**
eOEMPANotRequired / csOEMPANotRequired / "NotRequired"	No IO stream is sent by the HW to the computer.
eOEMPAOnCycleOnly / csOEMPAOnCycleOnly / "OnCycleOnly"	IO streams are sent on each cycle.
eOEMPAOnSequenceOnly / csOEMPAOnSequenceOnly / "OnSequenceOnly"	IO streams are sent on each sequence.
eOEMPAOnDigitalInputOnly / csOEMPAOnDigitalInputOnly / "OnDigitalInputOnly"	IO streams are sent on any modification of digital inputs. You need to specify the mask via "**SetRequestIODigitalInputMask**".
eOEMPAOnDigitalInputAndCycle / csOEMPAOnDigitalInputAndCycle / "OnDigitalInputAndCycle"	IO streams are sent on each cycle and each time a digital input has been updated. You need to specify the digital input mask via "**SetRequestIODigitalInputMask**".
eOEMPAOnDigitalInputAndSequence / csOEMPAOnDigitalInputAndSequence / "OnDigitalInputAndSequence"	IO streams are sent on each sequence and each time a digital input has been updated. You need to specify the digital input mask via "**SetRequestIODigitalInputMask**".

The following is the API to set the digital input mask to send IO stream:

API: "CHWDeviceOEMPA::**SetRequestIODigitalInputMask**" (C++) /
"csHWDeviceOEMPA.**SetRequestIODigitalInputMask**" (C#) /
Section "**Root**" the keys "**RequestIODigitalInputMaskRising**" and
"**RequestIODigitalInputMaskFalling**" in OEMPA file

This is the API to set the external trigger signal used for the cycle:

API: "CHWDeviceOEMPA::**SetExternalTriggerCycle**" (C++) /
"csHWDeviceOEMPA.**SetExternalTriggerCycle**" (C#) /
Section "**Root**" the keys "**ExternalTriggerCycle**" in OEMPA file

This is the API to set the external trigger signal used for the sequence:

API: "CHWDeviceOEMPA::**SetExternalTriggerSequence**" (C++) /
"csHWDeviceOEMPA.**SetExternalTriggerSequence**" (C#) /
Section "**Root**" the keys "**ExternalTriggerSequence**" in OEMPA file

4.4.4.1 Encoders

Two encoders (X and Y) are available. You will need to specify them, and several parameters are required:

Feature	**API C++ /** **API C# /** **OEMPA file key**	**comment**
Type	CHWDeviceOEMPA::SetEncoderType / csHWDeviceOEMPA.SetEncoderType / "Encoder1Type" in section "Root" for X and "Encoder2Type" in section "Root" for Y	Different types are available, see below.
Wire A	CHWDeviceOEMPA::SetEncoderWire1 / csHWDeviceOEMPA.SetEncoderWire1 / "Encoder1A" in section "Root" for X and "Encoder2A" in section "Root" for Y	"iEncoderIndex" is 0 for X and 1 for Y. You need to specify one digital input. "DigitalInputOff" can be used to free the encoder.
Wire B	CHWDeviceOEMPA::SetEncoderWire2 / csHWDeviceOEMPA.SetEncoderWire2 / "Encoder1B" in section "Root" for X and "Encoder2B" in section "Root" for Y	"iEncoderIndex" is 0 for X and 1 for Y. You need to specify one digital input.

Resolution (optional)	CSWDevice::lSetResolution /
	csSWDevice.lSetResolution /
	for X "SWEncoder1Resolution" and "SWEncoder1Divider" in section "Root",
	for Y "SWEncoder2Resolution" and "SWEncoder2Divider" in section "Root"
		Resolution is defined by two numbers: the resolution and the divider. Generally the divider is equal to 1. You can access to the encoder class via "CHWDevice::GetSWDevice" and "CSWDevice::GetSWEncoder" ("csHWDevice.GetSWDevice" and "csSWDevice.GetSWEncoder")

The resolution is optional because you can get raw encoder values instead of decoded value (the resolution is taken into account):

EncoderDecodedValue = EncoderRawValue * Divider / Resolution

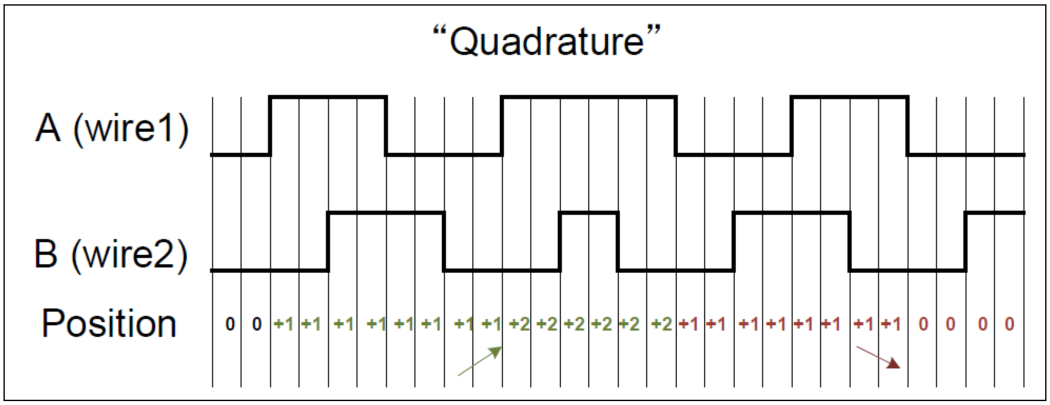
If the trigger mode is “Internal” then you don’t need to specify the resolution. If the trigger mode is “Encoder” then you can specify the encoder acquisition step with function “SetTriggerEncoderStep”. Two functions are available, one of them allows you to specify the acquisition step as a distance; but in this case you need to specify the encoder resolution beforehand. The two functions are defined as the following:

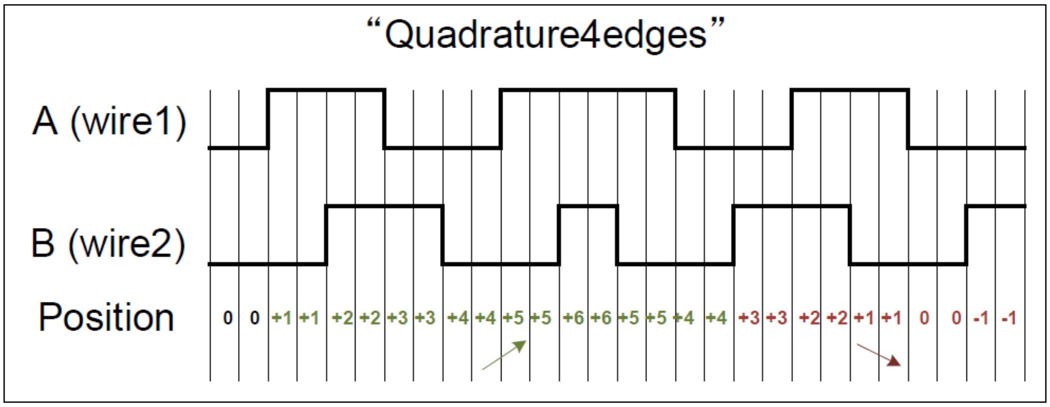
API 1: Input parameter is an integer, this is the step of the raw encoder to trig the pulse.
“CHWDeviceOEMPA::**SetTriggerEncoderStep**(DWORD EncoderRawStep)” (C++) /
“csHWDeviceOEMPA.**SetTriggerEncoderStep**(DWORD EncoderRawStep)” (C#) /
In OEMPA file there is no integer data.
API 2: Input parameter is a float value.
This is the distance in meter to trig the pulse, the formula used to get the RawEncoderStep is the following:
EncoderRawStep = round(dStep*1.0e3*Resolution/Divider);
“CHWDeviceOEMPA::**SetTriggerEncoderStep**(double dStep)” (C++) /
“csHWDeviceOEMPA.**SetTriggerEncoderStep**(double dStep)” (C#) /
Section “**Root**” the key “**TriggerEncoderStep**” in OEMPA file
See the example in paragraph 4.4.4.4 “IO Connector”.

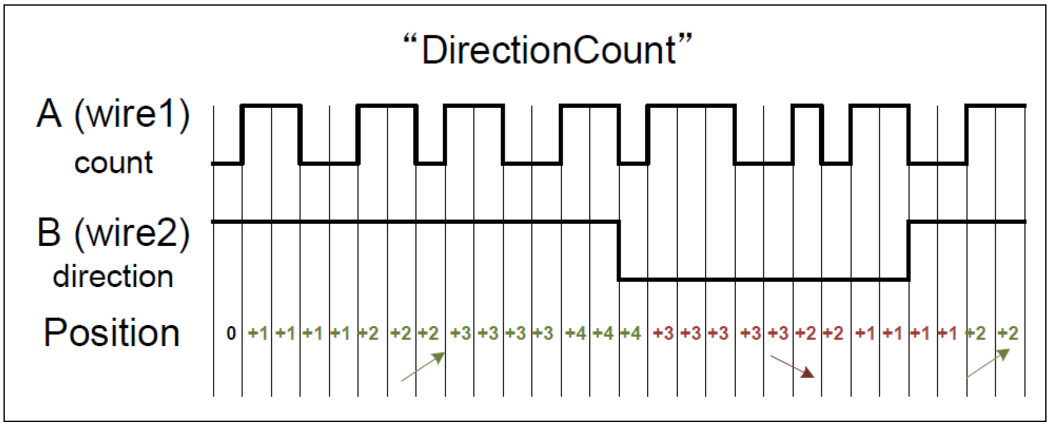
Encoder types are the following (detailed sketches are given after the following table):

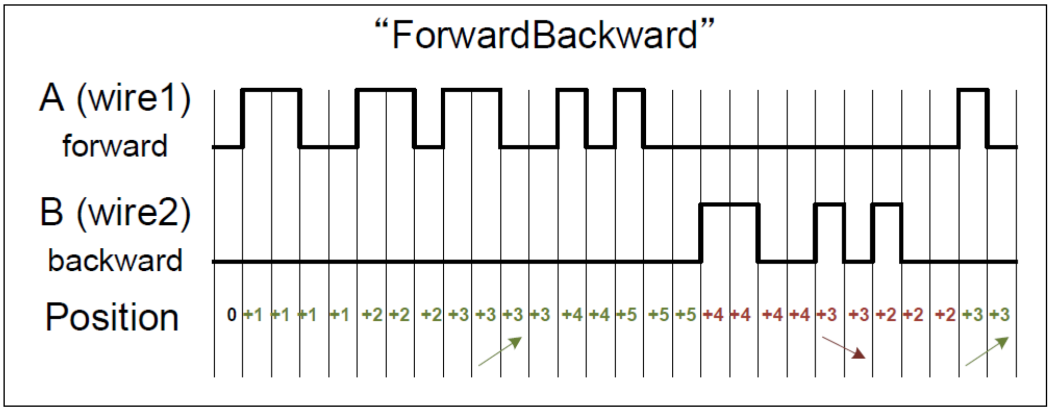
enumEncoderType (C++) / csEnumEncoderType (C#) / OEMPA file string	**Comment**
eStaticScan / csStaticScan / “StaticScan”	Useful to disable the encoder. If the encoder is disabled the corresponding digital inputs are free for other general purposes.
eEncoderQuadrature / csEncoderQuadrature / “Quadrature”	This mode is commonly used in incremental rotary encoders. The position value is incremented or decremented depending on the phase shift between A and B. If A is ahead of B: incrementing, if B is ahead of A: decrementing
eEncoderQuadrature4Edges / csEncoderQuadrature4Edges / “Quadrature4Edges”	It is the same as the mode "Quadrature", but with 4 times the resolution.

eEncoderDirectionCount / csEncoderDirectionCount / "DirectionCount"	In this mode, the encoder steps are on the phase A and the phase B gives the direction. When B is high: incrementing, when B is low: decrementing. The value is incremented or decremented when there is a rising edge on A.
eEncoderForwardBackward / csEncoderForwardBackward / "ForwardBackward"	In this mode, phase A is used to increase the encoder value, and B is used to decrease it.









You can also adjust the FW debouncer with the following API:

API: “CHWDeviceOEMPA::**SetEncoderDebouncer**(double dTime)” (C++) /
“csHWDeviceOEMPA.**SetEncoderDebouncer**(double dTime)” (C#) /
Section “**Root**” the key “**DebouncerEncoder**” in OEMPA file

Once the setting is completed, you can get encoder using many different functions:

Get Encoder	**Values**	**Function (C++) /** **Function (C#)**
Last encoder (Lock/Unlock not required)	raw	CSWEncoder::GetInspectionHWValue / csSWEncoder::GetInspectionHWValue
	decoded	CSWEncoder::GetInspectionSWValue / csSWEncoder::GetInspectionSWValue
Current encoder	raw and decoded	Encoders are delivered automatically as input of the A-scan and C-scan acquisition callback function.

From IO stream	raw and decoded	Register an acquisition callback function for IO stream: CHWDevice::SetAcquisitionIO_0x00010101 / csHWDevice.SetAcquisitionIO_0x00010101

See the example in paragraph 4.4.4.4 “IO Connector”.

Once an encoder has been enabled, some digital inputs are reserved to manage it, and those inputs cannot be reused as general purpose digital inputs. If you want to reuse them as general purpose digital inputs you need to free the encoder. Here is an example of this (OEMPA file format):

OEMPA file to enable encoder (some digital inputs are reserved)	OEMPA file to disable encoder (digital inputs 1 to 4 can be reused as general purpose digital inputs)

[Root]
CycleCount=-1
TriggerMode=Encoder
RequestIO=OnCycleOnly
TriggerEncoderStep=1.000000 mm
Encoder1Type=Quadrature
Encoder1A=DigitalInput01
Encoder1B=DigitalInput02
Encoder2Type=Quadrature
Encoder2A=DigitalInput03
Encoder2B=DigitalInput04
SWEncoder1Resolution=4
SWEncoder1Divider=1
SWEncoder2Resolution=4
SWEncoder2Divider=2
	[Root]
	CycleCount=-1
	TriggerMode=Internal
	Encoder1Type=StaticScan
	Encoder2Type=StaticScan
	Encoder1A=DigitalInputOff
	Encoder1B=DigitalInputOff
	Encoder2A=DigitalInputOff
	Encoder2B=DigitalInputOff

4.4.4.2 Digital inputs

Some digital inputs are used for:

- Encoder wires.
- External signals to trig pulser.
- General-purpose inputs.

A digital input can be used only once. For example, if a digital input is used by an encoder, this input is not available as a general-purpose input or external trigger.

Feature	**Comment**
Encoder wires	See the above paragraph 4.4.4.1 “Encoders”.
Cycle trigger	See the above paragraph 4.4.4 “Trigger”.
Sequence trigger	See the above paragraph 4.4.4 “Trigger”.
General purpose inputs	IO stream are requested (see the above paragraph 4.4.4 “Trigger”) and the value “DigitalInput” should be used (for example “OnDigitalInputOnly”, “OnDigitalInputAndCycle”). You need also to define the digital input mask on which IO streams are sent.

You can also adjust the FW debouncer with the following API:

API: “CHWDeviceOEMPA::**SetDigitalDebouncer**(double dTime)” (C++) /
“csHWDeviceOEMPA.**SetDigitalDebouncer**(double dTime)” (C#) /
Section “**Root**” the key “**DebouncerDigital**” in OEMPA file

Once the setting is complete, many functions are available to get the digital inputs:

Get Digital Inputs	**Lock/Unlock**	**Function (C++) / Function (C#)**
Last values	not required	CSWDeviceOEMPA::GetDigitalInput / csSWDeviceOEMPA::GetDigitalInput
Request current HW values	required	CHWDeviceOEMPA::GetDigitalInput/ csHWDeviceOEMPA::GetDigitalInput

See the example in paragraph 4.4.4.4 “IO Connector”.

4.4.4.3 Digital outputs

Six digital outputs are available. You can manage them with the following API:

“CHWDeviceOEMPA::**SetDigitalOutput**(int **X**,**enumOEMPAMappingDigitalOutput out**)” (C++) /
“csHWDeviceOEMPA.**SetDigitalOutput**(int **X**,**csEnumOEMPAMappingDigitalOutput out**)” (C#) /
Section “**Root**” the key “**DigitalOutputX**” in OEMPA file.
X : integer range 0 to 5

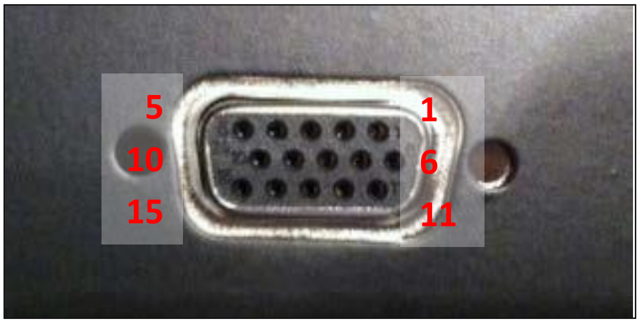
enumOEMPAMappingDigitalOutput (C++) / enumOEMPAMappingDigitalOutput (C#) / OEMPA file string	**Comment**
eOEMPALow / csOEMPALow /	Low Level (GND).

"Low"	
eOEMPAHigh / csOEMPAHigh / "High"	High Level.
eOEMPASignalCycle / csOEMPASignalCycle / "SignalCycle"	The corresponding pin rises and stays high for 6us at the beginning of each cycle.
eOEMPASignalSequence / csOEMPASignalSequence / "SignalSequence"	The corresponding pin rises and stays high for 6us at the beginning of each sequence (i.e. at the beginning of the cycle 0).

See the example in paragraph 4.4.4.4 "IO Connector".

4.4.4.4 IO Connector

Here is the I/O connector on the back panel of the OEMPA system:



D-sub 15 - female connector. The following table shows the pin assignment of this connector:

Pin #	Function	Comment
1	Digital Output 01	"Open-Collector" output with an internal 10 KΩ pull-up to 5V.
2	Digital Output 02	"Open-Collector" output with an internal 10 KΩ pull-up to 5V.
3	+5V, Output	
4	Digital Output 03	"Open-Collector" output with an internal 10 KΩ pull-up to 5V.
5	Digital Output 04	"Open-Collector" output with an internal 10 KΩ pull-up to 5V.
6	Digital Output 05	"Open-Collector" output with an internal 10 KΩ pull-up to 5V.
7	Digital Output 06	"Open-Collector" output with an internal 10 KΩ pull-up to 5V.
8	Digital Input 01	Input with an internal 1 KΩ pull-up to 5V.
9	Digital Input 02	Input with an internal 1 KΩ pull-up to 5V.
10	Digital Input 03	Input with an internal 1 KΩ pull-up to 5V.
11	Digital Input 04	Input with an internal 1 KΩ pull-up to 5V.
12	Digital Input 05	Input with an internal 1 KΩ pull-up to 5V.
13	N/C	

14	Digital Input 06	Input with an internal 1 KΩ pull-up to 5V.
15	**ISOLATOR GROUND**	NOT same GND as case

Example:

- Beginning of the [Root] section of the OEMPA file

OEMPA file	Comment

SWEncoderResolution=851
SWEncoderDivider=100
Encoder1A=DigitalInput01
Encoder1B=DigitalInput02
RequestIODigitalInputMaskRising=60
RequestIODigitalInputMaskFalling=60
DigitalOutput0=High
DigitalOutput1=Low
	Encoder Wire A = digital input 01 (Pin #8)
	Encoder Wire A = digital input 02 (Pin #9)
	All inputs except encoder wires (32+16+8+4)
	All inputs except encoder wires
	Digital output 01 Pin #1 high
	Digital output 02 Pin #2 low

For the next part of the [Root] section, different cases are used: RequestIO=OnDigitalInput	If the digital input 05 (Pin #12) is set from low to high, the IO stream is sent and the IO callback function is called with the following input: pIOHeader->inputs=0x0000001c pIOHeader->edges=0x00000010

RequestIO=OnDigitalInputAndCycle
TriggerMode=Encoder
TriggerEncoderStep=1.0
	Emitter pulses are triggered every 9 encoder step:
	round(1.0*851/100)=9
	9, 18, 27, 36 …

RequestIO=OnDigitalInputAndCycle
TriggerMode=Internal
	A-scan and C-scan callback function return the following decoded value:
	EncoderDecodedValue=EncoderRawValue*100/851
	In this formula data are non integer, this is the difference with the “Encoder” mode.

4.4.4.5 Temperature

Temperature probes are positioned throughout the device. If the ventilation airflow is insufficient, then the temperature will increase. If the temperature is too high, you must stop the device in order prevent damage to the components (see the *Hardware_Introduction.pdf* document for more information). You can read the temperatures by requesting them explicitly during setting, or automatically during acquisition:

API temperature request (setting time)	“CHWDeviceOEMPA::GetTemperatureSensor” (C++) / “csHWDeviceOEMPA.GetTemperatureSensor” (C#)
API temperature request (during acquisition)	“structAcqInfoEx::lMaxTemperature” (C++) / “csStructAcqInfoEx.lMaxTemperature” (C#) “CSubStreamIO_0x0101::maxTemperature” (C++) / “csSubStreamIO_0x0101.maxTemperature” (C#)

The parameter “structAcqInfoEx::lMaxTemperature” returns the maximum temperature as part of the A-scan or C-scan stream and the parameter “CSubStreamIO_0x0101::maxTemperature” returns the maximum temperature as part of the IO Stream. The value may be read within the corresponding callback function. See paragraph 4.4.2, Callback function, for more information. The details of the data streams can be found in the sections following this one.

The following functions are also useful for reading temperature:

“CSWDeviceOEMPA::GetTemperatureCount” (C++) / “csSWDeviceOEMPA.GetTemperatureCount” (C#)	Return the board count and the maximum sensor count per board.
“CSWDeviceOEMPA::GetTemperatureSensorCount” (C++) / “csSWDeviceOEMPA.GetTemperatureSensorCount” (C#)	Return the sensor count for each board.

You can find an example of querying temperature in “OEMPATool” (button “Status”) and in “OEMPAFormExample” (button “Status”, only version 1.1.5.0).

4.4.5 A-scan

The stream format coming from FW is not openly documented, but that being said you only need to understand the acquisition callback function. To get A-scan data, you must register a callback function that delivers all A-scans sent by the device. The callback function prototype is named “pTypeAcquisitionAscan_0x00010103” (defined in “UTBasicType.h” for C++) and “TypeAcquisitionAscan_0x00010103” (delegate defined in “csDriverOEMPA” for C#):

```
//C++
typedef UINT (WINAPI *pTypeAcquisitionAscan_0x00010103)(void
*pAcquisitionParameter,structAcqInfoEx &acqInfo,const CStream_0x0001 *pStreamHeader,const
CSubStreamAscan_0x0103 *pAscanHeader,const void *pBufferMax,const void *pBufferMin,const
void *pBufferSat);
```

```
//C#
public delegate int TypeAcquisitionAscan_0x00010103(Object pAcquisitionParameter, ref
csAcqInfoEx acqInfo, ref csHeaderStream_0x0001 pStreamHeader, ref csSubStreamAscan_0x0103
pAscanHeader, void* pBufferMax, void* pBufferMin, void* pBufferSat);
```

Here are the relevant input parameters:

- "pAcquisitionParameter": first common input, this is a user defined input parameter.
- "acqInfo": second common input, general information such as encoder, digital inputs, maximum temperature reading and multiple channel indexes.
- "pStreamHeader": third common input, frame header with the setting identifier.
- "pAscanHeader": A-scan header ("CSubStreamAscan_0x0103" for C++, "csSubStreamAscan_0x0103" for C#).
- "pBufferMax": array of points.
- "pBufferMin": array of points. Not useful if there is no compression. If you have compression, then minimum A-scan could be different than maximum. See next paragraph for more explanation.
- "pBufferSat": array of points. Can be used to check saturation of your focal law. See next paragraph for more explanation.

The first three inputs are explained in paragraph 4.4.7 "Callback common inputs".

Here are some of the members of "**CSubStreamAscan_0x0103**" C++ / "**csSubStreamAscan_0x0103**" C#:

Member	**Comment**
"version"	This value is 1 for amplitude result only and 2 for amplitude and time of flight results.
"seqLow" "seqHigh"	Sequence number (LSB and MSB).
"cycle"	Cycle number.
"dataCount"	Point count.
"dataSize"	Byte count of one data (example for 8 bits dataSize=1).
"IFNotInitialized"	This information indicates whether the gate result used to synchronize the A-scan has been valid for the first time. This information is useful only if echo tracking is used to synchronize the A-scan.
"IFOldReference"	This information indicates whether the gate result used to synchronize the A-scan is an old value, or if it has been refreshed (new detection). This information is useful only if echo tracking is used to synchronize the A-scan.
"dacIFNotInitialized"	This information lets you know if the gate result used to synchronize the DAC has been valid for the first time. This information is useful only if echo tracking is used to synchronize the DAC.

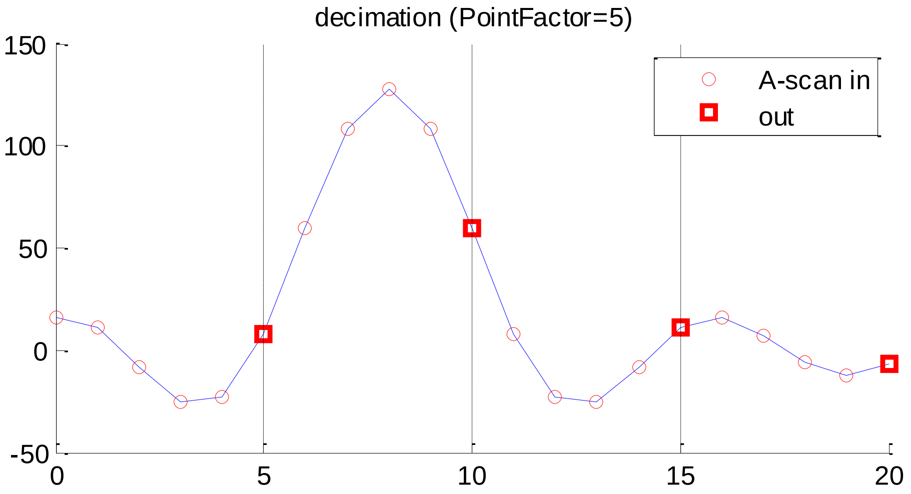
"dacIFOldReference"	This information lets you know if the gate result used to synchronize the DAC is an old value or if it has been refreshed (new detection). This information is useful only if echo tracking is used to synchronize the DAC.

See the paragraph 4.4.7.3 "Acquisition channel" for comments about acquisition channel members.

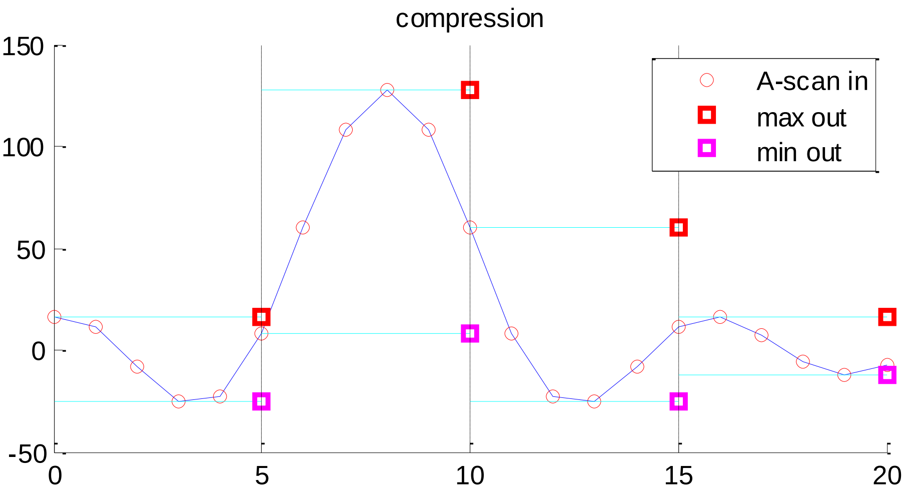
4.4.5.1 FW parameters

The following features are available:

- "Easy" parameters: bit size, gain (analog, digital), rectification, time slot, start. They are documented in the current paragraph.
- Filter: see paragraph 4.4.5.3 "Filters".
- DAC: see paragraph 4.4.5.2 "DAC".
- Range with or without compression/decimation: instead of using the high-frequency digitizer, you can decrease the sample count. There are two modes:
  - Decimation: the digitizing frequency is smaller and only the first A-scan point over N points is sent to the computer. The signal frequency structure is not modified. Maximum and minimum of the signal could be lost.



  - Compression: the maximum/minimum for N A-scan points are sent to the computer. The signal frequency structure is modified.



  - You can choose to send either maximum buffer, minimum buffer, or both.
- For the same specified A-scan range, two functions are available:
  - Point count as an input: the digitizing factor (also named point factor) linked to the digitizing frequency is an output.
  - Point factor as an input: a factor linked with the digitizing frequency is an input and the point count an output.

- “1/n A-scan”: this feature is useful when you don’t need to store all A-scan, you need only A-scan for the display. In this case instead of requesting all A-scans, it is better to request only 1 A-scan for n pulses. This is to decrease the acquisition throughput to avoid data loss.
- Echo tracking: it is possible to synchronize the A-scan gate on the detection of a HW C-scan gate (echo tracking). See paragraph 4.4.8.4 “Echo Tracking API” for more information.

Feature	API C++ / API C# / OEMPA file key	comment
Bit size	CHWDeviceOEMPA::SetAscanBitSize / csHWDeviceOEMPA.SetAscanBitSize / “AscanBitSize” in section “Root”	Different values are available, see below.
Gain (digital and analog)	CHWDeviceOEMPA::SetGainDigital / csHWDeviceOEMPA.SetGainDigital / “GainDigital” in section “Cycle:X”	To get the best signal/noise ratio, increase the analog gain first and then the digital gain. Beam correction is
	CHWDeviceOEMPA::SetGainAnalog / csHWDeviceOEMPA.SetGainAnalog /	

	“GainAnalog” in section “Cycle:X”	the same as digital gain. Use beam correction only for calibration, the digital gain should be the same for all cycles of the same channel.
	CHWDeviceOEMPA::SetBeamCorrection / csHWDeviceOEMPA.SetBeamCorrection / “BeamCorrection” in section “Cycle:X”	
Rectification	CHWDeviceOEMPA::SetAscanRectification / csHWDeviceOEMPA.SetAscanRectification / “Rectification” in section “Cycle:X”	Different values are available, see below.
Time slot	CHWDeviceOEMPA::SetTimeSlot / csHWDeviceOEMPA.SetTimeSlot / “TimeSlot” in section “Cycle:X”	
Start	CHWDeviceOEMPA::SetAscanStart / csHWDeviceOEMPA.SetAscanStart / “Start” in section “Cycle:X”	International unit (s). µs used in OEMPA file.
Range (default point count)	CHWDeviceOEMPA::SetAscanRange / csHWDeviceOEMPA.SetAscanRange / “Range” in section “Cycle:X”	The maximum digitizing frequency is used.
Range, point count is an input.	CHWDeviceOEMPA::SetAscanRangeWithCount / csHWDeviceOEMPA.SetAscanRangeWithCount / “Range”, “Compression” and “PointCount” in section “Cycle:X”	The formula is given below. The point factor is an output.
Range, point factor is an input.	CHWDeviceOEMPA::SetAscanRangeWithFactor / csHWDeviceOEMPA.SetAscanRangeWithFactor / “Range”, “Compression” and “PointFactor” in section “Cycle:X”	The formula is given below. The point count is an output.
1/n A-scan	CHWDeviceOEMPA::SetAscanRequest / csHWDeviceOEMPA.SetAscanRequest / “AscanRequest” in section “Root”	Two parameters: the mode (different values are available, see below) and the frequency.
	CHWDeviceOEMPA::SetAscanRequestFrequency / csHWDeviceOEMPA.SetAscanRequestFrequency / “AscanRequestFrequency” in section “Root”	

You can select each A-scan buffer you want to receive in the acquisition callback function:

Buffer	**Function C++ / function (C#) / OEMPA file**	**Comment**
Maximum (pBufferMax)	CHWDeviceOEMPA::EnableAscanMaximum/ csHWDeviceOEMPA.EnableAscanMaximum/	Useful in case of compression and decimation.

	“Maximum” in section “Cycle:X”	
Minimum (pBufferMin)	CHWDeviceOEMPA::EnableAscanMinimum/ csHWDeviceOEMPA.EnableAscanMinimum/ “Minimum” in section “Cycle:X”	Useful only in case of compression (**coming soon**).
Saturation (pBufferSat)	CHWDeviceOEMPA::EnableAscanSaturation/ csHWDeviceOEMPA.EnableAscanSaturation/ “Saturation” in section “Cycle:X”	Useful to determine if the analog gain is too high (**coming soon**). If some data in this buffer is greater than 0, then the beam signal (after reception focusing) is too high.

Here is the relationship between “PointCount” (this is also the input member “CSubStreamAscan_0x0103::dataCount” ” C++ / “csHeaderAscan.dataCount” C# of the A-scan acquisition callback function parameter) and “PointFactor” (digitizing factor):

	Point Factor	**Digitizing frequency (MHz)**
Compression	ceil(((PointCount-1)*Step)*256.0/Range)	100 x PointFactor / 256
Decimation	round((round(Range/Step)+1.0)/PointCount)	100 / PointFactor

Step is 10 ns. Range is the A-scan range

Here are the available rectification options:

enumRectification (C++) / csEnumRectification (C#) / OEMPA file string	**Comment**
eSigned / csSigned / “Signed”	Not rectified.
eUnsigned / csUnsigned / “Unsigned”	Rectified.
eUnsignedPositive / csUnsignedPositive / “UnsignedPositive”	Rectified (only positive).
eUnsignedNegative / csUnsignedNegative / “UnsignedNegative”	Rectified (only negative).

See rectification examples in the paragraph 4.4.6.1 “C-scan/FW parameters”.

Here are the available bit-size options:

enumBitSize (C++) / csEnumBitSize (C#) / OEMPA file string	**Comment**
e8Bits / cs8Bits / “8Bits”	8 bits.
e12Bits / cs12Bits / “12Bits”	12 bits.
e16Bits / cs16Bits / “16Bits”	16 bits.
eLog8Bits / csLog8Bits / “Log8Bits”	Logarithmic unsigned 8 bits.

Depending on the A-scan setting, here is some input data of the A-scan acquisition callback:

CSubStreamAscan_0x0103 (C++) / csHeaderAscan_0x0103 (C#)	8 bits NotRectified	12 b NR	16 b NR	8 b R	12 b R	16 b R	Log 8 b
"dataSize" C++ / "dataSize" C#	1	2	2	1	2	2	1
"bitSize" C++ / "bitSize" C#	8	12	16	8	12	16	8
"sign" C++ / "sign" C#	1	1	1	0	0	0	0

b=bit, NR=NotRectified, R=Rectified.

Here are the available A-scan request modes:

enumAscanRequest (C++) / csEnumAscanRequest (C#) / OEMPA file string	**Comment**
eAscanAll / csAscanAll / "AscanAll"	All A-scans are sent by the HW.
eAscanSampled / eAscanSampled / "AscanSampled"	Only 1 A-scan among N is sent by the HW. The A-scan frequency should be specified.

Note that a C-scan HW gate can synchronize the A-scan start. The parameters used to specify the tracking are exactly the same as those used to synchronize a gate on another gate (see paragraph 4.4.8.4 "Echo Tracking API" for more information):

A-scan tracking	Comment
[**Cycle:0\Gate:1**]	
Start=5.0000 us	
Stop=15.0000 us	
[Cycle:X]	Specification of the A-scan for the cycle X
TrackingAscanEnable=1	=> the start of the A-scan is synchronized on a C-scan HW **gate**
TrackingAscanIndexGate=**1**	=> this **gate** is the **Gate 1**
TrackingAscanIndexCycle=**0**	=> this is the **Gate 1** of the **cycle 0 ("Cycle:0\Gate:1")**

4.4.5.2 DAC

DAC can be set using two different API functions:

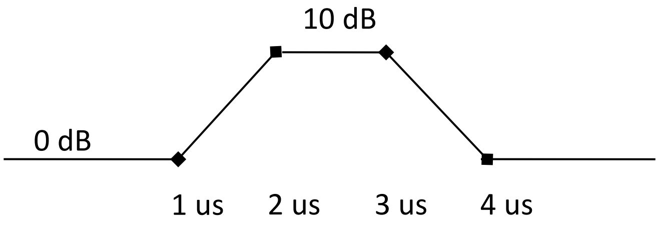
- Slope of the amplification.
    - Slope is the difference of amplification between two times (unit is dB/s).
- Gain of the amplification (version 1.1.5.0).
    - The first gain should be 0.0 dB. If you need a different value please use the "GainDigital" or the "BeamCorrection".
    - With this function there is an additional Boolean input parameter ("AutoStop") that can be used to automatically return the gain to 0dB. The slope for this last added point is +/-100.0 dB/us.

The total digital gain is:

GainTotal = GainDigital + BeamCorrection + DAC

The range of this total digital gain is: 0.0dB to +80.0 dB, so caution should be used when combining these three amplifications. The range of the DAC slope is approximately -320.0 dB/us to +319.0 dB/us. See the 'parameter list' document for more information.

Here is an example of DAC:

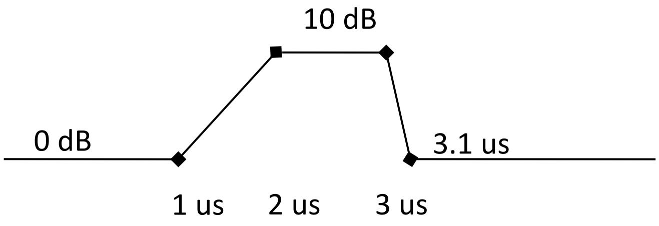


OEMPA file syntax (slope)	OEMPA file syntax (gain)

[Cycle:0]
DACEnable=1
DACTof.count=4
DACTof=1.000;2.000;3.000;4.000 us
DACSlope.count=4
DACSlope=10.0;0.0;-10.0;0 dB/us
	[Cycle:0]
	DACEnable=1
	DACTof.count=4
	DACTof=1.000;2.000;3.000;4.000 us
	DACGain.count=4
	DACGain=0.0;10.0;10.0;0.0 dB
	DACAutoStop=0

In the case of the gain it is also possible to use the "AutoStop" input Boolean parameter:

OEMPA file syntax (gain)	curve

[Cycle:0]
DACEnable=1
DACTof.count=4
DACTof=1.000;2.000;3.000 us
DACGain.count=4
DACGain=0.0;10.0;10.0 dB
DACAutoStop=1
	

Note that the start-time of the DAC can be synchronized by a C-scan HW gate. The parameters used to specify the tracking are the same parameters used to synchronize a gate on another gate (see paragraph 4.4.8.4 "Echo Tracking API" for more information):

Dac tracking	Comment
[**Cycle:0\Gate:1**]	

Start=5.0000 us	
Stop=15.0000 us	
[Cycle:X]	Specification of the Dac for the cycle X
TrackingDacEnable=1	=> the start of the dac is synchronized on a C-scan HW **gate**
TrackingDacIndexGate=**1**	=> this **gate** is the **Gate 1**
TrackingDacIndexCycle=**0**	=> this is the **Gate 1** of the **cycle 0 ("Cycle:0\Gate:1")**

Additional explanation about API function input parameters:

- API function with array input (C++)

CHWDeviceOEMPA::SetDACSlope(int iCycle,int &iCount,double *pdTime,float *pfSlope)

```
double adTime[4]={1e-6; 2e-6; 3e-6; 4e-6};//unit=second (s)
float afSlope[4]={10e6f; 0.0f; -10e6f; 0.0f};//unit=dB/s
int iCount=4;
if(pHWDeviceOEMPA->LockDevice())
{
       if(!pHWDeviceOEMPA->SetDACSlope(iCycleIndex, iCount, adTime,afSlope))
              {iError = 1000;goto return_error;}
       if(!pHWDeviceOEMPA->UnlockDevice())
              {iError = 1001;goto return_error;}
}
```

- API function with callback input (C++)

CHWDeviceOEMPA::SetDACSlope(int iCycle, int &iCountMax,structCallbackArrayFloatDac &dac)

```
bool  WINAPI  APISamplesCallbackGetSizeDac(structCallbackArrayFloatDac  *pCallbackArray,int
&iSize)
{
       int *piCount=(int*)pCallbackArray->apParameter[2];
       if(!piCount)
              return false;
       iSize = *piCount;
       return true;
}
bool  WINAPI  APISamplesCallbackSetSizeDac(structCallbackArrayFloatDac  *pCallbackArray,int
iSize)
{
       int *piCount=(int*)pCallbackArray->apParameter[2];
       if(!piCount)
              return false;
       if(iSize>4)
              return false;
       *piCount = iSize;
```

```
	return true;
}
bool  WINAPI  APISamplesCallbackSetDataDac(structCallbackArrayFloatDac  *pCallbackArray,int 
iIndex,double dTime,float fSlope)
{
	double *pdTime=(double*)pCallbackArray->apParameter[0];
	float *pfSlope=(double*)pCallbackArray->apParameter[1];
	int *piCount=(int*)pCallbackArray->apParameter[2];

	if(!pdTime || !pfSlope || !piCount)
		return false;
	if(iIndex<0)
		return false;
	if(iIndex>=*piCount)
		return false;
	pdTime[iIndex] = dTime;
	pfSlope[iIndex] = fSlope;
	return true;
}
bool  WINAPI  APISamplesCallbackGetDataDac(structCallbackArrayFloatDac  *pCallbackArray,int 
iIndex,double dTime,float fSlope)
{
	double *pdTime=(double*)pCallbackArray->apParameter[0];
	float *pfSlope=(double*)pCallbackArray->apParameter[1];
	int *piCount=(int*)pCallbackArray->apParameter[2];

	if(!pdTime || !pfSlope || !piCount)
		return false;
	if(iIndex<0)
		return false;
	if(iIndex>=*piCount)
		return false;
	dTime = pdTime[iIndex];
	fSlope = pfSlope[iIndex];
	return true;
}
double adTime[4]={1e-6; 2e-6; 3e-6; 4e-6};
float afSlope[4]={10e6f; 0.0f; -10e6f; 0.0f};
int iCount=4;
callbackArrayFloatDac.pSetSize = APISamplesCallbackSetSizeDac;
callbackArrayFloatDac.pSetData = APISamplesCallbackSetDataDac;
callbackArrayFloatDac.pGetSize = APISamplesCallbackGetSizeDac;
callbackArrayFloatDac.pGetData = APISamplesCallbackGetDataDac;
callbackArrayFloatDac.apParameter[0] = &dacWr[iCycleIndex];
if(pHWDeviceOEMPA->LockDevice())
{
	if(!pHWDeviceOEMPA->SetDACSlope(iCycleIndex,iCount,callbackArrayFloatDac))
		{iError = 1000;goto return_error;}
	if(!pHWDeviceOEMPA->GetDACSlope(iCycleIndex,iCount,callbackArrayFloatDac))
```

```
              {iError = 1000;goto return_error;}
       if(!pHWDeviceOEMPA->UnlockDevice())
              {iError = 1001;goto return_error;}
}
```

- API function for C#

csHWDeviceOEMPA.SetDACSlope(int iCycle,ref acsDac dac)

```
acsDac dac;
dac.list = new csDac[4];
dac.list[0] = new csDac(1e-6, 10.0e6f);
dac.list[1] = new csDac(2e-6, 0.0f);
dac.list[2] = new csDac(3e-6, -10.0e6f);
dac.list[3] = new csDac(4e-6, 0.0f);
if(hwDeviceOEMPA.LockDevice())
{
       if (!hwDeviceOEMPA.SetDACSlope(0, ref dac))
              bRet = false;
       if(!hwDeviceOEMPA.UnlockDevice())
              bRet = false;
}
```

4.4.5.3 Filters

You can define up to 15 digital filters for all cycles. Then you can choose inside this set which one you want for each cycle (this is the filter index). Only FIR (Finite Impulse Response) filter are supported - no IIR (Infinite Impulse Response). The filter is implemented inside the FW: the sample frequency is 50 MHz, and tap count is 64, the formula of the filter is the following:

$$y(k) = \sum_{n=0}^{64} coef(n).x(k-n)$$

x = input signal
y = output signal
coef = coefficients, symmetrical structure:
coef(64-i) = coef(i) i range: 0 to 32

Because the coefficients are symmetrical, the API takes only half of them:

API SetFilter(enumOEMPAFilterIndex eFilter,WORD &wScale,short wValue[34]) C++
SetFilter(csEnumOEMPAFilterIndex eFilter,WORD &wScale,array<short> wValue[34]) C#
"FilterCount" key in section "Root" and "Title", "Scale", "Coefficient" keys
in section "Filter:X" (OEMPA file).

where "wValue[i+1]=coef(i)" I range is 0 to 32, "wValue[0]" should be 0. The scale is used as a divider after the summation. Input parameters of functions "SetFilter" are quite complex. The function named "computeFIRcoefficientsKaiser64_50MHz" in the dll "ComputeFIR.dll" can help you to set values for those input parameters. Input parameters of this last function are the same as the inputs of the dialog "DefaultConfiguration", so it is easier to use. Here is an example with the following filter: FIR High Pass 0.5 MHz, stop gain 40 dB:

```
short int hk[g_iDEFAULT_ORDER_FIR_HW];
WORD wScaleCoef;
bool bRet;
bRet = CComputeFIR::computeFIRcoefficientsKaiser64_50MHz(40.0, eHighPass,
                                0.0, 0.5, hk, wScaleCoef, 0, NULL);
if(!bRet)
{
        //error
}
```

The result is the following: "wScaleCoef=15" and "hk={0,-44,-55,-68,-82,-97,-115,-133,…-574,-592,-609,-623,-634,-643,-650,-654,32113}" and then you can call "SetFilter(eOEMPAFilter1,wScaleCoef,hk)" (member of "CHWDeviceOEMPA").

You can also read the section concerning "Default Configuration" in *Software_Utilities.pdf*.

For each cycle, a filter index in the range 1 to 15 must be selected.

API CHWDeviceOEMPA::SetFilterIndex(int iCycle,enumOEMPAFilterIndex eFilterIndex) C++
csHWDeviceOEMPA.SetFilterIndex(int iCycle,csEnumOEMPAFilterIndex eFilterIndex) C#
"FilterIndex" key in section "Cycle:X" (OEMPA file).

4.4.6 C-scan

The stream format coming from the FW is not openly documented. The user only needs to understand the acquisition callback function. To get C-scan you can register a callback function that delivers all C-scan data sent by the device. The callback function prototype is named "pTypeAcquisitionCscan_0x00010X02" (defined in "UTBasicType.h" for C++) and "TypeAcquisitionCscan_0x00010X02" (delegate defined in "csDriverOEMPA" for C#):

```
//C++
typedef UINT (WINAPI *pTypeAcquisitionCscan_0x00010X02)(void
*pAcquisitionParameter,structAcqInfoEx &acqInfo,const CStream_0x0001 *pStreamHeader,const
CSubStreamCscan_0x0X02 *pCscanHeader,const structCscanAmp_0x0102 *pBufferAmp, const
structCscanAmpTof_0x0202 *pBufferAmpTof);

//C#
public delegate int AcquisitionCscan_0x00010X02(Object pAcquisitionParameter, ref
csAcqInfoEx acqInfo, ref csHeaderStream_0x0001 pStreamHeader, ref csSubStreamCscan_0x0X02
pCscanHeader, ref csCscanAmp_0x0102[] pBufferAmp, ref csCscanAmpTof_0x0202[]
pBufferAmpTof);
```

Here are the input parameters:

- “pAcquisitionParameter”: first common input. This is a user-defined input parameter.
- “acqInfo”: second common input and general information, such as encoder, digital inputs, maximum temperature reading and multiple channel indexes.
- “pStreamHeader”: third common input, frame header with the setting identifier.
- “pCscanHeader”: C-scan header (“CSubStreamCscan_0x0X02” C++ / “csSubStreamCscan_0x0X02” C#).
- “pBufferAmp”: significant only if the “version” of “pCscanHeader” is 1. Array of gates result (with amplitude only, “structCscanAmp_0x0102” for C++ / “csCscanAmp_0x0102” for C#).
- “pBufferAmpTof”: significant only if the “version” of “pCscanHeader” is 2. Array of gates result (with amplitude and time of flight, “structCscanAmpTof_0x0202” for C++ / “csCscanAmpTof_0x0202” for C#).

The first three inputs are explained in paragraph 4.4.7 “Callback common inputs”.

Here are some of the members of “**CSubStreamCscan_0x0X02**” C++ / “**csSubStreamCscan_0x0X02**” C#:

Member	**Comment**
“version”	This value is 1 for amplitude result only and 2 for amplitude and time of flight results.
“seqLow” and “seqHigh”	Sequence number (LSB and MSB).
“cycle”	Cycle number.
“count”	Gate count.

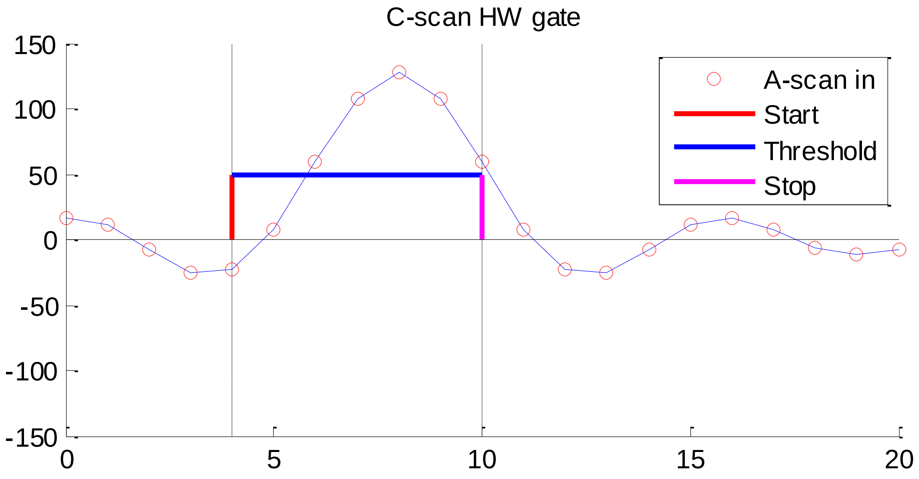
The following are the members of “structCscanAmp_0x0102” for C++ / “csCscanAmp_0x0102” for C# and “**structCscanAmpTof_0x0202**” for C++ / “**csCscanAmpTof_0x0202**” for C#:

Member	**Comment**
“gateId”	Gate identifier (range is 0 to 3), you must enable the corresponding gate in your setting.
“byAmp”	Amplitude result: signed or unsigned value depending on the gate setting. Unsigned value if the amplitude mode is “PeakToPeak” or if the rectification mode is rectified.
“AmpOverThreshold”	Information to determine if the amplitude is greater than the threshold.
“wTof”	Time of flight result (signed value).
“TofValidity”	Information to determine if the time of flight result is valid.

“IFNotInitialized”	This information lets you know if the gate result used for synchronization has been valid for the first time. This information is useful only if echo tracking is used to synchronize the gate.
“IFOldReference”	This information lets you know if the gate result used for synchronization is an old value or if it has been refreshed (new detection). This information is useful only if echo tracking is used to synchronize the gate.

See paragraph 4.4.7.3 “Acquisition channel” for comments about acquisition channel members.

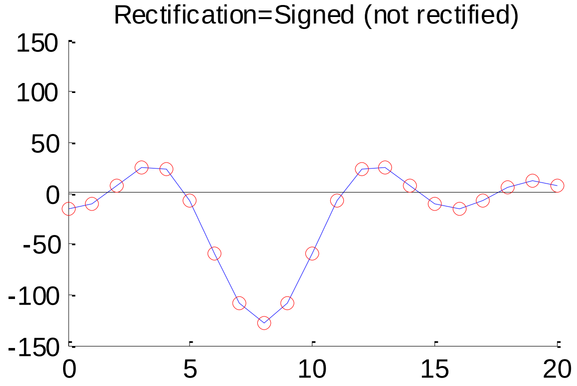
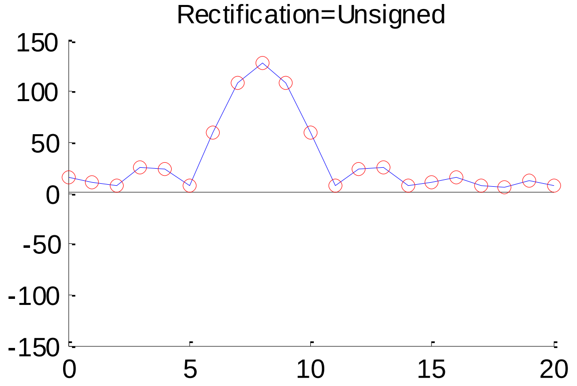
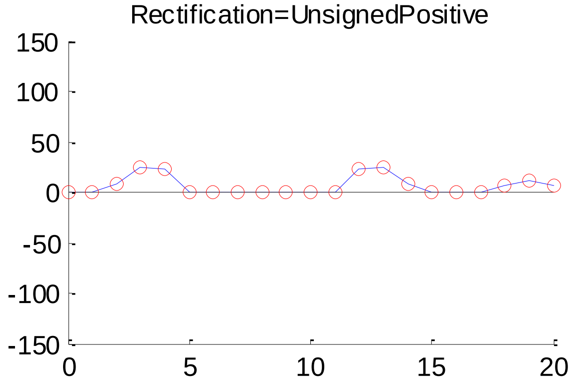
4.4.6.1 FW parameters

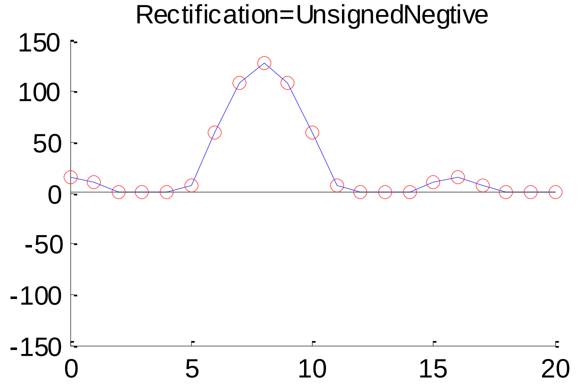


Feature	**API C++ / API C# / OEMPA file key**
Start	CHWDeviceOEMPA::SetGateStart / csHWDeviceOEMPA.SetGateStart / “Start” in section “Cycle:X\Gate:Y”
Stop	CHWDeviceOEMPA::SetGateStop / csHWDeviceOEMPA.SetGateStop / “Stop” in section “Cycle:X\Gate:Y”
Threshold, amplitude mode, time of flight mode and rectification	CHWDeviceOEMPA::SetGateModeThreshold / csHWDeviceOEMPA.SetGateModeThreshold / “Threshold”, “ModeAmp”, “ModeTof”, “Rectification” in section “Cycle:X\Gate:Y”

Rectification for C-scan could be different than the one for A-scan. Here are the different rectifications for C-scan results:

enumRectification (C++) /	**Figure**

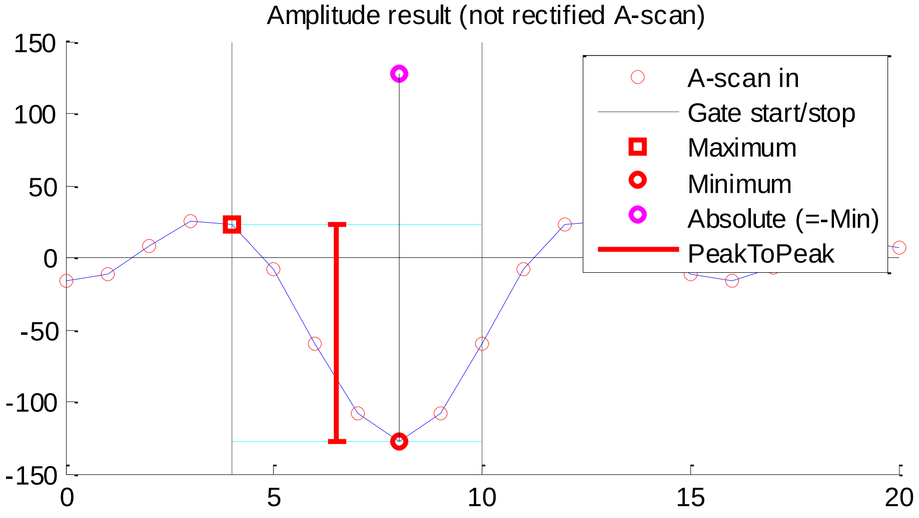
csEnumRectification (C++) / OEMPA file values of "Rectification"	
eSigned / csSigned / "Signed"	
eUnsigned / csUnsigned / "Unsigned"	
eUnsignedPositive / csUnsignedPositive / "UnsignedPositive"	

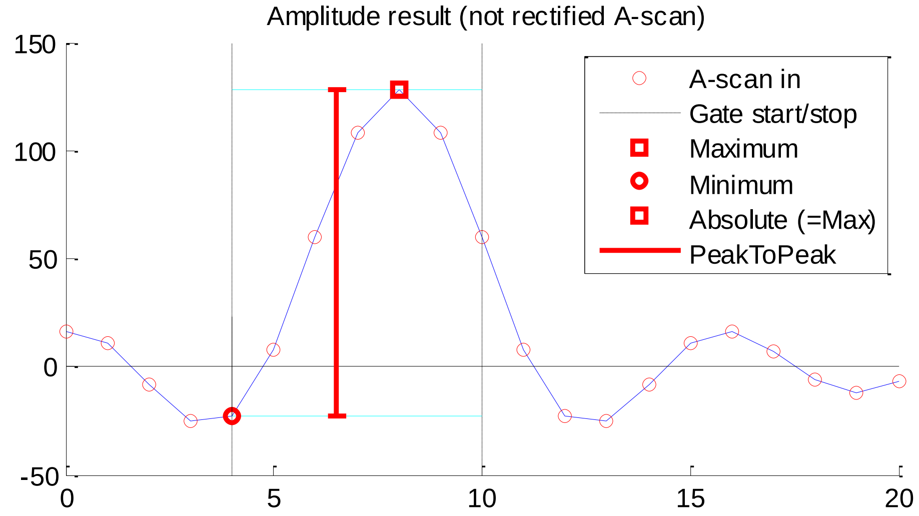
eUnsignedNegative / csUnsignedNegative / “UnsignedNegative “	

Amplitude mode (not significant if rectification is used, that is if rectification is different than “eSigned”):

enumGateModeAmp (C++) / **csEnumGateModeAmp (C++) /** **OEMPA file values of “ModeAmp”**	**Comment** See figure “Amplitude result (not rectified A-scan)”
eAmpAbsolute / csAmpAbsolute / “Absolute”	The absolute maximum value of the absolute minimum and the maximum.
eAmpMaximum / csAmpMaximum / “Maximum”	The maximum value.
eAmpMinimum / csAmpMinimum / “Minimum”	The minimum value.
eAmpPeakToPeak / csAmpPeakToPeak / “PeakToPeak”	The peak to peak value.

One bit is sent to the computer with the amplitude to specify if the result is over the threshold or not (bit “AmpOverThreshold”).





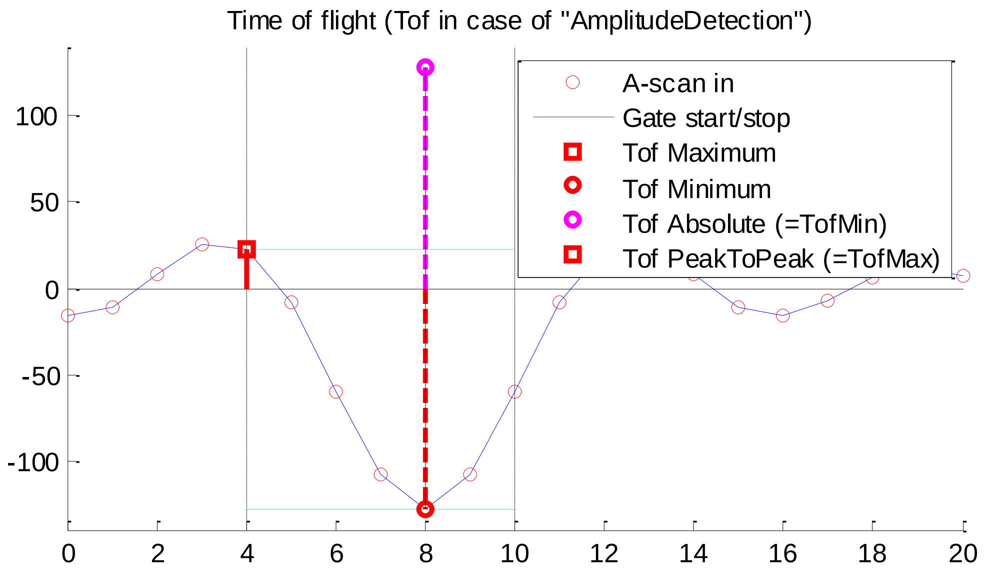
The amplitude of the “Absolute” could be the amplitude of the opposite of the minimum (in case of negative minimum) or the amplitude of the maximum.

Time of flight mode:

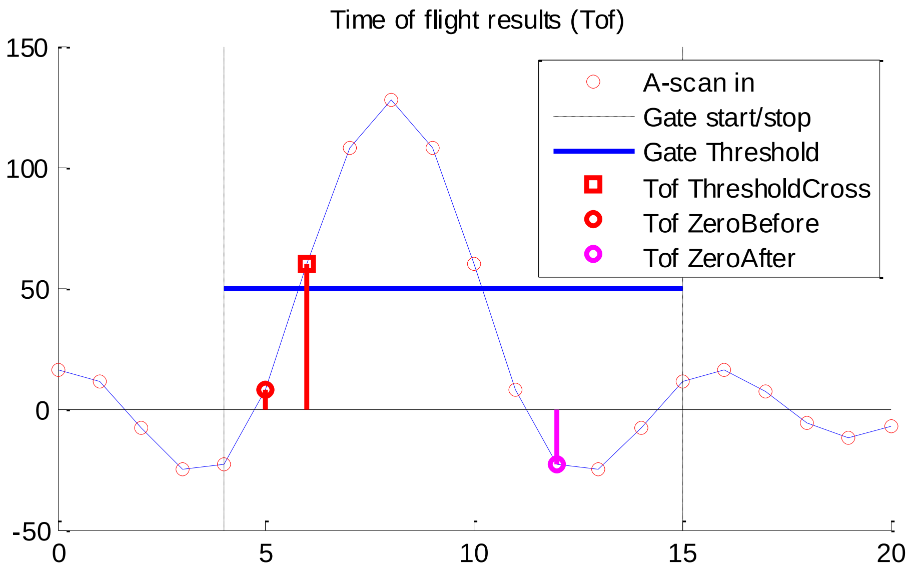
enumGateModeTof (C++) / **csEnumGateModeTof (C++) /** **OEMPA file values of “ModeTof”**	**Comment**	**Threshold parameter**
eTofAmplitudeDetection / csTofAmplitudeDetection / “AmplitudeDetection“	The time of flight is for the detected amplitude. See figure “Time of flight result (“AmplitudeDetection”)”.	Not used.

eTofThresholdCross / csTofThresholdCross / “ThresholdCross “	The time the A-scan cross the threshold is the final result.	Useful
eTofZeroFirstAfterThresholdCross / csTofZeroFirstAfterThresholdCross / “ZeroFirstAfterThresholdCross“	The first time the A-scan is 0 after crossing the threshold is the final result.	Useful
eTofZeroLastBeforeThresholdCross / csTofZeroLastBeforeThresholdCross / “ZeroLastBeforeThresholdCross“	The last time the A-scan is 0 before crossing the threshold is the final result.	Useful

One bit is sent to the computer to specify if the returned time of flight is valid or not (bit “TofValidity”), if the time of flight is out of the gate its value is 0 and the valid bit is 0.



The time of flight for the absolute mode could be either the time of flight of the minimum or the time of flight of the maximum, whereas PeakToPeak is the time of flight of the maximum.



In the case that the time of flight is out of the gate, the return value is 0 and the valid bit is 0 (bit “TofValidity”). Due to sampling, the TOF result cannot be exactly the time where the signal crosses the threshold or the "0 axis", the result is taken on the first CLK period after the cross is detected.

4.4.7 Callback common inputs

The first three inputs are the same for A-scan and C-scan callback:

- “pAcquisitionParameter”: this is a user defined input parameter. This value is set at the same time callback function is registered. Use the function “CHWDevice::SetAcquisitionParameter” for C++ and “csHWDevice.SetAcquisitionParameter” for C# to set this value.
- “acqInfo”: general information, such as encoder, digital inputs and multiple channel information. See below.
- “pStreamHeader”: frame header with the setting identifier. This is useful to know if acquisition data has been produced with an old setup or current one.
    - This is an optional parameter (delivered inside each acquisition data).
    - The application setting software is responsible to set it (see function “SetSettingId”).

The second input (“structAcqInfoEx &acqInfo” for C++ and “csAcqInfoEx acqInfo” for C#) is used to send many types of data:

- Encoder.
- Acquisition channel information.

- Full Matrix Capture information.
- Maximum Temperature reading

4.4.7.1 Time Stamp

"timeStampLow" and "timeStampHigh" are also a common member of main specific inputs:

Callback type	Structure
A-scan	CSubStreamAscan_0x0103
C-scan	CSubStreamCscan_0x0X02
IO stream	CSubStreamIO_0x0101

The unit of the time stamp is microsecond, it is a 64-bit value (low and high 32 bits). This value is set to 0 at start-up. It is only incremented when the pulsers are enabled. With version 1.1.5.0 the user can reset the value with the following API:

4.4.7.2 CHWDeviceOEMPA::ResetTimeStampEncoder

The parameter gives raw encoder value and decoded values (encoder resolution is taken into account) if the driver is used to manage encoders. You can disable the driver from managing encoders, but you must then manage them yourself - see the paragraph 4.5.4 "Encoder management" within "Special features".

4.4.7.3 Acquisition channel

The member "bMultiChannel" of the parameter "structAcqInfoEx &acqInfo" for C++ and "csAcqInfoEx acqInfo" for C# specifies if multiple acquisition is enabled for the high level API. It is possible to set this value with the following API:

Set (C++ / C#)	Get (C++ / C#)
CSWDeviceOEMPA::EnableMultiChannel / csSWDeviceOEMPA.EnableMultiChannel	CSWDeviceOEMPA::IsMultiChannelEnabled / csSWDeviceOEMPA.IsMultiChannelEnabled

This software parameter is saved in the software configuration file. If you change this parameter, it is not taken into account until the driver is restarted. The acquisition channel identifiers are not linked to this software parameter, so you can use them independently.

'Acquisition channel' means that the acquisition data is not only delivered with the cycle information but also with one or several acquisition channel identifiers. You can use these identifiers as you want. A channel identifier is added to the setting and sent to the FW for each required acquisition data (A-scan or C-scan). A channel identifier is a 16-bit value that you can use as you want. Its default value is 0xffff, which means it is not significant. Although the name of each identifier is defined, the name has no significance; so **you can use those acquisition**

channel identifiers for the purpose you want. There are 3 channel identifiers for A-scan and only 1 for C-scan.

For example, in the case of A-scan, the first identifier could be the channel index and the second identifier the sub-cycle index. The GainDigital could be the same value for all sub-cycles of each channel. The BeamCorrection will be specific for each cycle.

Here are the acquisition channel identifiers for A-scan:

A-scan channel Identifier	Members of "**CSubStreamAscan_0x0103**" C++ / "**csSubStreamAscan_0x0103**" C#	"CHWDeviceOEMPA" function (C++) / "csHWDeviceOEMPA" function (C#) / OEMPA file syntax
#1	"FWAcqIdChannelProbe"	SetAscanAcqIdChannelProbe / SetAscanAcqIdChannelProbe / "AcqIdChannelProbe" in section "Cycle:X"
#2	"FWAcqIdChannelScan"	SetAscanAcqIdChannelScan / SetAscanAcqIdChannelScan / "AcqIdChannelScan" in section "Cycle:X"
#3	"FWAcqIdChannelCycle"	SetAscanAcqIdChannelCycle / SetAscanAcqIdChannelCycle / "AcqIdChannelCycle" in section "Cycle:X"

Here are the acquisition channel identifiers for C-scan:

C-scan channel Identifier	*Members of* "**structCscanAmp_0x0102**" for C++ / "**csCscanAmp_0x0102**" for C# *and* "**structCscanAmpTof_0x0202**" for C++ / "**csCscanAmpTof_0x0202**" for C#	"CHWDeviceOEMPA" function (C++) / "csHWDeviceOEMPA" function (C#) / OEMPA file syntax
#1	"wAcqIdAmp"	SetGateAcqIDAmp / SetGateAcqIDAmp / "AcqIDAmp" in section "Cycle:X\Gate:Y"
#2	"wAcqIdTof"	SetGateAcqIDTof / SetGateAcqIDTof / "AcqIDTof" in section "Cycle:X\Gate:Y"

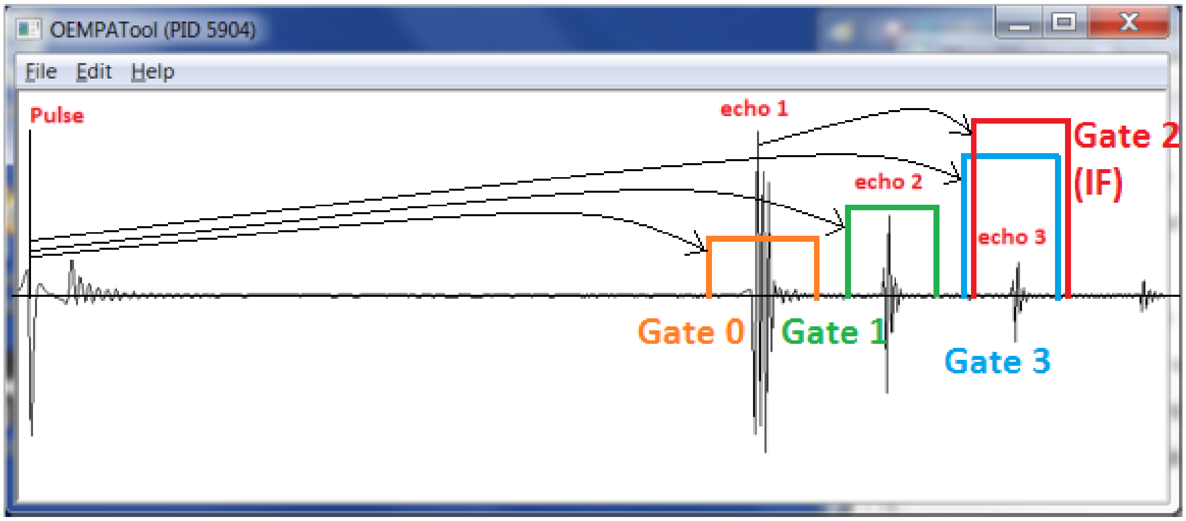
4.4.8 HW Gates

Some explanation is required concerning HW gates:

- Wedge delay: this input parameter is a time reference, not an ultrasonic parameter.
    - This is a reference time for setting input: start of gates (A-scan and C-scan gates).
    - This is a reference time for results output: time of flight of C-scan gates.
- Echo tracking: one gate could be used to synchronize another one.

- The Gate tracking feature allows you to synchronize the Start and Stop of one gate with the time of flight result of another. Without this feature, the time reference is the pulse, but with the Gate tracking, the reference becomes the time of flight result of a HW C-scan gate from the previous cycle.

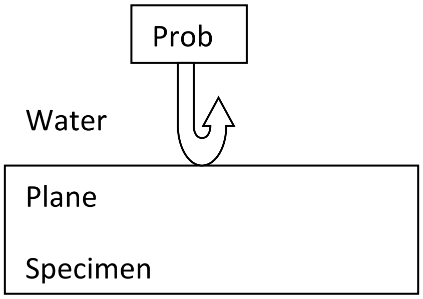
We will use “OEMPATool” to show an example with the following A-scan (a single cycle):



In this example the threshold is not significant.

Pulse is on the left (**Pulse**), entrance echo (**echo 1**) on the right with 3 others echoes (**echo 2**, **echo 3**, and another one which isn’t considered) in the specimen. The A-scan range is 50 us and the wedge delays are null.

OEMPA file	**Example diagram**

[Cycle:0]
Start=0.0 us
Range=50.0000 us
[Cycle:0\Pulser]
WedgeDelay=**0.0000** us
[Cycle:0\Receiver]
WedgeDelay=**0.0000** us
	

4.4.8.1 C-scan Gates 0 and 1

Here are the wanted gates to detect “**echo 1**” and “**echo 2**”:

	Gate 0	Gate 1
Start (us)	25	35
Range (us)	10	10

OEMPA file	**Comment**
[Cycle:0]	
Start=0.0 us	
Range=50.0000 us	
GateCount = 2	File format: 2 gates section are defined.
[Cycle:0\Gate:0]	
Enable=1	Enable the HW Gate 0.
Start=25.0000 us	
Stop=35.0000 us	
[Cycle:0\Gate:1]	
Enable=1	Enable the HW Gate 1.
Start=35.0000 us	
Stop=45.0000 us	
[Cycle:0\Pulser]	
WedgeDelay=0.0000 us	Wedge delays are null.
[Cycle:0\Receiver]	
WedgeDelay=0.0000 us	

Here is the gate detection (button “Status” of the “HW dialog”):

	Amplitude	Time of flight
Gate 0 (**echo 1**)	94 %	32.38 us
Gate 1 (**echo 2**)	58 %	38.12 us

4.4.8.2 Wedge delay

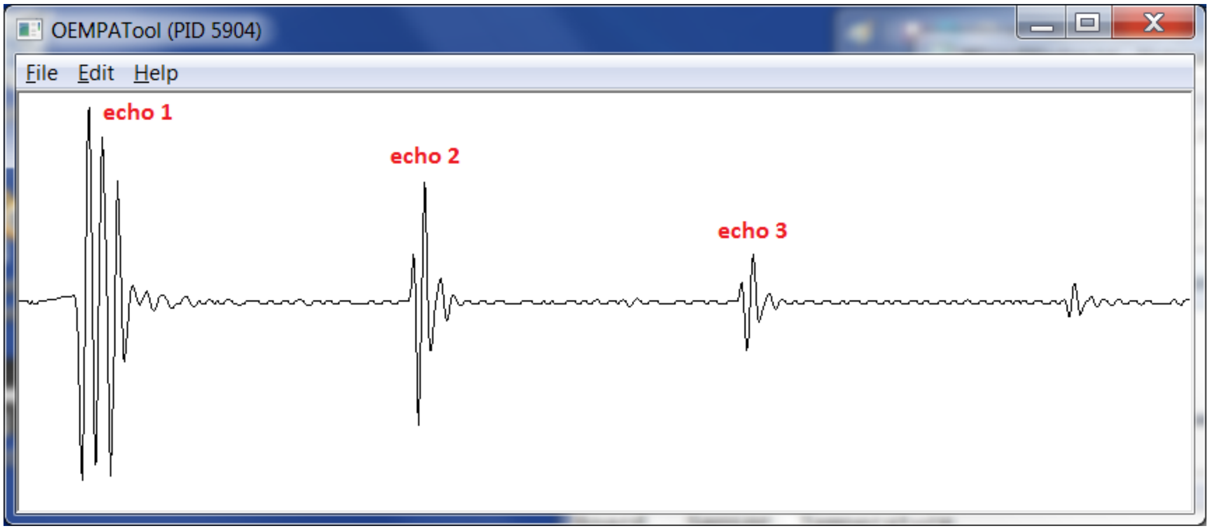
The interesting part of the A-scan is inside the specimen. In order to remove the wedge part, two special parameters “WedgeDelay” are used; one for emission and another one for reception. The reference time is near the entrance echo instead of the pulse:

ReferenceTime = WedgeDelay_SectionPulser + WedgeDelay_SectionReceiver
ReferenceTime = 15.58 + 15.58 = 31.16 us

With this new reference time, it is possible to keep the A-scan "Start" with value 0 us and to decrease the A-scan "Range". Gates start should also be updated:

OEMPA file	Comment
[Cycle:0]	
Start=0.0 us	
Range=20.0000 us	50-31.16 is about 20 us
[Cycle:0\Gate:0]	
Start=-5.0000 us	25-31.16 is about -5 us
Stop=5.0000 us	35-31.16 is about +5 us
[Cycle:0\Gate:1]	
Start=5.0000 us	35-31.16 is about +5 us
Stop=15.0000 us	45-31.16 is about +15 us
[Cycle:0\Pulser]	
WedgeDelay=**15.5800** us	
[Cycle:0\Receiver]	
WedgeDelay=**15.5800** us	

Here is the A-scan:



Entrance echo is on the left (echo 1) and the 3 others echoes (echo 2, echo 3 and another one) on the right. Here are the gate detections (button "Status" of the "HW dialog"):

	Amplitude	Time of flight
Gate 0 (echo 1)	94 %	1.20 us
Gate 1 (echo 2)	58 %	6.96 us

Here are the absolute times of flight (time reference is the pulse):

- Gate 0 detection = 1.20 + 31.16 = 32.36 us (instead of 32.38 us when wedge delay is null)
- Gate 1 detection = 6.96 + 31.16 = 38.12 us (instead of 38.12 us when wedge delay is null)

So the results are the same whether you use the wedge delay or not. But with wedge delay many benefits are available, for example:

- To easily display the S-scan.
- To define the gates referenced to the interface.

Instead of using time, it is possible to use a distance unit. As the speed is 6300 m/s here are the results:

	Time (us)	Distance (mm)
A-scan start	20 us	63
Gate 0 start	-5	-15.75
Gate 0 stop	5	15.75
Gate 1 start	5	15.75
Gate 1 stop	15	47.25

Formula is “Distance = 6300*Time/2”.

4.4.8.3 Echo tracking (IF)

We will now demonstrate the usage of echo tracking, in the case of our basic example, to detect the third echo (**echo 3**). This echo will be detected by two different gates:

- Gate 2 synchronized on the gate 1 time of flight result (echo tracking).
- Gate 3 synchronized on the pulse (no echo tracking).

New parameters that are disabled by default should be used now:

OEMPA file	**Comment**

[Cycle:0]
Start=0.0 us
Range=20.0000 us
[Cycle:0\Gate:0]
Start=-5.0000 us
Stop=5.0000 us
	Gate 0 definition

[Cycle:0\Gate:1]	Gate 1 definition
Start=5.0000 us	
Stop=15.0000 us	
[Cycle:0\Gate:2]	Gate 2 definition
Start=5.0000 us	=> value added to the time of flight result of gate 1
Stop=15.0000 us	=> value added to the time of flight result of gate 1
TrackingStartEnable=1	=> the start of the gate is synchronized on another **gate**
TrackingStartIndexGate=**1**	=> this **gate** is the **Gate 1** (used to detect **echo 2**)
TrackingStartIndexCycle=**0**	=> this is the **Gate 1** of the **cycle 0** **("Cycle:0\Gate:1")**
TrackingStopEnable=1	=> the stop of the gate is synchronized on another **gate**
TrackingStopIndexGate=**1**	=> this **gate** is the **Gate 1** (used to detect **echo 2**)
TrackingStopIndexCycle=**0**	=> this is the **Gate 1** of the **cycle 0** **("Cycle:0\Gate:1")**
[Cycle:0\Gate:3]	Gate 3 definition
Start=10.0000 us	=> to detect the third echo
Stop=20.0000 us	=> to detect the third echo
[Cycle:0\Pulser]	
WedgeDelay=**15.5800** us	
[Cycle:0\Receiver]	
WedgeDelay=**15.5800** us	

Here are the results:

	Gate 0 (**echo 1**)	Gate 1 (**echo 2**)	Gate 2 (**echo 3**)	Gate 3 (**echo 3**)
Amplitude	94 %	58 %	30 %	30 %
Time of flight	1.20 us	6.95 us	12.56 us	12.56 us

So gates 2 and 3 detect the same echo (**echo 3**): echo tracking is fine!

4.4.8.4 Echo Tracking API

Here is the API:

C-scan gate	**Function (C++) / Function (C#) / OEMPA file keys**	**Function parameters** iCycle is the index of the cycle and iGate is the gate index for which echo tracking is set.
Start	CHWDeviceOEMPA::SetTrackingGateStart / csHWDeviceOEMPA.SetTrackingGateStart / "TrackingStartEnable", "TrackingStartIndexGate" and	(int iCycle, int iGate, bool bEnable, int iTrackingCycleIndex, iTrackingCycleGate) "bEnable" stands for "TrackingStartEnable",

	"TrackingStartIndexCycle" in section "Cycle:X_Gate:Y"	"iTrackingCycleIndex " stands for "TrackingStartIndexCycle" and "iTrackingGateIndex" stands for "TrackingStartIndexGate".
Stop	CHWDeviceOEMPA::SetTrackingGateStop / csHWDeviceOEMPA.SetTrackingGateStop / "TrackingStopEnable", "TrackingStopIndexGate" and "TrackingStopIndexCycle" in section "Cycle:X_Gate:Y"	(int iCycle, int iGate, bool bEnable, int iTrackingCycleIndex, iTrackingCycleGate) "bEnable" stands for "TrackingStopEnable", "iTrackingCycleIndex " stands for "TrackingStopIndexCycle" and "iTrackingGateIndex" stands for "TrackingStopIndexGate".

In the same way that the HW C-scan gate can be used to synchronize another C-scan gate, it is also possible to synchronize:

- The A-scan gate
- The DAC

Here is then API for the A-scan start gate:

Function (C++) / Function (C#) / OEMPA file keys	**Function parameters** iCycle and iGate are the index of the gate for which echo tracking is set.
CHWDeviceOEMPA::SetTrackingAscan / csHWDeviceOEMPA.SetTrackingAscan / "TrackingAscanEnable", "TrackingAscanIndexGate" and "TrackingAscanIndexCycle" in section "Cycle:X"	(int iCycle, int iGate, bool bEnable, int iTrackingCycleIndex, iTrackingCycleGate) "bEnable" stands for "TrackingAscanEnable", "iTrackingCycleIndex " stands for "TrackingAscanIndexCycle" and "iTrackingGateIndex" stands for "TrackingAscanIndexGate".

Here is the API for the DAC:

Function (C++) / Function (C#) / OEMPA file keys	**Function parameters** iCycle and iGate are index of the gate for which echo tracking is set.
CHWDeviceOEMPA::SetTrackingDac / csHWDeviceOEMPA.SetTrackingDac / “TrackingDacEnable”, “TrackingDacIndexGate” and “TrackingDacIndexCycle” in section “Cycle:X”	(int iCycle, int iGate, bool bEnable, int iTrackingCycleIndex, iTrackingCycleGate) “bEnable” stands for “TrackingDacEnable”, “iTrackingCycleIndex “ stands for “TrackingDacIndexCycle” and “iTrackingGateIndex“ stands for “TrackingDacIndexGate”.

4.4.9 Data loss

When the throughput is too high you can lose some acquisition data. This error is not notified as others (see for example the group “Acquisition” of the “OEMPAApplicationExample” dialog).

This problem is not the same as a communication error.

Function	Comment
CSWDevice::GetStreamError (C++) csSWDevice.GetStreamError (C#)	To get communication error count, this is a critical error.
CSWDevice::GetLostCountAscan (C++) csSWDevice.GetLostCountAscan (C#)	To query how many acquisition A-scans have been lost. Indicates that the throughput is too high.
CSWDevice::GetLostCountCscan (C++) csSWDevice.GetLostCountCscan (C#)	To query how many C-scans have been lost. Indicates that the throughput is too high.
CSWDevice::GetLostCountUSB3 (C++) csSWDevice.GetLostCountUSB3 (C#)	To query how many substreams (A-scan, C-scan, IO) have been lost over USB3. Indicates data is unable to be processed quickly enough.
CSWDevice::GetLostCountEncoder (C++) csSWDevice.GetLostCountEncoder (C#)	To query how many encoder acquisitions were lost. This indicates that the mechanical speed is too high. A new trigger was requested by the encoder before the previous one had completed.

Generally, data loss occurs because the A-scan throughput is too high. Here are some ways to avoid data loss:

- Disable the A-scan

- Decrease the PRF (increase TimeSlot)
- Increase the A-scan compression
- Use the “1/n A-scan” feature (refer to paragraph 4.4.5.1 “FW parameters” within 4.4.5 “A-scan” in this document)

Use caution when processing the data loss event: a little more processing on the computer (hard disk, heavy processing time) is enough to add delay in the hardware and more acquisition data could be lost by the device.

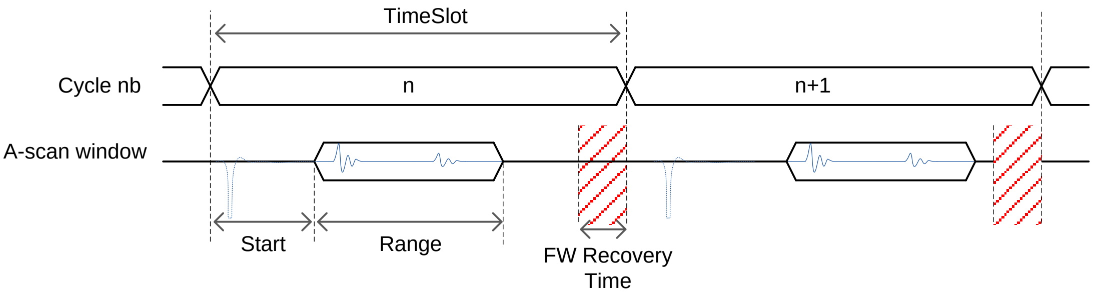
If the throughput is too high, the device will send a notification error. It is not possible to print this error in the dump file or to display it in the “UTKernelError” because it adds more computational burden that can further disturb the device and cause more acquisition data loss (snowball effect). It is better to process this error by the application software itself (for example, by displaying it) to prevent further acquisition loss.

4.4.10 FW Recovery Time

A “cycle” in Phased Array mode is basically the interval of time in which the hardware sends pulses to the UT probe and acquire the corresponding data. In a sectorial scan, one cycle will correspond to an angle. In a linear scan, one cycle will correspond to an aperture.

Before the start of a new cycle, the firmware needs to have a time margin in order to be ready for the next cycle. This is what is called "FW Recovery Time". This parameter is slightly different between Phased Array, Multi-Channel (see section 6.4), or Full Matrix Capture (see section 7.4) modes.

The A-scan window must be finished before the beginning of the "FW Recovery Time".



$$\boxed{TimeSlot \ \geq AscanStart + AscanStop + FWRecoveryTime} \quad \text{(PA.a)}$$

The value of "FWRecoveryTime" can vary depending on the FW version. Basically, it is either 12.5us or 3.5us. When a HW calibration is performed (see paragraph 4.5.3, “Calibration”), the "time offset" result is added to this value (usually less than 1us). An API function allows the developer to know the value (available from 1.1.5.3t and later).

Function	Comment
CSWDevice::GetFWAscanRecoveryTime (C++) csSWDevice.GetFWAscanRecoveryTime (C#)	To get the values of the "FW Recovery Time" values of the Hardware connected. Note: Includes the "time offset" result of HW calibration

4.5 Special features

4.5.1 Configuration file

Characteristics of configuration files:

- One single software configuration file used by the driver.
- As many hardware configuration files (OEMPA file) as you want to setup the device.
    - “Configuration file” without further specification means “hardware configuration file”.

The (hardware) configuration file has two formats: binary and text format. The text format is easy to use but in the case of big files it can take longer to read. In the case of text files, some parameters can be used with the default values so you don’t have to specify them in the file. In this case the text file is shorter.

This file format is useful to debug your application software because you can exit your program and run “OEMPATool” to connect the device and read-back its configuration and save it in a file. Opening this file with any text editor can show you the FW parameters.

There are two types of configuration files:

- The default configuration file is automatically loaded at connection time.
- The inspection configuration file is loaded by the user after connection. One file is used for each inspection.

There are two default configuration files that are loaded at connection time:

- The global default configuration file:
    - This file is loaded at connection time only if the following conditions are all valid:
        - The device has not yet loaded any cycle.

        - The parameter (“LoadDefaultSetup”) is enabled. This parameter is stored in the software configuration file.
        - No specific default configuration file has been saved.
    - This default configuration file has only one cycle with an aperture of one element for emission and reception: the first element of the probe is used only.
    - This file is the same for all devices.
- The specific default configuration file:
    - You will often use the same encoder or the same filters for all inspections. It is better to define these common parameters in a single configuration file. Then, when you want to change the common parameters, you need only to change them in this single file.
    - This file is specific for each device.
    - The name is “HW_XXX.txt” where “XXX” is the board name of the device.
    - You can use the “DefaultConfiguration” SW to create it.

You can specify the “LoadDefaultSetup” with the following function of the high level API:

Set (C++ / C#)	Get (C++ / C#)
CSWDeviceOEMPA::EnableLoadDefaultSetup / csSWDeviceOEMPA.EnableLoadDefaultSetup	CSWDeviceOEMPA::IsLoadDefaultSetupEnabled / csSWDeviceOEMPA.IsLoadDefaultSetupEnabled

This software parameter is saved in the software configuration file. If changed, it is not taken into account until the driver is restarted.

Here is the API to manage the hardware configuration file:

Feature	**Function C++ / Function C#**
To get the global default setup file for Phased Array (loaded at connection time)	CSWDevice::GetSetupFileDefaultPA / csSWDevice.GetSetupFileDefaultPA
To get the global default setup file for Multiple Channel (loaded at connection time)	CSWDevice::GetSetupFileDefaultMC / csSWDevice.GetSetupFileDefaultMC
To get the specific default setup file (loaded at connection time)	CSWDevice::GetConfigurationFilePath/ csSWDevice. GetConfigurationFilePath
To set the specific default setup file	Use the default configuration SW to save it for the first time with the right name and in the right folder.
To get the current setup file (last loaded file)	CSWDevice::GetSetupFileCurrent / csSWDevice.GetSetupFileCurrent

To set the current setup file	CSWDevice::SetSetupFileCurrent / csSWDevice.SetSetupFileCurrent

Configuration files are linked with the customized API, you will find more information about it (API) in the document “OEMCustomizedAPI”.

4.5.1.1 Callback function

For both text and binary formats it is possible to register a callback function that is called each time a file is saved or loaded. It can be used to edit the configuration data in the following cases:

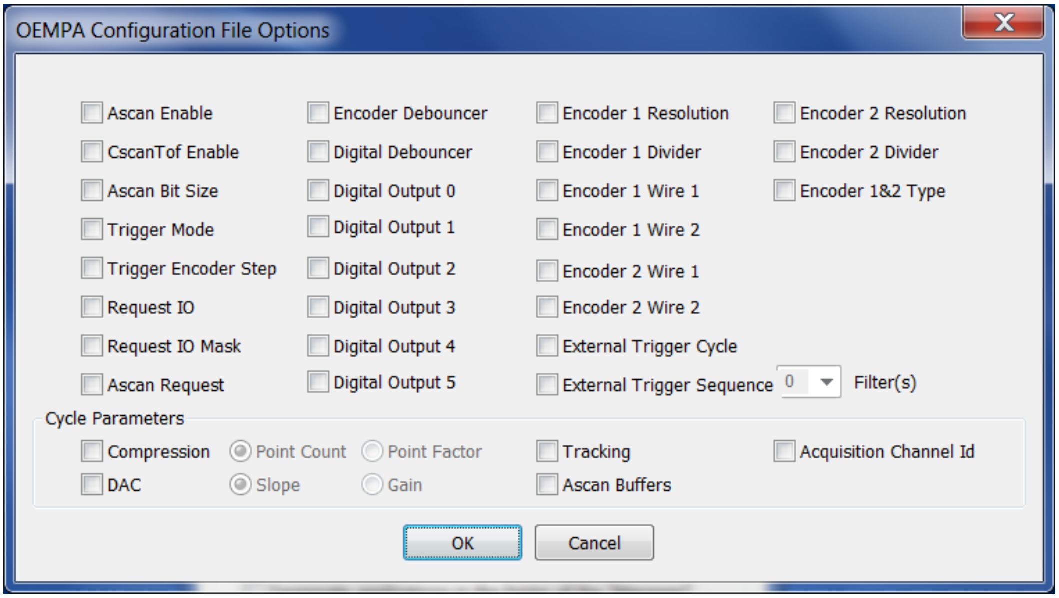
- Before saving a configuration file: to add a specific feature, for example to enable the “1/n A-scan” feature.
- Just after loading a configuration file (before writing the configuration data to the device): to add a specific feature, for example to add calibration correction for each probe element.

4.5.1.2 Optional parameters

With the text file format, some parameters have default values, so it is possible to hide them in the file. You can select which parameters are hidden in the text file format:

- Run the “DefaultConfiguration” SW for the default configuration file. Press the button “Option”. The “Filters” combo is enabled. Cycle parameters are not significant.
- Run the “Manager” SW for inspection configuration file. Press the button “Options”. The “Filters” combo is disabled. If you want to add filters in your inspection configuration you need to do it manually (copy/paste from the default configuration in which you can find filters).

Here is the dialog in the case of the “Manager” (version 1.1.5.0):



OEMPA Configuration File Options

□ Ascan Enable
□ CscanTof Enable
□ Ascan Bit Size
□ Trigger Mode
□ Trigger Encoder Step
□ Request IO
□ Request IO Mask
□ Ascan Request

□ Encoder Debouncer
□ Digital Debouncer
□ Digital Output 0
□ Digital Output 1
□ Digital Output 2
□ Digital Output 3
□ Digital Output 4
□ Digital Output 5

□ Encoder 1 Resolution
□ Encoder 1 Divider
□ Encoder 1 Wire 1
□ Encoder 1 Wire 2
□ Encoder 2 Wire 1
□ Encoder 2 Wire 2
□ External Trigger Cycle
□ External Trigger Sequence 0 Filter(s)

□ Encoder 2 Resolution
□ Encoder 2 Divider
□ Encoder 1&2 Type

Cycle Parameters

□ Compression ● Point Count ○ Point Factor □ Tracking □ Acquisition Channel Id
□ DAC ● Slope ○ Gain □ Ascan Buffers

Parameters in section “Root” of configuration files:

CheckBox	**Parameters**	**See paragraph 4.4 “Acquisition” sub paragraph**
“Ascan Enable”	“AscanEnable”	“Enable/Disable”
“Cscan Tof Enable”	“EnableCscanTof”	“Enable/Disable”
“Ascan Bit Size”	“AscanBitSize”	“A-scan” / “FW parameters”
“Trigger Mode”	“TriggerMode”	“Trigger”
“Trigger Encoder Step”	“TriggerEncoderStep”	“Trigger” / “Encoder”
“Request IO”	“RequestIO”	“Trigger”
“Request IO Mask”	“RequestIODigitalInputMaskRising” and “RequestIODigitalInputMaskFalling”	“Trigger”
“Ascan Request”	“AscanRequest”	“A-scan” / “FW parameters”
“Encoder Debouncer”	“DebouncerEncoder”	“Trigger” / “Encoder”
“Digital Debouncer”	“DebouncerDigital”	“Trigger” / “Digital inputs”
“Digital Output” 0 to 5	“DigitalOutputX”	“Trigger” / “Digital outputs”
“Encoder 1 Resolution”	“SWEncoder1Resolution”	“Trigger” / “Encoder”
“Encoder 1 Divider”	“SWEncoder1Divider”	“Trigger” / “Encoder”
“Encoder 1 Wire 1”	“Encoder1A”	“Trigger” / “Encoder”
“Encoder 1 Wire 2”	“Encoder1B”	“Trigger” / “Encoder”
“Encoder 2 Resolution”	“SWEncoder2Resolution”	“Trigger” / “Encoder”
“Encoder 2 Divider”	“SWEncoder2Divider”	“Trigger” / “Encoder”
“Encoder 2 Wire 1”	“Encoder2A”	“Trigger” / “Encoder”

“Encoder 1 Wire 2”	“Encoder2B”	“Trigger” / “Encoder”
“Encoder 1 & 2 Type”	“Encoder1Type” and “Encoder2Type”	“Trigger” / “Encoder”
“External Trigger Cycle”	“ExternalTriggerCycle”	“Trigger”
“External Trigger Sequence”	“ExternalTriggerSequence”	“Trigger”

Other parameters of configuration files:

CheckBox	**Parameters**	**File section**	**See paragraph 4.4 “**Acquisition**” sub paragraph**
“Tracking”	“TrackingStartEnable” “TrackingStartIndexCycle” “TrackingStartIndexGate” “TrackingStopEnable” “TrackingStopIndexCycle” “TrackingStopIndexGate”	“Cycle:X\Gate:Y”	“HW Gates” / “Echo Tracking API”
	“TrackingAscanEnable” “TrackingAscanIndexCycle” “TrackingAscanIndexGate”	“Cycle:X”	“A-scan” / “FW parameters”
	“TrackingDacEnable” “TrackingDacIndexCycle” “TrackingDacIndexGate”	“Cycle:X”	“A-scan” / “DAC”
“Compression” and radio button “Point Count”	“Compression” and “PointCount”	“Cycle:X”	“A-scan” / “FW parameters”
“Compression” and radio button “Point Factor”	“Compression” and “PointFactor”	“Cycle:X”	“A-scan” / “FW parameters”
“DAC” and radio button “Slope”	“DACTof” and “DACSlope”	“Cycle:X”	“A-scan” / “DAC”
“DAC” and radio button “Gain”	“DACTof” and “DACGain”	“Cycle:X”	“A-scan” / “DAC”

“Ascan Buffers”	“Maximum”, “Minimum” and “Saturation”	“Cycle:X”	“A-scan” / “FW parameters”
“Acquisition Channel Id”	“AcqIdChannelProbe”, “AcqIdChannelScan” and “AcqIdChannelCycle”	“Cycle:X”	“A-scan” / “FW parameters”
	“AcqIDAmp” “AcqIDTof”	“Cycle:X\Gate:Y”	“C-scan” / “FW parameters”

For example, if you checked the “Ascan Enable” and then save a configuration file, the “AscanEnable” parameter is saved with the default value:

```
[Root]
AscanEnable=1
```

If you save the configuration file with the value “AscanEnable=0”, you must also use the callback function (see the previous paragraph).

4.5.2 Specific device

The 128/128 and 256/256 systems are special devices for which two or four connections are used instead of a single one. It is only necessary to connect with the master unit. The driver will automatically connect to the slave device if the parameter “MatchedDevice” stored in the flash file of the master device is enabled.

You can specify the parameter “MatchedDevice” with the following function of the high level API:

Set (C++ / C#)	CSWDeviceOEMPA::EnableMatchedDeviceHwLink / csSWDeviceOEMPA.EnableMatchedDeviceHwLink
Get (C++ / C#)	CSWDeviceOEMPA::IsMatchedDeviceHwLinkEnabled / csSWDeviceOEMPA.IsMatchedDeviceHwLinkEnabled

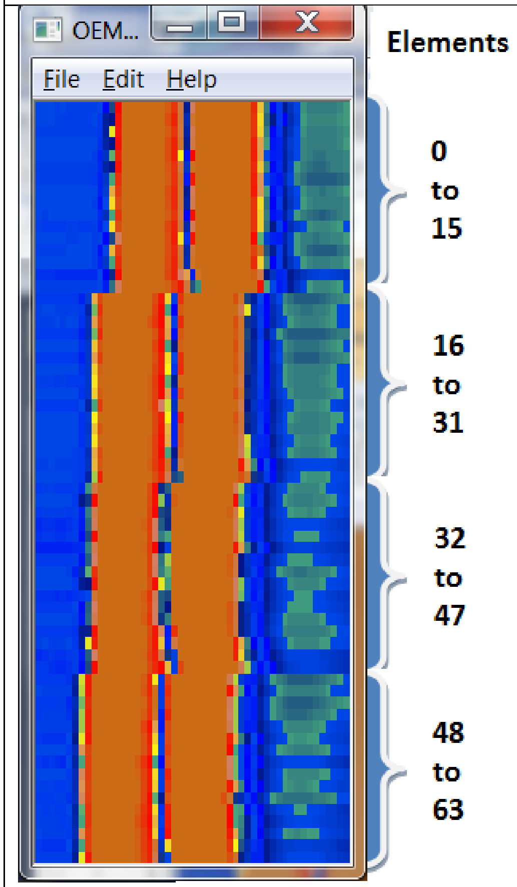
This software parameter is saved in the software configuration file. If changed, it is applied immediately; you don’t need to restart the driver.

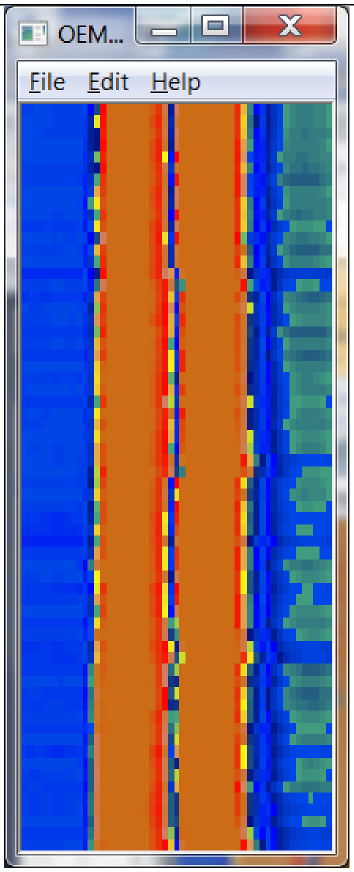
4.5.3 Calibration

The OEM-PA **must** be calibrated at startup (except for the OEM-PA 16/16 version, which does not require calibration). This calibration aligns the delay of all channels, so that when a delay value is set, the value represents the same amount of delay for each channel. This operation must be performed each time OEM-PA is turned ON. Once the calibration is complete, it is not necessary to calibrate again until the system is turned OFF.

As of version **1.1.5.4a** (and newer) all calibration parameters are saved in the device so that if you reconnect to the device after disconnecting, *but without restarting the device*, the calibration information will be ready to be applied to new setups.

Here is an example with an OEM-PA 64/64:

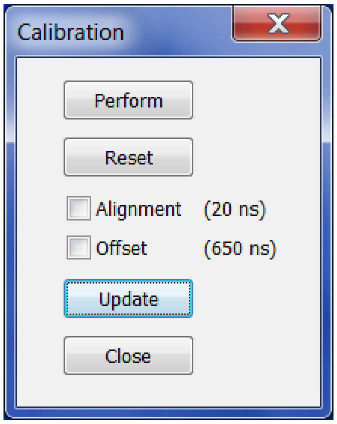
<table>
<tr><td>Before calibration</td><td>After calibration</td></tr>
<tr><td>


Shift is 45 ns (range is 0.5 us).
</td><td>


Alignment is corrected (but the time offset is not yet corrected).
</td></tr>
</table>

After calibration, all elements are aligned but there is still the same time offset for all elements. The last step of calibration can be used to correct this time offset. This second step is not supported by all FW (only FW Id 0x1200 and newer).

Calibration is available in version 1.1.5.0. The calibration is performed by a correction on delays when the OEMPA file is loaded. You need to enable it before loading the OEMPA file. The dialog "Calibration" in "OEMPATool" is an example of how to manage it:

Dialog	Comment
 Calibration Perform Reset Alignment (20 ns) Offset (650 ns) Update Close	Button "Perform": to run a new calibration. Button "Reset": to remove the calibration. CheckBox "Alignment": to correct the alignment of all elements. The correction is not applied immediately but when you load a setup (correction on delays). If this checkbox is unchecked no correction is applied to the loaded setup. CheckBox "Offset" (version 1.1.5.0n and newer): to correct the time offset. The time offset is sent to the HW, so the correction is applied directly. Button "Update" (version 1.1.5.0n and newer): After calibration is performed, you can press this button to reload the linear scan of all elements (calibration setup) and acquire all ascan to refresh the Bscan display.

Calibration feature	**Function (C++) / Function (C#)**
	CHW=CHWDeviceOEMPA,
	csHW=csHWDeviceOEMPA,
	CSW=CSWDeviceOEMPA,
	csSW=csSWDeviceOEMPA
		Comment

Perform	CHW::PerformCalibration / csHW.PerformCalibration No need to call "LockDevice/ UnlockDevice".	To perform a new calibration. It is better to remove the probe from the connector. This function returns the alignment delay (stored internally) and the time offset.
Enable	CSW::EnableAlignment / csSW.EnableAlignment	To enable/disable current alignment result.

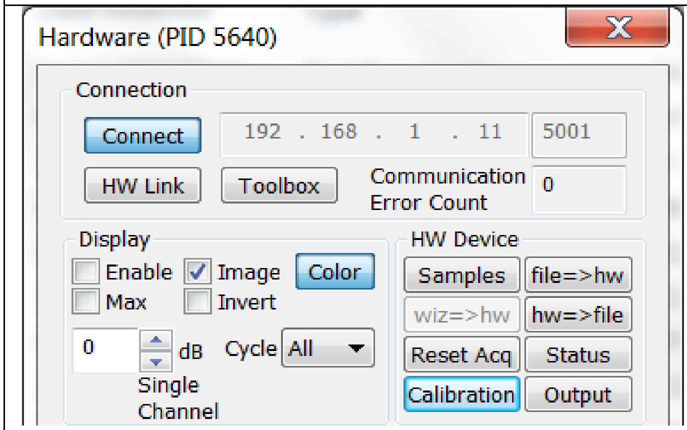
Reset	CSW::ResetAlignment / csSW.ResetAlignment	To reset alignment result.
Request calibration performed	CSW::IsCalibrationPerformed or CSW::IsAlignmentPerformed / csSW.IsCalibrationPerformed or csSW.IsAlignmentPerformed	To query if the calibration has been performed.
Request enabled	CSW::IsAlignmentEnabled / csSW.IsAlignmentEnabled	To query if the alignment is enabled.
Correction of the time offset.	CHW::SetTimeOffset / csHW.SetTimeOffset Call of “LockDevice/ UnlockDevice” is required only for this function.	Use “PerformCalibration” to get the time offset and then you can call “SetCalibrationOffset” to correct it.
Change default UT parameters.	CSW::SetCalibrationParameters / csSW.SetCalibrationParameters	UT parameters used by the calibration process, by default: pulse width is 100 ns, ascan start is 0 us, ascan range is 2.0 us, the analog gain is 15 dB and the digital gain is 20 dB.

It is better to remove the probe from the connector during calibration, but it is not absolutely required. In such cases, be careful of the calibration process: the first falling edge of the shot is used at 50 % to detect the delay of each element. With the probe connected, if this falling edge is not clean, you can change the UT parameter with function “SetCalibrationParameters”.

Calibration time is less than 1 second (except for OEM-PA 128/128 which is about 6 seconds).

Further explanation on use of the “Calibration” dialog (SW version 1.1.5.0n and HW version 0x1200):

Because the "Calibration" dialog is modal, before opening it check the checkbox "Image" of the dialog "HW" and change the palette with the right "Color" you want. Then you can press the button "Calibration" to display the dialog.



192 . 168 . 1 . 11 5001

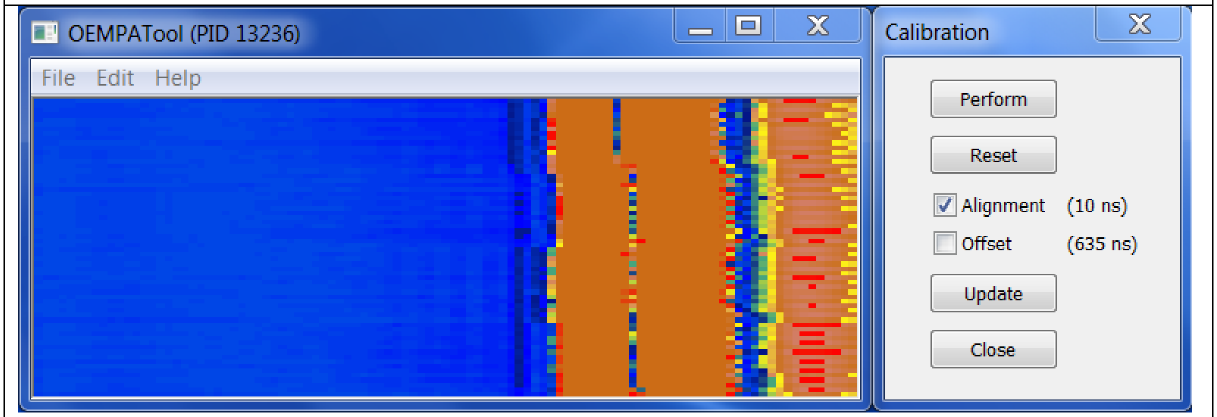
Communication Error Count 0

☐ Enable ☑ Image

☐ Max ☐ Invert

0 dB Single Channel Cycle All

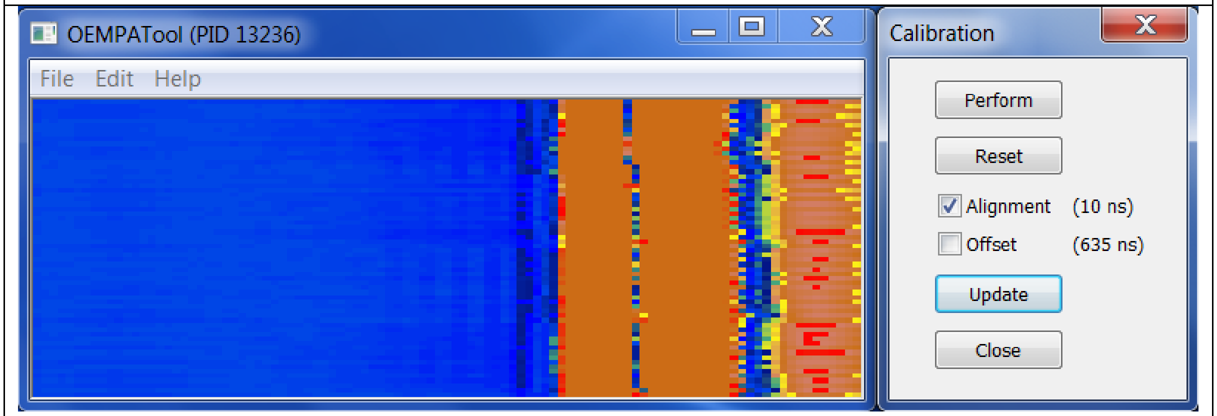
In the "Calibration" dialog press the button "Perform" - it is better to remove the probe from the connector during calibration. At the end the checkbox "Alignment" is automatically checked, but the display shows the B-scan for which no correction has been applied.



☑ Alignment (10 ns)

☐ Offset (635 ns)

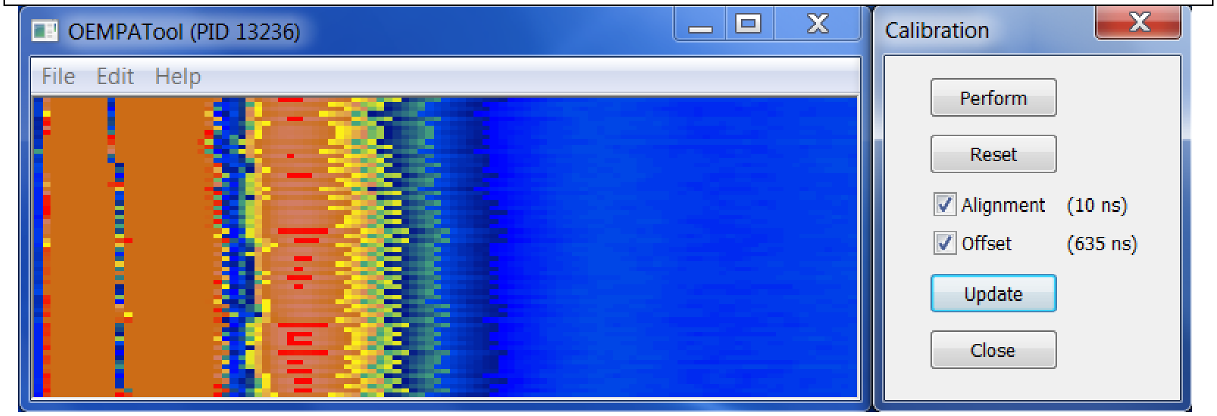
To display the corrected alignment, you need to press the button "Update" to reload the linear scan of all elements (calibration setup) and acquire all corrected A-scans ("Alignment" is checked).



☑ Alignment (10 ns)

☐ Offset (635 ns)

To correct the time offset, check the checkbox "Offset" and press the button "Update" again.



☑ Alignment (10 ns)
☑ Offset (635 ns)

4.5.4 Encoder management

If the default algorithm of the driver to manage encoders doesn't fit your needs, you can also manage them by yourself. For this purpose call the following function to disable the driver management:

Feature	Function C++ / Function C#
Enable/disable encoder management by the driver	CHWDevice::EnableDriverEncoderManagement / csHWDevice.EnableDriverEncoderManagement
To know if the driver is used to manage encoders.	CHWDevice::IsDriverEncoderManagementEnabled / csHWDevice.IsDriverEncoderManagementEnabled

When the driver is disabled, in order to manage encoders:

- You must register your own callback function to manage IO stream.
- The encoder input parameter of A-scan and C-scan callback functions are not significant. You have to compute the encoder yourself. Note that in each sub-stream (IO, A-scan, C-scan) you can find the time stamp (see paragraph 4.4.7.1 "Time Stamp"), and the cycle number, so you can link A-scan and C-scan with IO sub-streams.

To register an acquisition callback function for IO streams:

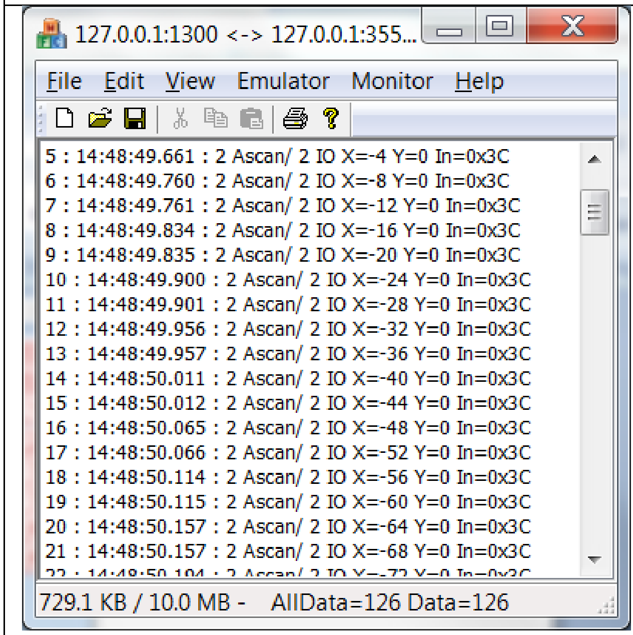
API CHWDevice::SetAcquisitionIO_0x00010101 (C++)
csHWDevice.SetAcquisitionIO_0x00010101 (C#)

Here are the differences when the driver is enabled or disabled to manage encoders:

Encoders are managed by the driver	Encoders are managed by the user (not by the driver)
An internal acquisition callback function is automatically registered and called on each new IO sub-stream.	The user must register an acquisition callback function to manage IO streams.
A-scan and C-scan acquisition callback function: Encoder input parameter is significant (the link with IO stream is already managed).	A-scan and C-scan acquisition callback function: Encoder input parameter is NOT significant. You have to compute it yourself.
IO stream internal acquisition callback function is called from a first loop. A second loop is used to call IO stream, A-scan and C-scan user acquisition callback functions (order is the same as the order of IO, A-scan and C-scan sub-streams in the stream sent by the FW).	IO stream acquisition callback function is called from a first loop, and a second loop is used to call A-scan and C-scan acquisition callback functions.

Example:

```
SetAcquisitionIO_0x00010101(CallbackIOStream)
SetAcquisitionAscan_0x00010103(CallbackAscanStream)
//setup with only 2 cycles (C-scan acquisition callback is managed the same as the A-scan callback)
```

Streams list	Stream (*)	sequence	cycle	TimeStamp
 127.0.0.1:1300 <-> 127.0.0.1:355... 5 : 14:48:49.661 : 2 Ascan/ 2 IO X=-4 Y=0 In=0x3C 6 : 14:48:49.760 : 2 Ascan/ 2 IO X=-8 Y=0 In=0x3C 7 : 14:48:49.761 : 2 Ascan/ 2 IO X=-12 Y=0 In=0x3C 8 : 14:48:49.834 : 2 Ascan/ 2 IO X=-16 Y=0 In=0x3C 9 : 14:48:49.835 : 2 Ascan/ 2 IO X=-20 Y=0 In=0x3C 10 : 14:48:49.900 : 2 Ascan/ 2 IO X=-24 Y=0 In=0x3C 11 : 14:48:49.901 : 2 Ascan/ 2 IO X=-28 Y=0 In=0x3C 12 : 14:48:49.956 : 2 Ascan/ 2 IO X=-32 Y=0 In=0x3C 13 : 14:48:49.957 : 2 Ascan/ 2 IO X=-36 Y=0 In=0x3C 14 : 14:48:50.011 : 2 Ascan/ 2 IO X=-40 Y=0 In=0x3C 15 : 14:48:50.012 : 2 Ascan/ 2 IO X=-44 Y=0 In=0x3C 16 : 14:48:50.065 : 2 Ascan/ 2 IO X=-48 Y=0 In=0x3C 17 : 14:48:50.066 : 2 Ascan/ 2 IO X=-52 Y=0 In=0x3C 18 : 14:48:50.114 : 2 Ascan/ 2 IO X=-56 Y=0 In=0x3C 19 : 14:48:50.115 : 2 Ascan/ 2 IO X=-60 Y=0 In=0x3C 20 : 14:48:50.157 : 2 Ascan/ 2 IO X=-64 Y=0 In=0x3C 21 : 14:48:50.157 : 2 Ascan/ 2 IO X=-68 Y=0 In=0x3C 22 : 14:48:50.104 : 2 Ascan/ 2 IO X=-72 Y=0 In=0x3C 729.1 KB / 10.0 MB - AllData=126 Data=126	5 IO-1	1	0	0x68875D03
	5 A-1	1	0	0x68875D28
	5 IO-2	1	1	0x68875EF7
	5 A-2	1	1	0x68875F1C
	6 IO-1	2	0	0x68883016
	6 A-1	2	0	0x6888303A
	6 IO-2	2	1	0x6888320A
	6 A-2	2	1	0x6888322E
	7 IO-1	3	0	0x6888E167
	7 A-1	3	0	0x6888E18B
	7 IO-2	3	1	0x6888E35B
	7 A-2	3	1	0x6888E37F
	…	…	…	…

(*) : "5 IO-1" means: stream 5, sub-stream IO number 1.

Here is the corresponding call stack of the driver:

Encoders are managed by the driver	Encoders are managed by the user (not by the driver)
CallbackIOStream(5 IO-1)	CallbackIOStream(5 IO-1)
CallbackAscanStream(5 A-1)	CallbackIOStream(5 IO-2)
CallbackIOStream(5 IO-2)	CallbackAscanStream(5 A-1)
CallbackAscanStream(5 A-2)	CallbackAscanStream(5 A-2)
CallbackIOStream(6 IO-1)	CallbackIOStream(6 IO-1)
CallbackAscanStream(6 A-1)	CallbackIOStream(6 IO-2)
CallbackIOStream(6 IO-2)	CallbackAscanStream(6 A-1)
CallbackAscanStream(6 A-2)	CallbackAscanStream(6 A-2)
CallbackIOStream(7 IO-1)	CallbackIOStream(7 IO-1)
CallbackAscanStream(7 A-1)	CallbackIOStream(7 IO-2)
CallbackIOStream(7 IO-2)	CallbackAscanStream(7 A-1)
CallbackAscanStream(7 A-2)	CallbackAscanStream(7 A-2)

5. FIFO API

5.1 Introduction

This section explains the FIFO (**F**irst **I**n, **F**irst **O**ut buffer) of the OEM-PA driver.

The first way to collect acquisition data is to register a callback function to which the driver sends any new data coming from the HW. However, any big delay in this callback can disturb the device and cause data loss. For example, if SW mutual exclusion is present in the callback function, it can delay the callback and possibly lead to data loss. In the case of managed code (C# or C++/CLI), it is possible that the garbage collector can also increase the processing time.

The FIFO is useful to insulate the driver main thread from the user thread and reduce the potential for data loss. It is available for all languages (C++, C#, MATLAB, LabView, etc…). There is one FIFO for each callback function.

In some cases (MATLAB or LabView) it is also convenient to use the FIFO as a method of temporarily storing acquisition data. In the case of the MATLAB stub only a minimum API is provided. The basic FIFO classes are “CAcquisitionFifo” for C++ and “csAcquisitionFifo” for C#. The API is the same for those 2 classes.

The FIFO is only implemented in the 64-bit version of the SDK. The feature is not available in the 32-bit version. The FIFO can also be used for OEM-MC and FMC applications. (See Section 6 and Section 7.)

5.2 C++/C# API

Even if the FIFO is enabled, the callback may or may not be registered by the user code:

- If the callback is registered, then an internal thread is automatically launched by the driver to read any data in the FIFO and to dispatch it to the callback function, so that the FIFO is transparent for the user.
- If the callback is not registered, then the user is responsible for reading the data in from the FIFO. A specific API is available for this purpose.

The FIFO has been allocated for a maximum cycle count. Each new cycle is saved in the FIFO with a new index. To briefly sketch how the FIFO works, the range of the data index is something like 0 to the maximum cycle count minus one. The tail index is the oldest data and the head index is the index of the next new data that will be saved in the FIFO. For each new data acquisition, the head index is automatically incremented. Those two indexes can be retrieved with the function “GetItemLimit“. The user is responsible to remove data in the FIFO. Each time the tail is removed the tail index is incremented. If the data is not removed then the head index will overwrite the tail index and some data will be lost.

Two different functions are also available to retrieve the data index from the specified cycle/sequence and just from the cycle:

- **L**ast **I**n, **F**irst **O**ut (LIFO) function ("GetLifoItem"): This function is better when retrieving the most recent data to minimize the delay between the data acquisition and its display. But because each time that the data is removed, the oldest data is also automatically removed, data loss can occur.
- Standard **F**irst **I**n, **F**irst **O**ut (FIFO) function ("GetFifoItem"): This function is better to ensure there is no data loss.

The entry function to get the FIFO pointer is the following:

C++	C#
"CHWDevice::GetAcquisitionFifo"	"csHWDevice.GetAcquisitionFifo"

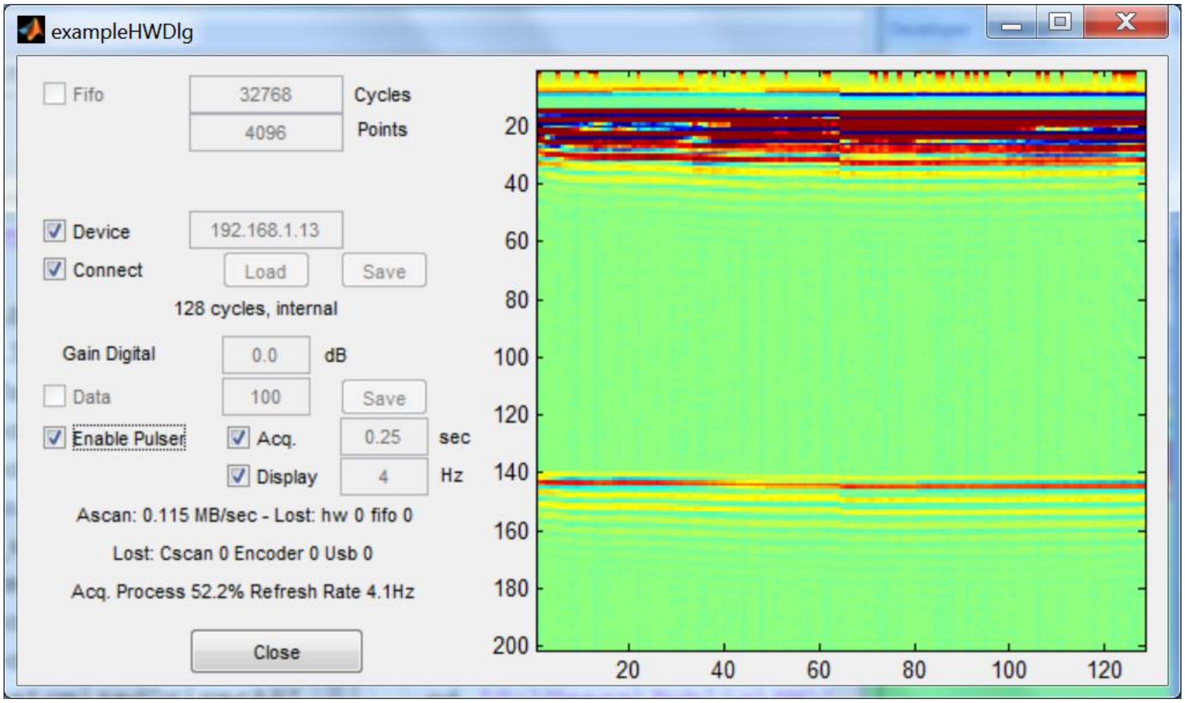
The full FIFO API is the following:

Member function (C++/C#)	Comment
Alloc	Enable the FIFO. Two inputs parameters: the data count and the size of the buffer to store all of them (data is A-scan, C-scan or IO depending of the FIFO type).
Desalloc	Disable the FIFO.
GetAlloc	Read back options of allocation.
GetCount	Count of current saved data.
GetLost	If data is read from the FIFO is slower than new data is written then some HW data will be lost once the buffer fills.
GetTotalCount	Total count of data that have been saved in the FIFO.
GetTotalByteCount	Total byte count of data that have been saved in the FIFO. This count takes into account the header of the data.
ResetCounters	Call this function to reset the different counters (see "GetCount", "GetLost", "GetTotalCount" and "GetTotalByteCount".
RemoveAll	To remove all current data in the FIFO. The device should be stopped otherwise the behavior is not safe.
RemoveTail	Remove the oldest data in the FIFO.
RemoveItem	Remove specific data in the FIFO. Each data saved in the FIFO is identified by an index named "Item".

GetFifoItem	Returns the data item for the specified cycle and sequence or just for the specified cycle. Output value -2 means missing item (the item valid range is positive). The search is performed from the oldest data to the newest (First In First Out).
GetLifoItem	Returns the data item for the specified cycle and sequence or just for the specified cycle. Output value -2 means missing item (the item valid range is positive). The search is performed from the newest data to the oldest (Last In First Out).
GetItemLimit	Get the range for the "Item" indexes. Two indexes are returned: "iIndexTail" and "iIndexHead". "iIndexTail" is the index of the oldest data and "iIndexHead" is the index of next data that will be stored in the FIFO. Any item is in the range 0 to CountMax-1 (CountMax is the maximum data count that has been defined with the function "Alloc").
IncrementItemIndex	This function can be used to browse the different data in the FIFO. That is, to increment the Item from the tail (oldest data) to the head (newest data). Here is an idea of the process with 2 examples (This is just an idea. The real processing is slightly different): Example 1: CountMax=128, iItem=10; IncrementItemIndex(iItem) => iItem=11 Example 2: CountMax=128, iItem=127; IncrementItemIndex(iItem) => iItem=0
DecrementItemIndex	This function can be used to browse the different data in the FIFO. That is, to decrement the Item from the head (newest data) to the tail (oldest data).
InFifo	This function is used by the driver to add new HW acquisition data in the FIFO.
OutAscan	To get an A-scan. The A-scan can be removed from the FIFO, or not removed so that it can be read another time. Should be called only for the A-scan FIFO.
OutCscan	To get a C-scan. The C-scan can be removed from the FIFO, or not removed so that it can be read another time. Should be called only for the C-scan FIFO.
OutIO	To get an IO. The IO can be removed from the FIFO, or not removed so that it can be read another time. Should be called only for the IO FIFO.

5.3 Matlab example

The best example of the FIFO API is the MATLAB script “ExampleHWDlg.m” included with SDK version “1.1.5.3w” and later. The FIFO can be enabled or disabled and in case it is enabled, FIFO (“std”) and LIFO (“Lifo”) outputs are available:



exampleHWDlg

□ Fifo 32768 Cycles

4096 Points

☑ Device 192.168.1.13

☑ Connect

128 cycles, internal

Gain Digital 0.0 dB

□ Data 100

☑ Enable Pulser ☑ Acq. 0.25 sec

☑ Display 4 Hz

Ascan: 0.115 MB/sec - Lost: hw 0 fifo 0

Lost: Cscan 0 Encoder 0 Usb 0

Acq. Process 52.2% Refresh Rate 4.1Hz

Addition details are provided in the *Matlab_Stub.pdf* document

6. Multiple Channel (OEM-MC)

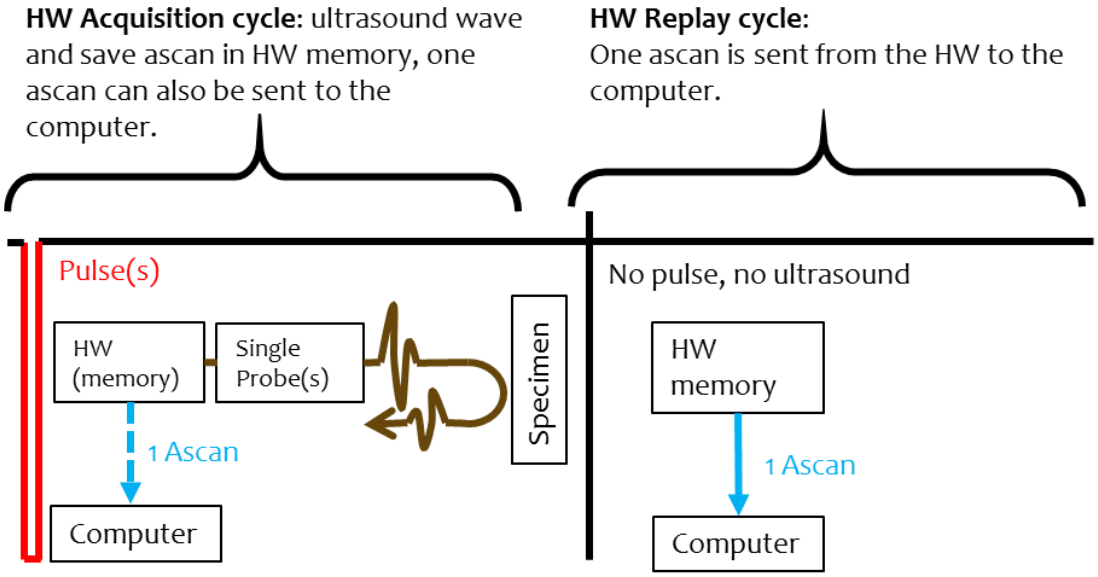
6.1 OEM-MC Cycles

The OEM-MC and OEM-PA share many of the same API commands. However the concept and meaning of cycles and the Time Slot can differ between the two devices depending on how the OEM-MC is used.

For an OEM-PA device, a cycle is the process of sending a pulse to the transducer to generate an ultrasound wave in the specimen and then to receive the A-scan in the device and send it to the computer within the allotted Time Slot (see paragraph 4.4.1, “Terminology”). All these processes are performed during the same cycle.

For an OEM-MC device, there are two different types of cycles:

- **Ultrasound cycle**: this cycle is required to pulse an ultrasound wave (one or many) in the specimen and to save A-scan in the OEMMC device memory. This is the first cycle of one sequence. One A-scan can also be sent to the computer.
  From May 2016 (driver version 1.1.5.3r and later), this cycle is referred to as a “**HWAcquisition cycle**”. *Previous documentation referred to this as an Acquisition Cycle.*
- **Data acquisition cycle**: each time you want to send one A-scan from the device to the computer one cycle should be defined. No ultrasound wave is propagated. This type of cycle is only used in Parallel Mode. (See paragraph 6.3.2 “Parallel Mode”.)
  From May 2016 (driver version 1.1.5.3r and later), this cycle is referred to as a “**HWReplay cycle**”. *Previous documentation referred to this as a Processing Cycle.*


HW Acquisition cycle: ultrasound wave and save ascan in HW memory, one ascan can also be sent to the computer.
HW Replay cycle: One ascan is sent from the HW to the computer.
Pulse(s)
HW (memory)
Single Probe(s)
Specimen
1 Ascan
Computer
No pulse, no ultrasound
HW memory
1 Ascan
Computer

6.2 OEM-PA / OEM-MC Decimation

When speaking about the OEM-MC, there are two types of decimation. The first type is the same as described in section 4.4.5 “A-scan” in the “FW parameters” sub-section for the OEM-PA. We will refer to it as OEM-PA compression. OEM-PA compression is the parameter “CompressionType” in section “Cycle:X” in the OEMPA or OEMMC file, it could be compression (maximum data of N) or decimation (the first data among N). The degree of compression is specified by the “PointCount” or “PointFactor” parameters.

The other type of decimation only applies to the OEM-MC. It is therefore referred to as OEM-MC decimation (or *input decimation*). It is the parameter “Decimation” in section “Cycle:X\Receiver” of an OEMMC file and it is only decimation (1 data every N). **OEMMC decimation is an optional feature which is not available by default.**

In the next section (6.3 Data Acquisition Modes) unless explicitly stated, the document is referring to OEM-MC decimation.

Usually when the OEM-MC decimation is enabled, OEM-PA compression is not used.

6.3 Data Acquisition Modes

There are two different modes for OEM-MC device:

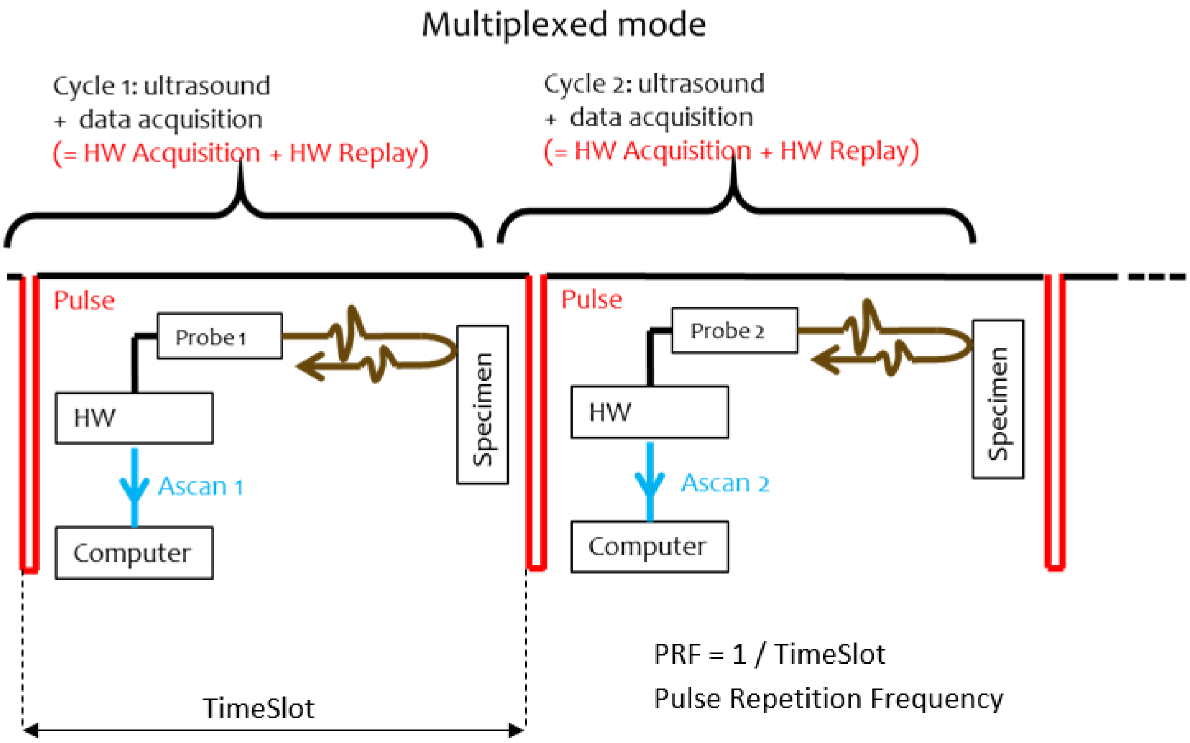
- Multiplexed mode
  - Only one pulse is triggered for each cycle.
  - HW memory is not used. Data is processed and returned each cycle, similar to an OEM-PA.
  - OEM-MC decimation is not available in this mode.
- Parallel mode
  - Pulses are triggered for all probes at the same cycle.
  - HW memory is used to store A-scans. Data is returned over the first HW Acquisition cycle and some number of HWReplay cycles.
  - OEM-MC decimation may be applied to HWReplay cycles.

In both modes it is possible to work in Pulse-Echo (the same probe transmits and receives) or in Pitch-Catch (the receiving probe is different from the transmitting probe).

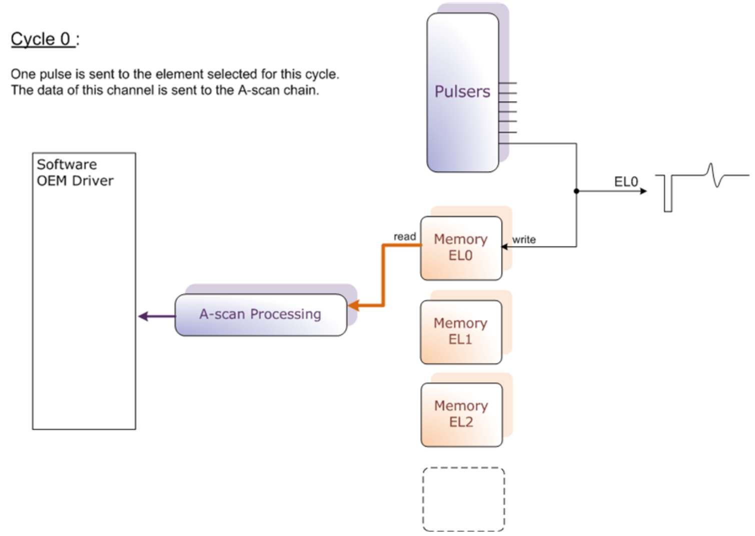
6.3.1 Multiplexed Mode

Multiplexed mode works in the following way:

- The A-scan is acquired and sent to the computer in the same cycle. The HW memory is not used, there are no limitation with A-scan range and echo tracking.
- OEM-PA compression can be used to improve the throughput (this parameter is the same as the one used with OEM-PA, parameter "CompressionType" in section "Cycle:X").

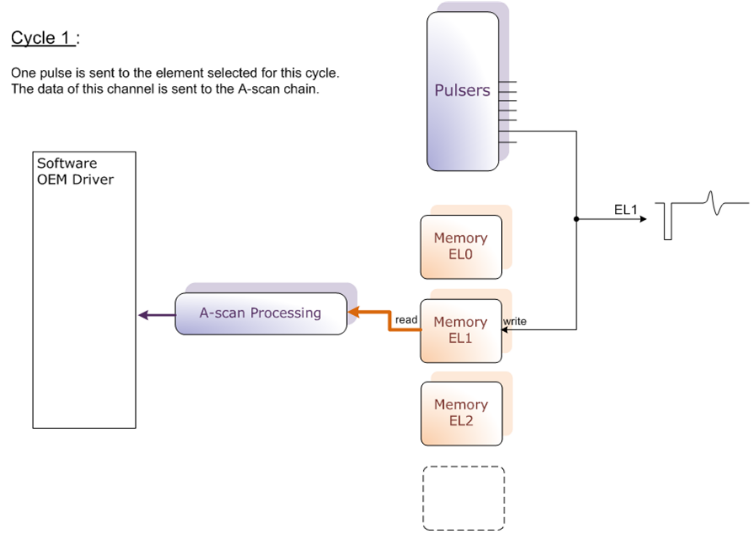


On the following page there is a block diagram of the processing for Multiplexed Mode:



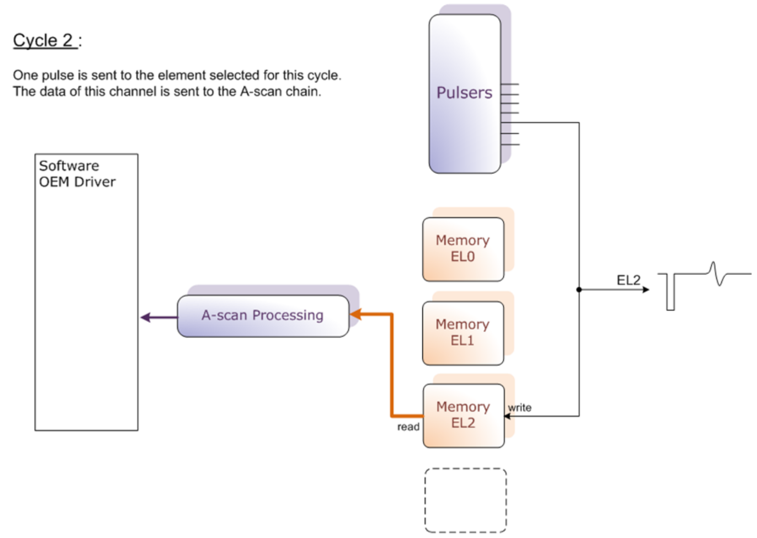
Cycle 0 :

One pulse is sent to the element selected for this cycle.
The data of this channel is sent to the A-scan chain.



Cycle 1 :

One pulse is sent to the element selected for this cycle.
The data of this channel is sent to the A-scan chain.



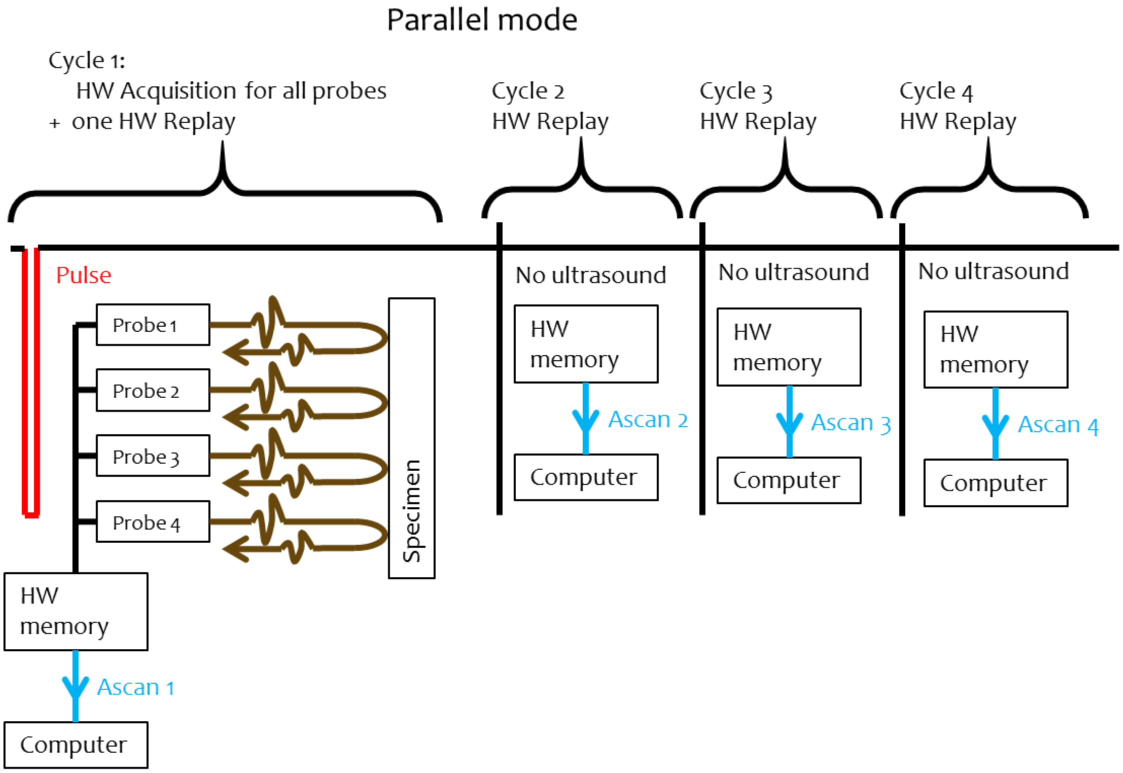
Cycle 2 :

One pulse is sent to the element selected for this cycle.
The data of this channel is sent to the A-scan chain.

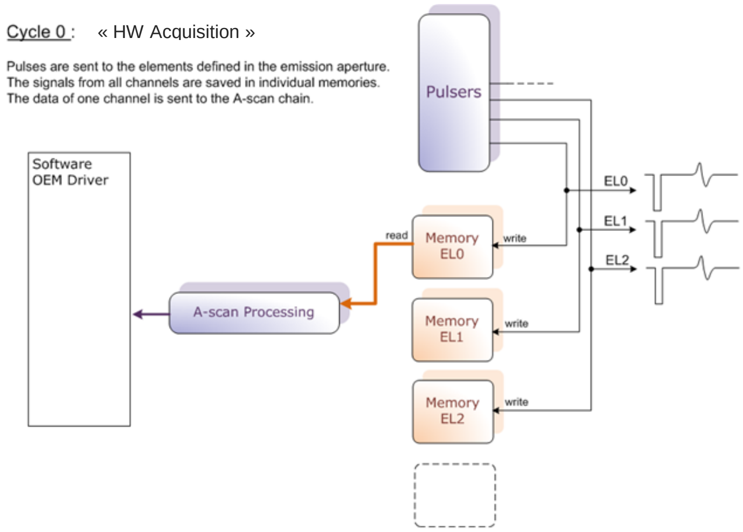
6.3.2 Parallel Mode

Parallel mode works in the following way:

- Ascans are saved in HW memory during the ultrasound cycle. This is why there are some limitations on Ascan range, echo tracking and filters.
- Data throughput is better than with the multiplexed mode especially if the input decimation is enabled for low frequency probe.
- In parallel mode, Time Slot ≠ 1/PRF. In fact, the parameter TimeSlot may be replaced by **HWAcquisitionTime** and **HWReplayTime** for HWAcquisition and HWReplay cycles respectively to distinguish between the timing requirements for the two different kinds of cycles.
- There is no Analog Gain for HW Replay cycles.

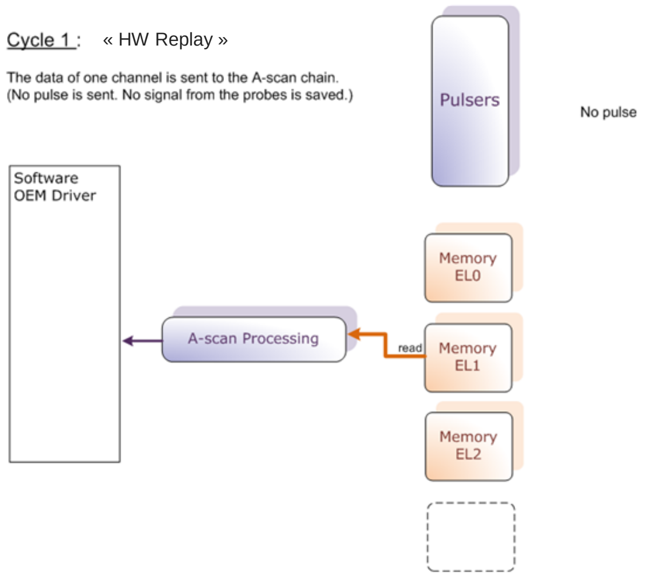


The block diagram on the next page shows how the processing is handled in parallel mode. (*In previous versions of the documentation, this diagram appeared with Cycle 0 as an Acquisition Cycle and Cycles 2 and 3 as Processing Cycles. It has been updated to reflect the new terminology "HW Acquisition Cycle" and "HW Replay Cycle".*)



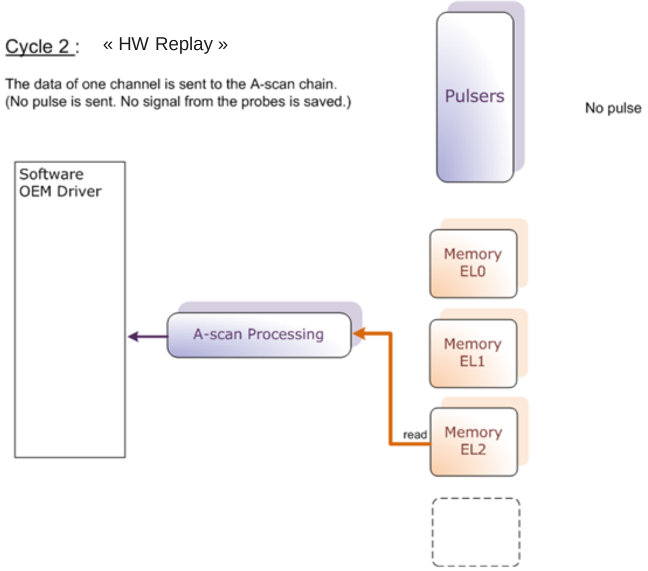
Cycle 0 : « HW Acquisition »

Pulses are sent to the elements defined in the emission aperture.
The signals from all channels are saved in individual memories.
The data of one channel is sent to the A-scan chain.



Cycle 1 : « HW Replay »

The data of one channel is sent to the A-scan chain.
(No pulse is sent. No signal from the probes is saved.)



Cycle 2 : « HW Replay »

The data of one channel is sent to the A-scan chain.
(No pulse is sent. No signal from the probes is saved.)

In parallel mode, when the HW memory is used to store the A-scans from all the probes at the same time, data may be written with different clock frequencies depnding on each cycle's input decimation factor, but all data are read with the 100 MHz clock. Writing is performed during the HW Acquisition cycle, reading is performed during the HW Replay cycle. The default "write" clock is 100 MHz when decimation is disabled. The maximum Ascan range written to HW memory depends on the input decimation factor:

Input Decimation	0	1	2	…	15
Digitizing frequency (MHz)	100	50	33.333	…	6.25
Max range (µs)	204.8	409.6	614.4	…	3276.8

Note: Digitizing frequency is the "write" clock in the HW memory.

Important: Input decimation is an optional feature which is not available by default. If you would like to have access to it, please consult AOS.

Because the reading clock frequency is always 100 MHz:

- If input decimation is enabled, then the A-scan processing time in the FW is quicker than multiplexed mode.
- Filters are usually defined for a digitizing frequency of 100 MHz. If decimation is enabled, then the real digitizing frequency is different. Be careful that the any filter used is designed for the actual digitizing frequency.

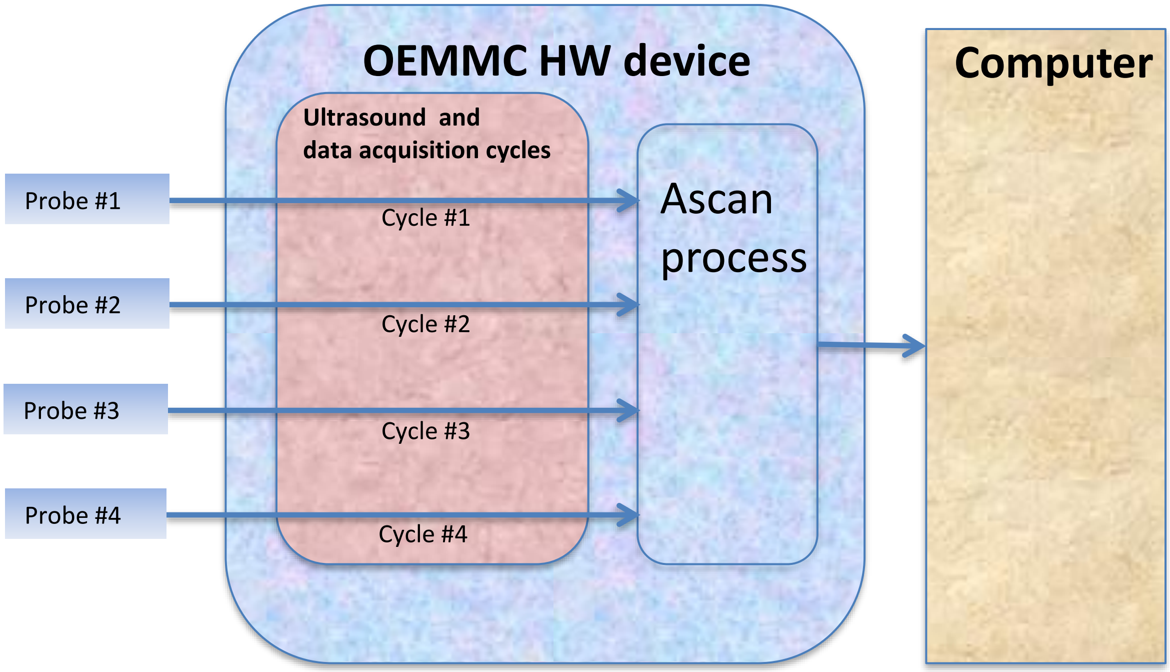
Input decimation for the first probe, which returns data during the HW Acquisition cycle, should be disabled. (The value of Input Decimenation should be 0.)

The benefits and disadvantages of using parallel mode to take advantage of the HW memory and input decimation are summarized below:

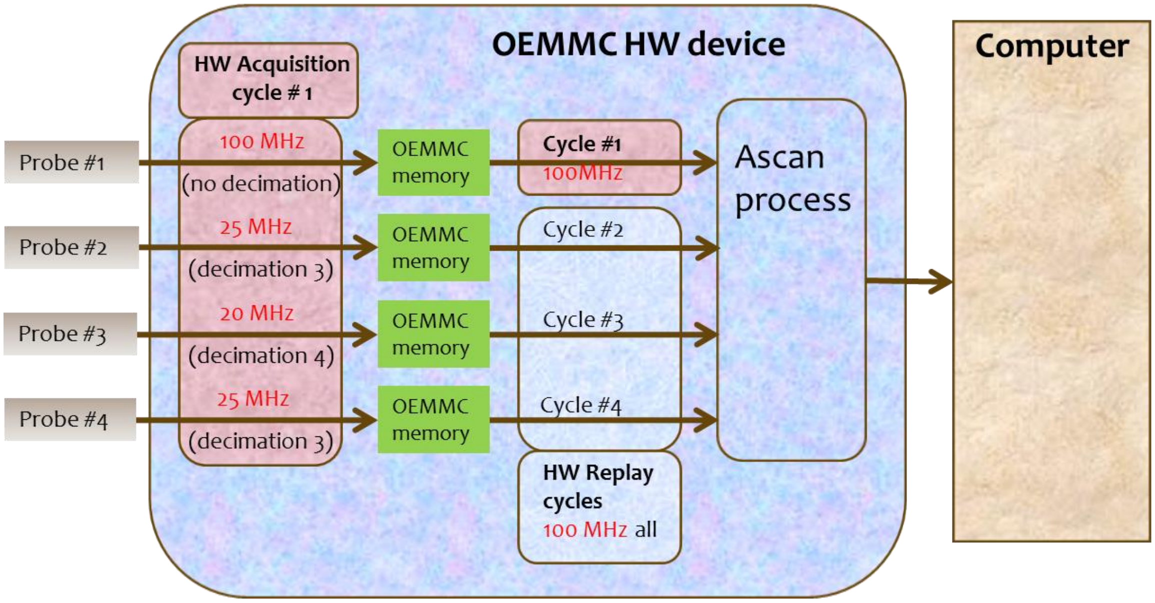
Benefits	Disadvantages
- Water path or wedge delay time is 0 µs for HW Replay cycles. The water path time is required only for the HW Acquisition cycle. (see paragraph 4.4.8.2, "Wedge delay") - Decimation decreases the A-scan point count, thus the quantity of data sent to the computer is lower: it is possible to reduce the Time Slot or to increase the A-scan range.	- Filters should be redesigned. - Echo tracking should be used with care. (See section 4.4.8, "HW Gates")

6.3.3 Summary

In multiplexed mode, the OEM-MC behaves similarly to an OEM-PA and returns an A-scan with each cycle as shown below.

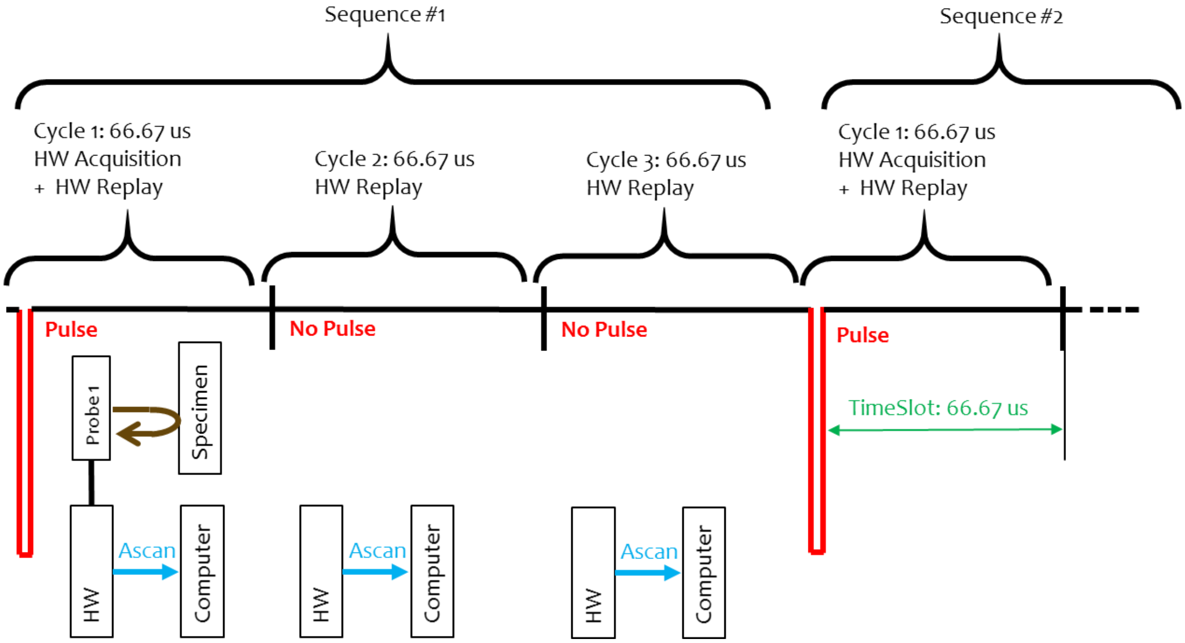


In parallel mode, the OEM-MC can pulse all probes at once. A-scans will be returned one at a time during HW Replay cycles. By taking advantage of decimation and the fact that a wedge delay need not be factored into each cycle, the speed with which all data is returned can be increased.



As mentioned before the PRF ≠ 1/Time Slot in parallel mode. The following simple example where the time slot is 66.67 µs for each cycle shows how PRF is determined in parallel mode.

In driver versions 1.1.5.3r and later, the duration of the HW Acquisition cycle is called **HWAcquisitionTime**, and the duration of the HW Replay cycles is called **HWReplayTime**.



The PRF is 1 / (3 x 66.67 µs) = 1 / 200 µs, that is 5,000 Hz. By using decimation and ignoring water path for cycles 2 and 3, HWReplayTime could be reduced, thus the PRF could be increased.

6.3.4 Multiplexed vs. Parallel Example

It can be helpful to look at a concrete example to demonstrate the difference in overall processing time for multiplexed and parallel modes. Consider the following:

- Water path of 40 mm (54 µs at 1480 m/s).
- Specimen thickness of 100 mm (35 µs at 5700 m/s).
  - The default digitizing frequency is 100 MHz, so the A-scan will have 3501 points.
- Two pulse/echo probes: 2 MHz and 1 MHz.
  - Input decimation can be used to select a digitizing frequency of 20 MHz for the 2 MHz probe (700 points) and 10 MHz for the 1 MHz probe (350 points).

For the **multiplexed** mode (input decimation disabled), the different processing times are:

- Water path: 54 µs.
- FW Ascan processing: 3501 x 10 ns = 35 µs.
- Transfer time (at 10 MB/sec): 3501 / (10MB/s) = 333 µs

  - This time could be decreased by using the OEM-PA compression.
  - But the purpose of this paragraph is only to explain the OEM-MC type decimation.

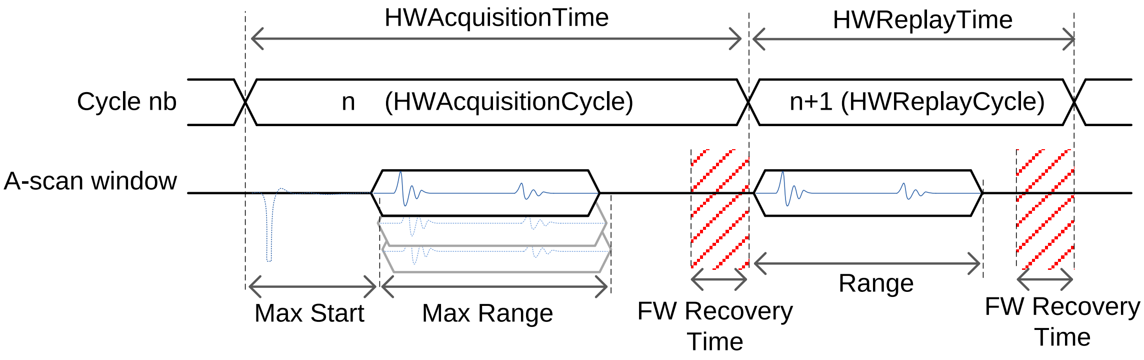
For the **parallel** mode (input decimation enabled except for the first element), the different processing times are:

- HW Acquisition cycle: processing times are the same as for the multiplexed mode.
- HW Replay cycle:
  - Water path: 0 µs (ultrasound is not used).
  - FW Ascan processing: 350 x 10 ns = 3.5 µs (data acquired with the digitizing frequency of 10 MHz are processed with the clock 100 MHz).
  - Transfer time: 350 / (10MB/s) = 33 µs.

So the processing time of the parallel mode is quicker except for the first cycle. The lower the probe frequency, the more there is to gain from OEMMC decimation.

6.4 FW Recovery Time

The formula that defines the "FW Recovery Time" limitation is different for HWAcquisition cycles and HWReplay Cycles.



6.4.1 HW Acquisition Cycle

This cycle must be longer than the longest channel that is defined in the HW Replay Cycles. The formula do not depends only on the parameters of the current Acquisition cycle, but also on the parameters (start and range) of the Replay cycles associated.

(The "Start" of the replay channels are defined in the HWAcquisitionCycle. The "Range" of the replay channels are defined inside each HWReplayCycle.)

$$\boxed{HWAcquisitionTime \geq Max\ Start + Max\ Range + FWRecoveryTime} \quad \text{(MC.a)}$$

6.4.2 HW Replay Cycle

Since the data has been saved from the "Start" value during the HWAcquisitionCycle, the data is read and processed directly from the needed points in HWReplay cycles. So the "Start" does not influence the limitation of the FWRecoveryTime.

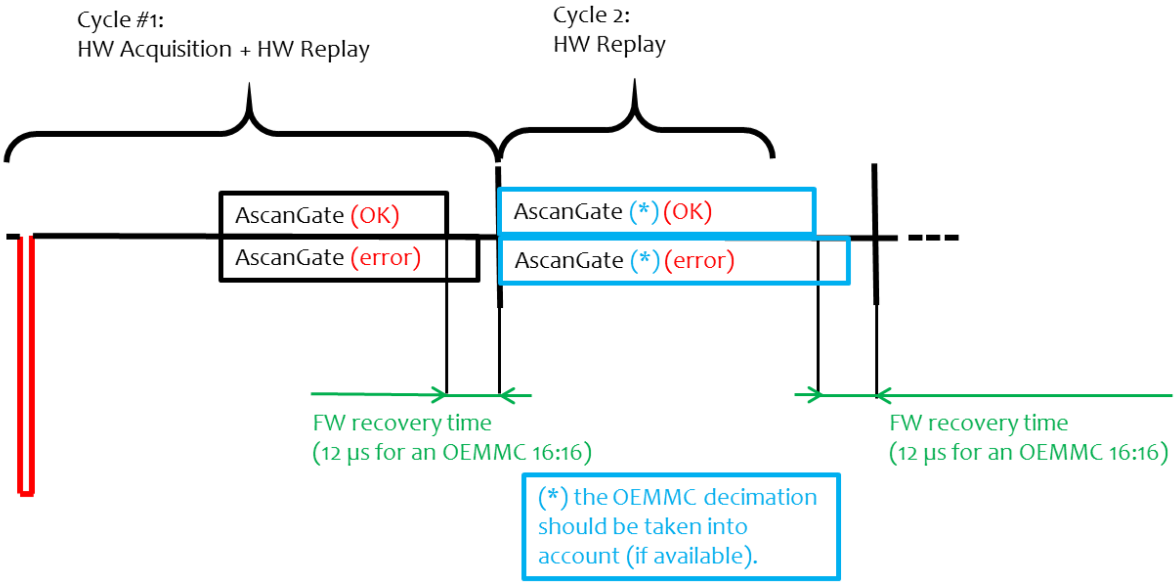
$$\boxed{HWReplayTime \geq Range + FWRecoveryTime} \qquad \text{(MC.b)}$$

The value of "FWRecoveryTime" is the same in both kinds of cycles. It can vary depending on the FW version. Basically, it is either 12.5us or 3.5us. A function of the API allows the developer to know the value. (See "GetFWAscanRecoveryTime" in paragraph 4.4.10 "FW Recovery Time".)

Because the “read” clock is always 100 MHz, “Range” for HWReplay cycles can be shorter than “Max Range” in HWAcquisition cycles if the input decimation for the cycle is nonzero.

6.4.3 Minimum Time Slot

For an OEM-MC, the minimum value for TimeSlot/HWAcquisitionTime/HWReplayTime is 15 µs. With such a low value “FWRecoveryTime” becomes a very important parameter. If the A-scan gate (Max Start + Max Range for HWAcquisition, Range for HWReplay) does not allow enough time for the FWRecoveryTime, then errors will occur.


Cycle #1:
HW Acquisition + HW Replay
Cycle 2:
HW Replay
AscanGate (OK)
AscanGate (error)
AscanGate (*) (OK)
AscanGate (*) (error)
FW recovery time
(12 µs for an OEMMC 16:16)
FW recovery time
(12 µs for an OEMMC 16:16)
(*) the OEMMC decimation should be taken into account (if available).

6.5 API

The API for the OEM-MC is divided between four groups of functions (sections in the OEMMC file):

- Basic (root)
- Cycle
- Pulser
- Receiver

Some commands are the same as they are for the OEM-PA,

6.5.1 Basic

To check that your HW supports the Multiple-Channel feature, call the following function:

API (SW): “CSWDeviceOEMPA::**IsMultiHWChannelSupported**” (C++) /
“csSWDeviceOEMPA.**IsMultiHWChannelSupported**” (C#) /

By default the Multiple-Channel feature is enabled. To know if this is the case, call the following function:

API (SW): “CSWDeviceOEMPA::**IsMultiHWChannelEnabled**” (C++) /
“csSWDeviceOEMPA.**IsMultiHWChannelEnabled**” (C#) /

If disabled just after calling “LockDevice”, call the following function to enable the Multiple Channel feature (input value “true”):

API (HW): “CHWDeviceOEMPA::**EnableMultiHWChannel**” (C++) /
“csHWDeviceOEMPA.**EnableMultiHWChannel**” (C#) /
Section “Root” key “**EnableMultiChannel**” (OEMMC file)*

For OEM-MC devices, **EnableMultiChannel=1** ***must appear in the root section****. This differentiates between OEMMC files and OEMPA files. By default the setting for OEMPA files is used. That is, the default value for EnableMultiChannel is 0.*

The following static function returns the maximum A-scan range (in seconds) saved in the HW memory during the acquisition cycle:

API (HW): “CHWDeviceOEMPA::**GetMultiHWChannelRangeMax**” (C++) /
“csHWDeviceOEMPA.**GetMultiHWChannelRangeMax**” (C#)

6.5.2 Cycle

The API for the OEM-MC must be able to differentiate between the two kinds of cycles used. To differentiate between a HWAcquisition and a HWReplay cycle the following API call is used:

To select the cycle type ("acquisition" or "Processing") use the following function:

API: “CHWDeviceOEMPA::**EnableMultiHWChannelAcquisition**” (C++) /
“csHWDeviceOEMPA.**EnableMultiHWChannelAcquisition**” (C#) /
Section “Cycle:X” key “**MultiChannelAcquisition**” (OEMMC file **prior to 1.1.5.3r**)
Section “Cycle:X” key “**HWAcquisition**” (OEMMC file **after to 1..1.5.3r**)

From driver version 1.1.5.3r and later, the parameter “MultiChannelAcquisition” in the OEMMC files has been renamed “HWAcquisition” to be more descriptive in distinguishing between HWReplay cycles and HWAcquisition Cycles. However, “MultiChannelAcquisition” may still be used after 1.1.5.3r if preferred.

Here is how the the parameter should appear in the “Cycle:X” section to distinguish between the two types of cycles.

Cycle type	**HW Acquisition**	**HW Replay** **(Parallel mode Only)**

Value of HWAcquisition /
MultiChannelAcquisition
	HWAcquisition = 1
	MultiChannelAcquisition = 1
		HWAcquisition = 0
		MultiChannelAcquisition = 0

The remaining cycle parameters are the same for HWAcquisition and HWReplay cycles:

API: “CHWDeviceOEMPA::**SetGainDigital**” (C++) /
“csHWDeviceOEMPA.**SetGainDigital** ” (C#) /
Section “Cycle:X” key “**GainDigital**” (OEMMC file)

API: “CHWDeviceOEMPA::**SetGainAnalog**” (C++) /
“csHWDeviceOEMPA.**SetGainAnalog** ” (C#) /
Section “Cycle:X” key “**GainAnalog**” (OEMMC file)*****
**Only necessary for HW Acquisition cycles. Not used for HW Replay Cycles.*

API: “CHWDeviceOEMPA::**SetAscanRange**” (C++) /
“csHWDeviceOEMPA.**SetAscanRange**” (C#) /
Section “Cycle:X” key “**Range**” (OEMMC file)
(“**SetAscanRangeWithCount**” / “**SetAscanWithFactor**” / “**Count**” are also available)

API: “CHWDeviceOEMPA::**SetTimeSlot**” (C++) /
“csHWDeviceOEMPA.**SetTimeSlot**” (C#) /
Section “Cycle:X” key “**TimeSlot**” (OEMMC file prior to 1.1.5.3r)
Section “Cycle:X” key “**HWAcquisitionTime/HWReplayTime”** (OEMMC file after 1.1.5.3r)

From driver version 1.1.5.3r and later, the parameter “TimeSlot” in the OEMMC files has been renamed “HWAcquisitionTime” or “HWReplayTime” to distinguish between types of cycles. However, TimeSlot may still be used after 1.1.5.3r if preferred.

API: “CHWDeviceOEMPA::**SetAscanRectification**” (C++) /
“csHWDeviceOEMPA.**SetAscanRectification**” (C#) /
Section “Cycle:X” key “**Rectification**” (OEMMC file)

API: “CHWDeviceOEMPA::**SetFilterIndex**” (C++) /
“csHWDeviceOEMPA.**SetFilterIndex**” (C#) /
Section “Cycle:X” key “**FilterIndex**” (OEMMC file)

API: “CHWDeviceOEMPA::**SetBeamCorrection**” (C++) /
“csHWDeviceOEMPA.**SetBeamCorrection**” (C#) /
Section “Cycle:X” key “**BeamCorrection**” (OEMMC file)
This gain is added to the “GainDigital” inside the HW.

Note there is no Start parameter in the Cycle section of an OEMMC file.

You can find a more complete list of available cycle parameters in the document *Software_Functions_&_Parameter_List.pdf*.

6.5.3 Pulser

You don’t need to specify pulser parameters of a HWReplay cycle. For a HWAcquisition cycle you have to specify the following two parameters:

To define the aperture:
API: “CHWDeviceOEMPA::**SetEmissionEnable**” (C++) /
“csHWDeviceOEMPA.**SetEmissionEnable** ” (C#) /
Section “Cycle:X\Pulser” key “**Element**” (OEMMC file)

To define the pulse width:
API: “CHWDeviceOEMPA::**SetEmissionWidths**” (C++) /
“csHWDeviceOEMPA.**SetEmissionWidths**” (C#) /
Section “Cycle:X\Pulser” key “**Width**” (OEMMC file)

The C++ aperture is defined by the following input parameter (one bit for each single element):

```
“DWORD adwHWAperture[4]”
  adwHWAperture[0] & 0x1 => 1 if the element #1 is enabled
  adwHWAperture[0] & 0x2 => 1 if the element #2 is enabled
  adwHWAperture[0] & 0x4 => 1 if the element #3 is enabled
  …
  adwHWAperture[1] & 0x1 => 1 if the element #33 is enabled
  adwHWAperture[2] & 0x2 => 1 if the element #34 is enabled
  …
Example: DWORD adwHWAperture[4]={0x00000f0f,0,0,0};
  It means that elements 1 to 4 and 9 to 12 are enabled.
  The syntax in OEMMC file is the following:
            Element.count=8
            Element=0;1;2;3;8;9;10;11
```

“SetEmissionWidths” should be called with the same input parameter as that of the function “SetEmissionEnable”.

Example:

OEMMC file	**Code (C++)**
[Cycle:0\Pulser]	DWORD adwAperture[4]={3,0,0,0};
Element.count=2	float afWidth[2]={0.5e-6,0.5e-6};
Element=0;1	
Width.count=2	pOEMPA->SetEmissionEnable(0, adwAperture);
Width=0.5;0.5 us	pOEMPA->SetEmissionWidths(0, adwAperture, afWidth);

In **multiplexed** mode, the Pulser section is mandatory for all cycles. Element.count and Width.count must both be 1 and only a single value will be entered for Element and Width.

However, in **parallel** mode, the Pulser section is only needed for the HW Acquisition cycle. For HW Replay cycles, the Pulser section can be omitted. In the HW Acquisition cycle, Element.count = Width.count = length of the Element and Width arrays.

6.5.4 Receiver

6.5.4.1 HWAcquisition cycle

In a HWAcquisition Cycle, you must define the A-scan start of all receivers - not only the A-scan start for the current cycle, but for all following HWReplay cycles. The start aperture is defined by consecutive elements from “StartFirst” (no gap is possible). The function is defined as the following:

API: “CHWDeviceOEMPA::**SetMultiHWChannelAcqWriteStart**” (C++) /
“csHWDeviceOEMPA.**SetMultiHWChannelAcqWriteStart**” (C#) /
Section “Cycle:X\Receiver” keys “**Start**”, “**StartFirst**”, and “**Element**” (OEMMC file)

Example 1:

OEMMC file	**Code (C++)**

[Cycle:0\Receiver]
Element.Count=1
Element=0
StartFirst=0
Start.count=4
Start=0.0;0.1;0.2;0.3 us
	float fStart[4]={0.0,0.1e-6,0.2e-6,0.3e-6};
	pOEMPA->SetMultiHWChannelAcqWriteStart(0,0,4, fStart)

Example 2:

OEMMC file	**Code (C++)**

[Cycle:0\Receiver]
Element.count=1
Element=1
StartFirst=1
Start.count=1
Start=50.0 us
 	
---OR---
 	
StartFirst=0
Start.count=2
Start=0.0;50.0 us
	int iStartCount=2;
	float pfStart[2]={0.0,50.0e-6};
	int iAcqElement=1;
	int iCycle=0;
	pOEMPA->SetMultiHWChannelAcqWriteStart(iCycle,iAcqElement,
	iStartCount, pfStart);
	
	// These lines are equivalent to the last 3 lines above.
	// They correspond to the same C++ code above. Using StartFirst is
	// equivalent to adding 0’s to the beginning of the Start array.

If OEM-MC decimation is enabled, you must specify the input decimation for the current cycle and all HW Replay Cycles. The input decimation for the first cycle should always be 0.

API: “CHWDeviceOEMPA::**SetMultiHWChannelAcqDecimation**” (C++) /
“csHWDeviceOEMPA.**SetMultiHWChannelAcqDecimation**” (C#) /
Section “Cycle:X\Receiver” key “**Decimation**” (OEMMC file)

Example:

OEMMC file	Comment

Start.count=3
Start=0.0;0.0;0.0 us
Decimation=0;3;3
	HWAcquisition cycle 100 MHz, HW Replay Cycles will be 25 MHz

Input decimation is an optional feature which is not available by default. If you would like to have access to it or would like information on how to use the C++ or C# API functions, please consult AOS.

6.5.4.2 HWAcquisition and HWReplay cycle

The element for which the acquisition data is sent to the computer is specified in the same way as the aperture of the pulsers. But instead of specifying more than one element, only one element could be defined in the aperture. The function is defined as the following:

API: “CHWDeviceOEMPA::**SetReceptionEnable**” (C++) /
“csHWDeviceOEMPA.**SetReceptionEnable**” (C#) /
Section “Cycle:X\Receiver” key “**Element**” (OEMMC file)

Example:

OEMMC file	Code (C++)

[Cycle:0\Receiver]
Element.count=1
Element=0
	DWORD adwAperture[4]={1,0,0,0};
	pOEMPA->SetReceptionEnable(0, adwAperture);

6.6 OEMMC File Examples

A few examples of OEMMC are presented here. Places were the keys have changed from previous SDK version are noted with the old keys used in (red parentheses) following the new key. The actual OEMMC file would not contain these lines.

Note: The old keys will still be handled correctly by the Customized (Medium Level) API so that OEMMC files created before 1.1.5.3r will still load without error.

6.6.1 Multiplexed Mode Example

Here is one example of a setting file that uses the OEM-MC in "multiplexed" mode. This setting is done to acquire the 3 first single-elements, one after the other.

OEMMC file	Comments

[Root]	
VersionDriverOEMPA=1.1.5.3	
CycleCount=3	3 cycles are defined.
EnableMultiChannel=1	Required for OEM-MC.
[Cycle:0]	
GainDigital=0.000000 dB	
Range=20.000000 us	
HWAcquisitionTime=2000.000 us	
(TimeSlot = 2000.0us)	For versions prior to 1.1.5.3r
PointCount=0	
Rectification=Signed	
FilterIndex=1	
GainAnalog=20.000000 dB	
GateCount=0	
BeamCorrection=0.000000 dB	
HWAcquisition=1	The first cycle is a HWAcquisition Cycle
(MultiChannelAcquisition=1)	For versions prior to 1.1.5.3r
[Cycle:0\Pulser]	
Element.count=1	
Element=0	Element #1
Width.count=1	
Width=0.1 us	Element #1
[Cycle:0\Receiver]	
Element.count=1	
Element=0	Element #1 is returned to the computer
StartFirst=0	
Start.count=1	
Start=0.0 us	Element #1
[Cycle:1]	
GainDigital=0.000000 dB	
Range=20.000000 us	
HWAcquisitionTime=2000.0 us	
(TimeSlot=2000.000000 us)	For versions prior to 1.1.5.3r

PointCount=0	
Rectification=Signed	
FilterIndex=1	
GainAnalog=20.000000 dB	
GateCount=0	
BeamCorrection=0.000000 dB	
HWAcquisition=1	The second cycle is also an HWAcquisition cycle
(MultiChannelAcquisition=1)	For versions prior to 1.1.5.3r
[Cycle:1\Pulser]	
Element.count=1	
Element=1	Element #2
Width.count=1	
Width=0.1 us	Element #2
[Cycle:1\Receiver]	
Element.count=1	
Element=1	Element #2 is returned to the computer
StartFirst=1	
Start.count=1	
Start=0.0 us	Element #2
[Cycle:2]	
GainDigital=0.000000 dB	
Range=20.000000 us	
HWAcquisitionTime=2000.0 us	
(TimeSlot=2000.000000 us)	For versions prior to 1.1.5.3r
PointCount=0	
Rectification=Signed	
FilterIndex=1	
GainAnalog=20.000000 dB	
GateCount=0	
BeamCorrection=0.000000 dB	
HWAcquisition=1	The third cycle is also a HWAcquisition cycle
(MultiChannelAcquisition=1)	For versions prior to 1.1.5.3r

[Cycle:2\Pulser]	
Element.count=1	
Element=2	Element #3
Width.count=1	
Width=0.1 us	Element #3
[Cycle:2\Receiver]	
Element.count=1	
Element=2	Element #3 is returned to the computer
StartFirst=2	
Start.count=1	
Start=0.0 us	Element #3

6.6.2 Parallel Mode Examples

The following examples demonstrate Multi-Channel acquisition in Parallel Mode.

6.6.2.1 Pulse/Echo with 2 Probes

OEMMC file	**Comments**
[Root]	
VersionDriverOEMPA=1.1.5.3	
CycleCount=2	2 cycles are defined.
EnableMultiChannel=1	Required for OEMMC.
[Cycle:0]	
GainDigital=0.000000 dB	
Range=30.000000 us	
HWAcquisitionTime=2000.0 us	
(TimeSlot=2000.000000 us)	For versions prior to 1.1.5.3r
PointCount=0	
Rectification=Signed	
FilterIndex=1	
GainAnalog=20.000000 dB	GainAnalog not needed in HWReplay cycles
GateCount=0	
BeamCorrection=0.0 dB	
HWAcquisition=1	The first cycle is a HWAcquisition cycle
(MultiChannelAcquisition=1)	For versions prior to 1.1.5.3r
[Cycle:0\Pulser]	

Element.count=2	
Element=0;1	Element #1 and #2
Width.count=2	
Width=0.5;0.5 us	Element #1 and #2
[Cycle:0\Receiver]	
Element.count=1	
Element=0	Element #1 is returned to the computer
StartFirst=0	
Start.count=2	
Start=0.0;0.1 us	Element #1 and #2
Decimation=0;1	Decimation disabled on Element #1; 50 MHz on #2
[Cycle:1]	
GainDigital=20.000000 dB	
Range=30.000000 us	
HWReplayTime=2000.0us	
(TimeSlot=2000.000000 us)	For versions prior to 1.1.5.3r
PointCount=0	
Rectification=Signed	
FilterIndex=1	
GateCount=0	
BeamCorrection=0.0 dB	
HWAcquisition=0	The second cycle is a HW Replay cycle
(MultiChannelAcquisition=0)	For versions prior to 1.1.5.3r
[Cycle:1\Receiver]	
Element.count=1	
Element=1	Element #2 is returned to the computer

6.6.2.2 Two single element (pitch/catch)

OEMMC file	**Comments**
[Root]	
VersionDriverOEMPA=1.1.5.3	
CycleCount=2	2 cycles are defined.
EnableMultiChannel=1	Required for OEMMC.

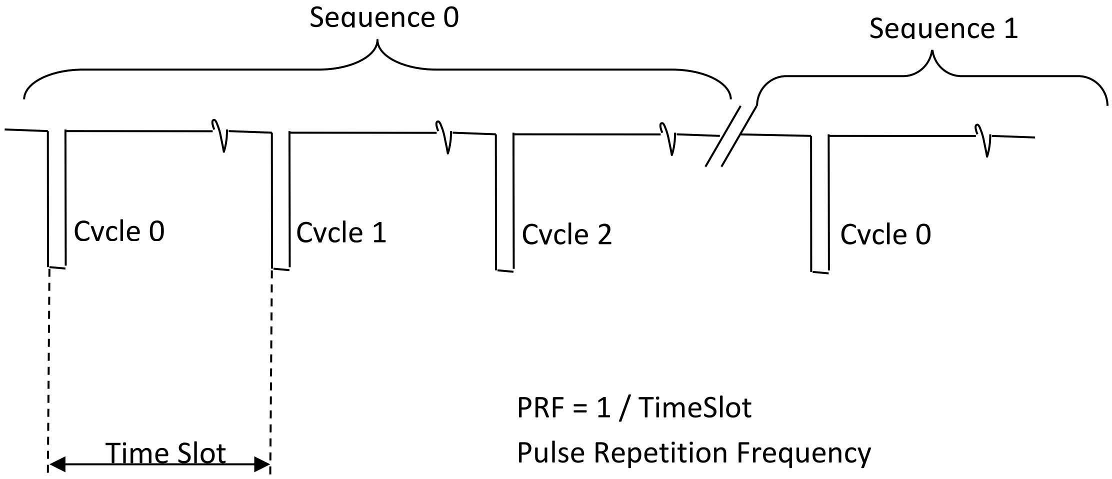
[Cycle:0]	
GainDigital=0.000000 dB	
Range=30.000000 us	
HWAcquisitionTime=2000.0us	
(TimeSlot=2000.000000 us)	For versions prior to 1.1.5.3r
PointCount=0	
Rectification=Signed	
FilterIndex=1	
GainAnalog=20.000000 dB	
GateCount=0	
BeamCorrection=0.0 dB	
HWAcquisition=1	The first cycle is a HW Acquisition cycle
(MultiChannelAcquisition=1)	For versions prior to 1.1.5.3r
[Cycle:0\Pulser]	
Element.count=1	
Element=0	Element #1
Width.count=1	
Width=0.5 us	Element #1
[Cycle:0\Receiver]	
Element.count=1	
Element=0	Element #1 is returned to the computer
StartFirst=0	
Start.count=2	
Start=0.0;0.1 us	Element #1 and #2
Decimation=0;0	Decimation disabled for both elements (This line could be omitted – only necessary if Decimation is enabled.)
[Cycle:1]	
GainDigital=20.000000 dB	
Range=30.000000 us	
HWReplayTime=2000.0us	
(TimeSlot=2000.000000 us)	For versions prior to 1.1.5.3r
PointCount=0	
Rectification=Signed	
FilterIndex=1	

GateCount=0	
BeamCorrection=0.0 dB	The second cycle is a HW Replay cycle
HWAcquisition=0	For versions prior to 1.1.5.3r
(MultiChannelAcquisition=0)	
[Cycle:1\Receiver]	
Element.count=1	Element #2 is returned to the computer
Element=1	

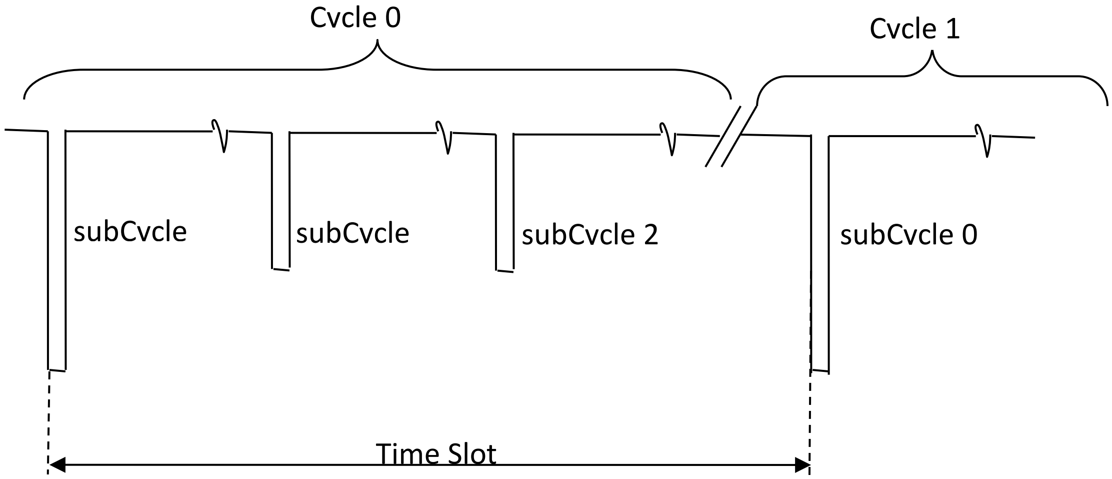
7. Full Matrix Capture (FMC)

FMC is managed by cycles and sub-cycles. For each cycle there is one pulse, and many data-streams are sent within sub-cycles.

A set of cycles is a sequence.



A set of subcycles is a cycle.

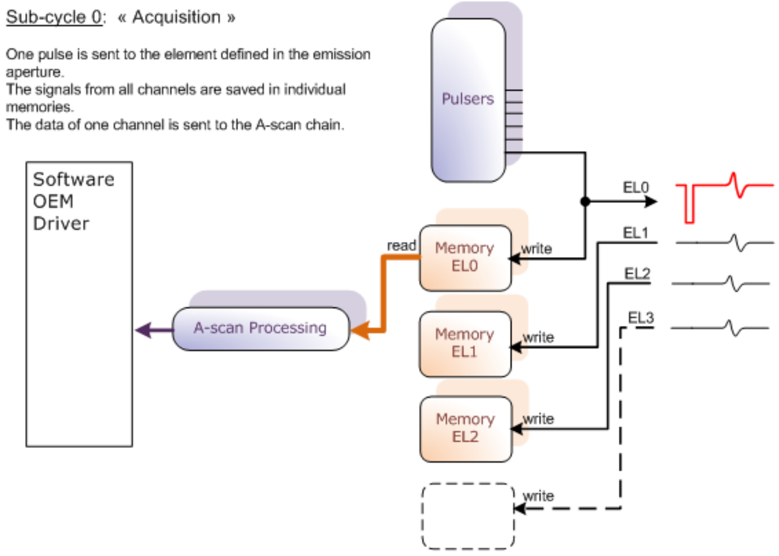


The following is a graphical representation of this process.

Cycle 0

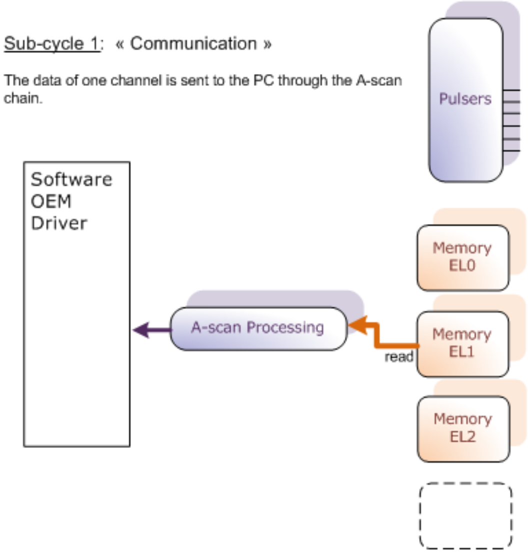
Sub-cycle 0: « Acquisition »

One pulse is sent to the element defined in the emission aperture.
The signals from all channels are saved in individual memories.
The data of one channel is sent to the A-scan chain.



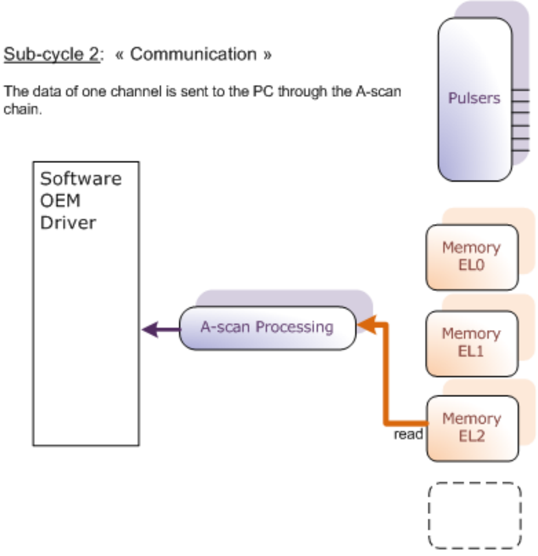
Sub-cycle 1: « Communication »

The data of one channel is sent to the PC through the A-scan chain.



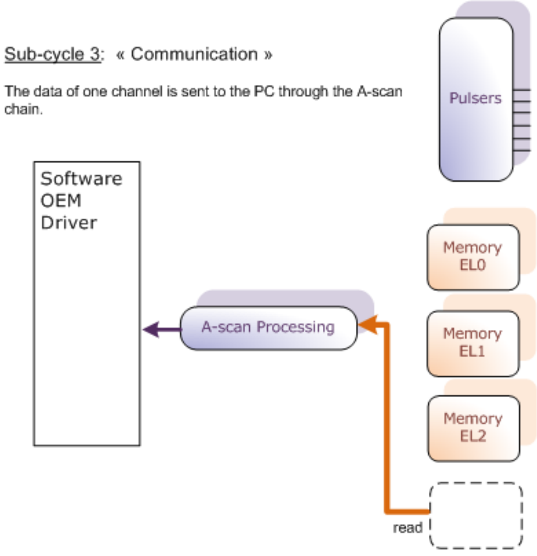
Sub-cycle 2: « Communication »

The data of one channel is sent to the PC through the A-scan chain.



Sub-cycle 3: « Communication »

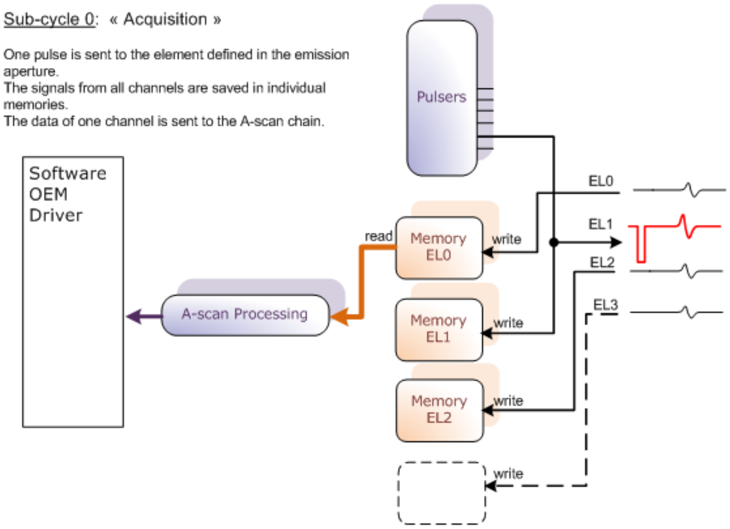
The data of one channel is sent to the PC through the A-scan chain.



Cycle 1

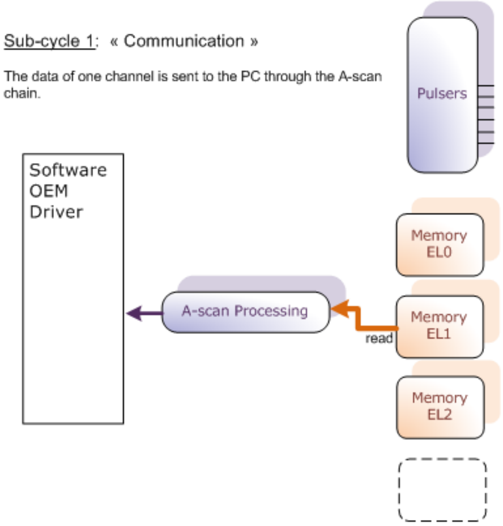
Sub-cycle 0: « Acquisition »

One pulse is sent to the element defined in the emission aperture.
The signals from all channels are saved in individual memories.
The data of one channel is sent to the A-scan chain.



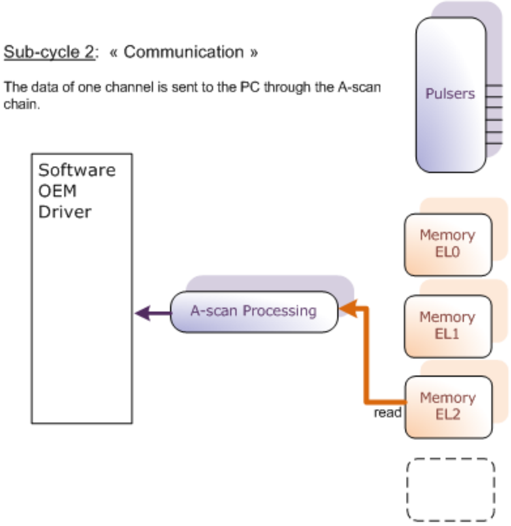
Sub-cycle 1: « Communication »

The data of one channel is sent to the PC through the A-scan chain.



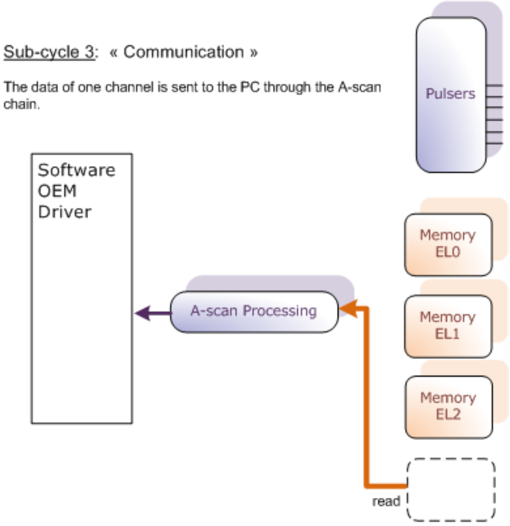
Sub-cycle 2: « Communication »

The data of one channel is sent to the PC through the A-scan chain.



Sub-cycle 3: « Communication »

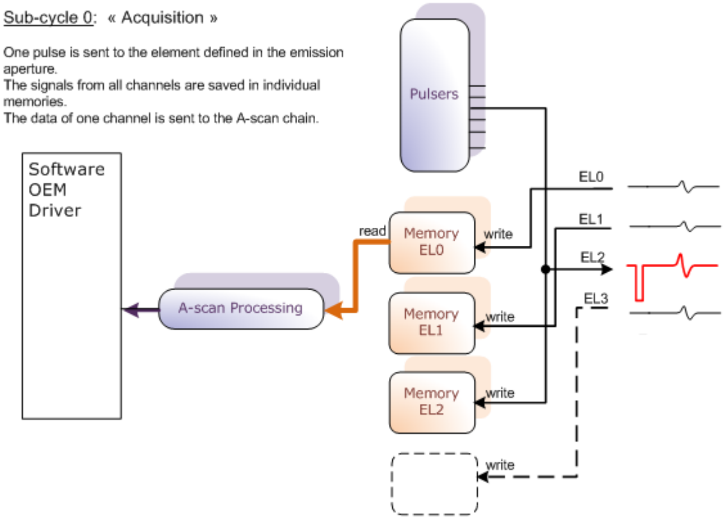
The data of one channel is sent to the PC through the A-scan chain.



Cycle 2

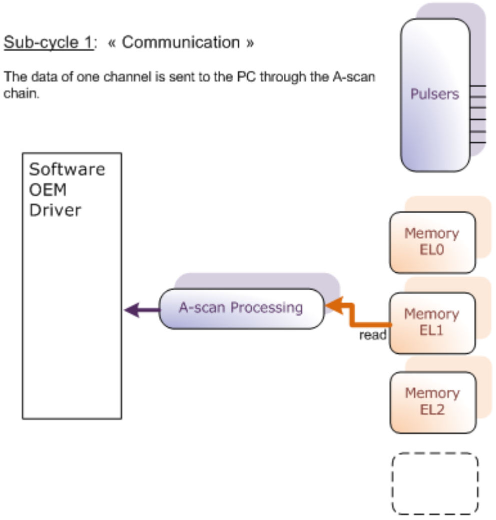
Sub-cycle 0: « Acquisition »

One pulse is sent to the element defined in the emission aperture.
The signals from all channels are saved in individual memories.
The data of one channel is sent to the A-scan chain.



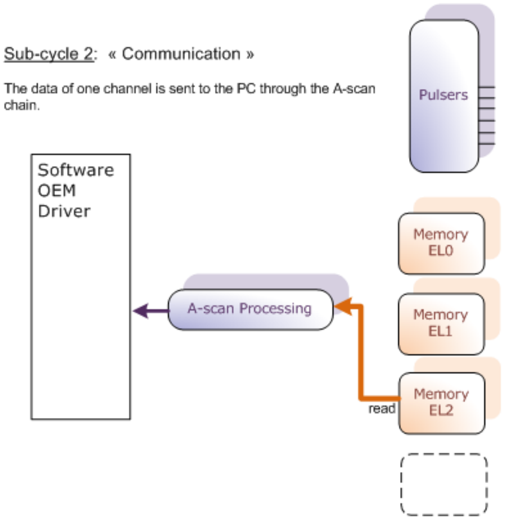
Sub-cycle 1: « Communication »

The data of one channel is sent to the PC through the A-scan chain.



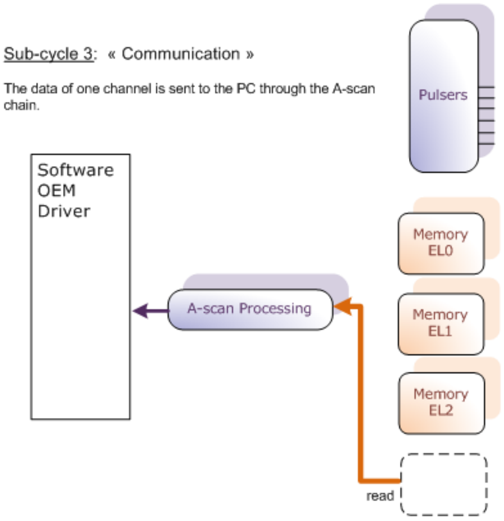
Sub-cycle 2: « Communication »

The data of one channel is sent to the PC through the A-scan chain.



Sub-cycle 3: « Communication »

The data of one channel is sent to the PC through the A-scan chain.



7.1 Basic

The following function can be used to query whether the HW supports the FMC feature:

API (SW): “CSWDeviceOEMPA::**IsFullMatrixCaptureReadWrite**” (C++) /
“csSWDeviceOEMPA.**IsFullMatrixCaptureReadWrite**” (C#) /

The FMC can be enabled or disabled (HW is used a standard phased array). You can call the following functions to query if the FMS is enabled:

API (SW): “CSWDeviceOEMPA::**IsFullMatrixCapture**” (C++) /
“csSWDeviceOEMPA.**IsFullMatrixCapture**” (C#) /
API (HW): “CHWDeviceOEMPA::**GetEnableFMC**” (C++) /
“csHWDeviceOEMPA.**GetEnableFMC**” (C#) /

To enable FMC call the following function:

API (HW): “CHWDeviceOEMPA::**EnableFMC**” (C++) /
“csHWDeviceOEMPA.**EnableFMC**” (C#) /

7.2 Setup

The method to setup the HW is the same as the one to manage phased array, with the following caveats:

- The reception aperture is the same for all cycles, and is defined by two or three parameters: “ElementStart”, “ElementStop”, and “ElementStep” (Value defaults to ElementStep=1 if not used).
- Delays of emission and reception should be 0.
- WedgeDelays should be 0.
- The definition of the reception aperture can be simplified - you don’t need to specify all elements of the aperture. You can specify only one element of your reception aperture instead of all elements, and the setting of this single element will be used for all others (“ElementStart” and “ElementStop” are taken into account for the delay and the gain).
    - It is possible to use an OEMPA linear scanning file with an aperture of 1 element originally built for a non-FMC system (first cycle=first probe element for emission and reception, second cycle=second probe element for emission and reception).
- Function to set the DDF cannot be called (“SetReceptionFocusing” and “EnableDDF”).
- The “TimeSlot” is specified for all acquired elements.
    - This means that there is a risk of data loss, because the acquisition data throughput is multiplied by the element count. “OEMPATool” has a special

callback to multiply the “TimeSlot” of the default configuration file by the number of elements, but this is the single exception.

- You must use the same “TimeSlot” for all cycles (the first “TimeSlot” is used as a reference to optimize the throughput).
- In order to optimize throughput, two other “SubTimeSlot” parameters may specified as a means to increase PRF – “FMCSubTimeSlotAcq” and “FMCSubTimeSlotReplay”. See section 7.4 FW Recovery Time and FMC Timeslot Considerations.

You can specify element start, stop, and step via the following API:

API: “CHWDeviceOEMPA::**SetFMCElementStart**(int &iElementIndex)” (C++) /
“csHWDeviceOEMPA.**SetFMCElementStart**(int &iElementIndex)” (C#) /
Section “**Root**” the key “**FMCElementStart**” in OEMPA file

API: “CHWDeviceOEMPA::**SetFMCElementStop**(int &iElementIndex)” (C++) /
“csHWDeviceOEMPA.**SetFMCElementStop**(int &iElementIndex)” (C#) /
Section “**Root**” the key “**FMCElementStop**” in OEMPA file

API: “CHWDeviceOEMPA::**SetFMCElement**(int &iElementStart, int &iElementStop, int &iElementStep)” (C++) /
“csHWDeviceOEMPA.**SetFMCElement**(int &iElementStart, int &iElementStop, int &iElementStep)” (C#) /
Section “**Root**” the keys “**FMCElementStart**”, “**FMCElementStart**”, and “**FMCElementStep**” in OEMPA file

Example 1:

OEMPA file	**Comment**

[Root]
EnableFMC=1
FMCElementStart=0
FMCElementStop=64
	Acquisition from element 0 (first element of the probe) to element 63 (last element of 64-element probe).

Example 2:

OEMPA file	**Comment**

[Root]
EnableFMC=1
FMCElementStart=0
FMCElementStop=30
FMCElementStep=2
	Acquisition from element 0 to element 30 only using every other element to acquire A-scans (ElementStep = 2).

7.2.1 Example

OEMPA file	Comment
[Root]	
VersionDriverOEMPA=1.1.5.4	
CycleCount=1	
EnableFMC=1	
FMCElementStart=0	Reception aperture is element #1 to
FMCElementStop=31	Element #32 (for all cycles).
FMCElementStep=1	
[Cycle:0]	
GainDigital=0.0 dB	
BeamCorrection=0.0 dB	
Start=0.000000 us	
Range=10.0 us	
TimeSlot=3200.0 us	TimeSlot of one element is 3200/32=100 us.
PointCount=0	
CompressionType=Compression	
Rectification=Signed	
FilterIndex=0	
GainAnalog=20.0 dB	
GateCount=0	
[Cycle:0\Pulser]	
WedgeDelay=0.000000 us	
Element.count=1	
Element=0	Pulse only for the element #1
Delay.count=1;1	
Delay=0.000000 us	Emission delay should be 0.
Width.count=1	
Width=0.50 us	
[Cycle:0\Receiver]	
WedgeDelay=0.000000 us	
Element.count=1	
Element=0	

Focusing=Standard
Delay.count=1;1
Delay=0.000000 us
Gain.count=1
Gain=0.000000 dB
FocalTimeOfFlight.count=1
FocalTimeOfFlight=0.000000 us
	Reception delay should be 0. This value is used for all element of the reception aperture.
	The "FocalTimeOfFlight" array is not required by all versions. If required then its value should be 0 us.

7.3 Acquisition

Data arrive in the acquisition callback function in the same way as for phased array. In the case of FMC, the parameter "structAcqInfoEx &acqInfo" for C++ and "csAcqInfoEx acqInfo" for C# has two extra members:

- "bFullMatrixCapture": false in the case of phased array and true in the case of FMC.
- "lFMCElementIndex": element index of the receiver (this parameter is valid only if "bFullMatrixCapture" is true).

7.4 FW Recovery Time and FMC Timeslot Considerations

(The following applies for 128/128 configuration, with FW versions from 16.05.27 - 128/128 COM1 v1.3.3.5 and other configurations on FW versions from 16.06.15 -v2.3.4.1)

In Full Matrix Capture mode, the hardware sends the data of several elements during one single cycle. For example, with the following settings in the Root section of a setup file, the hardware will send the data for all 64 elements in each of the 64 cycles (4096 raw A-scans in total):

```
CycleCount=64
FMCElementStart=0
FMCElementStop=63
```

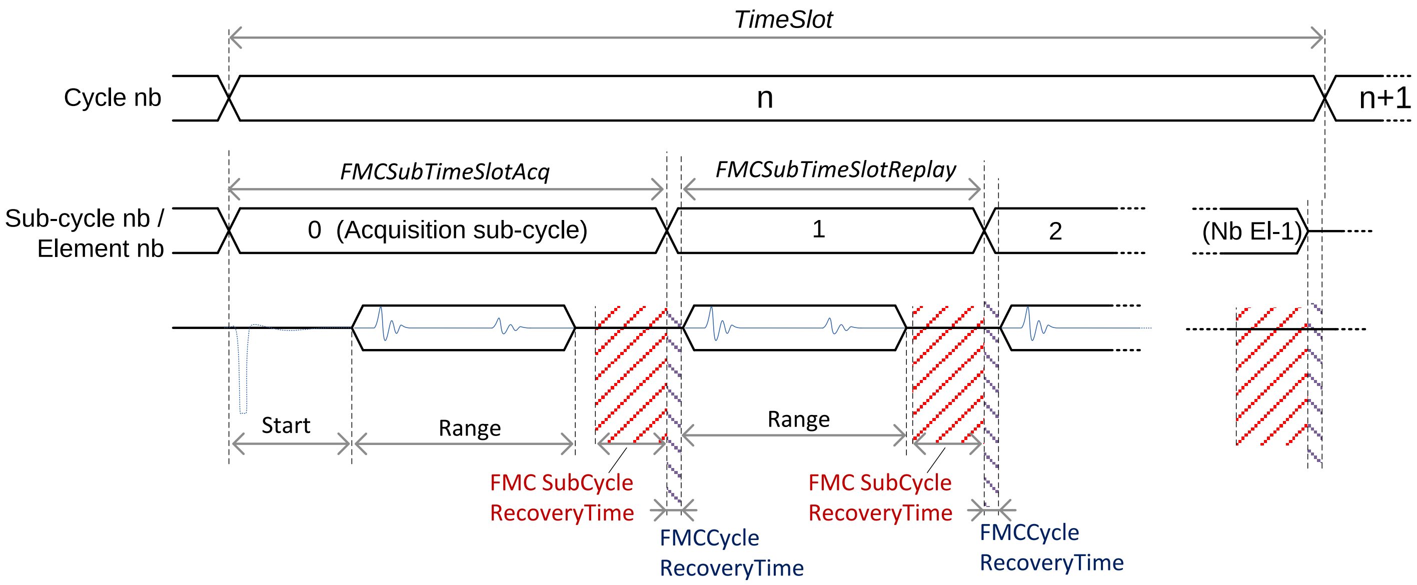
The firmware defines internally "sub-cycles" inside cycles to manage each element one after the other. (In the example, there are 64 sub-cycles inside each cycle). The limitation of the FW Recovery Time affects also the time of the sub-cycles.

There are 2 kinds of sub-cycles:

- *Acquisition sub-cycle*: The data of each element is saved inside individual buffers
- *Replay sub-cycles*: The previously saved data is read and sent to the PC.

The duration of these 2 kinds of sub-cycles can be defined with the 2 following parameters (the API can also calculate these 2 values from TimeSlot, Start and Range values, if the 2 parameters are not present in the setting file):

- *FMCSubTimeSlotAcq*
- *FMCSubTimeSlotReplay*



The 3 following formulas must be respected:

$$\boxed{FMCSubTimeSlotAcq \geq Start + Range + FMCSubCycleRecoveryTime} \quad \text{(FMC.a)}$$

$$\boxed{FMCSubTimeSlotReplay \geq Range + FMCSubCycleRecoveryTime} \quad \text{(FMC.b)}$$

$$\boxed{\begin{aligned} TimeSlot \geq\ & FMCSubTimeSlotAcq + (Nb\_El - 1) \times FMCSubTimeSlotReplay \\ & + Nb\_El \times FMCCycleRecoveryTime \end{aligned}} \quad \text{(FMC.c)}$$

In the even that ElementStep ≠ 0, equation FMC.c is modified:

$$\boxed{\begin{aligned} TimeSlot \geq\ & FMCSubTimeSlotAcq + \big((Nb_{El} \div \boldsymbol{Step}) - 1\big) \times FMCSubTimeSlotReplay \\ & + (Nb\_El \div \boldsymbol{Step}) \times FMCCycleRecoveryTime \end{aligned}} \quad \text{(FMC.c')}$$

The value of *FMCSubCycleRecoveryTime* is **10 us*** (value does include calibration time offset from HW Calibration, see paragraph 4.4.10 “FW Recovery Time”).

The value of *FMCCycleRecoveryTime* is **1.1 us***.

* Value as of September 2016. These values could change in the future FW versions.

The following API may be used to set timing values for sub-cycles:

API: "CHWDeviceOEMPA::**SetFMCSubTimeSlotAcqReplay**(int iCycle, double dAscanStart, double dAscanRange, double &dTimeSlot)" (C++) /
"csHWDeviceOEMPA.**SetFMCSubTimeSlotAcqReplay**(int iCycle, double dAscanStart, double dAscanRange, ref double dTimeSlot)" (C#)
Section "**Cycle**" the key "**TimeSlot**" in OEMPA file.
This is the simple way of setting the timing as the FMCSubTimeSlotAcq and FMCSubTimeSlotReplay are calculated and set automatically based on the inputs. To manually these values, the following functions may be used:

API: "CHWDeviceOEMPA::**SetFMCSubTimeSlotAcq**(int iCycle, double &dSubTimeSlot1)" (C++) /
"csHWDeviceOEMPA.**SetFMCSubTimeSlotAcq**(int iCycle, ref double dSubTimeSlot1)" (C#) /
Section "**Cycle**" the key "**FMCSubTimeSlotAcq**" in OEMPA file

API: "CHWDeviceOEMPA::**SetFMCSubTimeSlotReplay**(int iCycle, double &dSubTimeSlot2)" (C++) /
"csHWDeviceOEMPA.**SetFMCSubTimeSlotReplay**(int iCycle, ref double dSubTimeSlot2)" (C#) /
Section "**Cycle**" the key "**FMCSubTimeSlotReplay**" in OEMPA file

The following notes relate to OEMPA file settings:

- If **FMCSubTimeSlotAcq** and **FMCSubTimeSlotReplay** are not specified in **Cycle** section of the OEMPA file, they are set automatically according to **SetFMCSubTimeSlotAcqReplay**.
- **TimeSlot=0 us** may be validly used in the **Cycle** section of the OEMPA file. Setting TimeSlot to 0 will automatically choose the minimum values for TimeSlot, FMCSubTimeSlotAcq, and FMCSubTimeSlotReplay as constrained by recovery time and throughput limitations to maximize PRF.

The API functions that allow the developer to know Recovery Time differ those in paragraph 4.4.10. There is also a function for overall FMC Time Limitation. For FMC, the following functions are available (after version 1.1.5.3t):

Function	Comment
CSWDevice::GetFMCSubCycleRecoveryTime (C++) csSWDevice.GetFMCSubCycleRecoveryTime (C#)	To get the values of the "FW Recovery Time" values of the Hardware connected (result includes calibration time offset, see paragraph "FW Recovery Time").
CSWDevice::GetFMCCycleRecoveryTime (C++) csSWDevice.GetFMCCycleRecoveryTime (C#)	To get the values of the "FW Recovery Time" values of the Hardware connected
CSWDevice::GetFMCTimeLimitation (C++) csSWDevice.GetFMCTimeLimitation (C#)	If given the Start, Range, etc. for FMC, this function determines the minimum

	allowed values for TimeSlot and the sub-cycle times. These are the same values used for TimeSlot=0 cycles in FMC OEMPA files.