`USER GUIDE

Mosaic III Software Development Kit

v3.4

www.andor.com

Version 1.0 – June 2012

PAGE

SECTION 1 - Installation

1.1 - Technical Support

If you have any questions regarding the use of this equipment, please contact the representative from whom your system was purchased, or:

Europe

Andor Technology
7 Millennium Way
Springvale Business Park
Belfast
BT12 7AL
Northern Ireland
Tel. +44 (0) 28 9023 7126
Fax. +44 (0) 28 9031 0792
e-mail: productsupport@andor.com

USA

Andor Technology
425 Sullivan Avenue
Suite # 3
South Windsor
CT 06074
USA
Tel. (860) 290-9211
Fax. (860) 290-9566
e-mail: productsupport@andor.com

Asia-Pacific

Andor Technology (Japan)
7F Ichibancho Central Building
22-1 Ichiban-Cho,
Chiyoda-Ku
Tokyo 102-0082
Japan
Tel. +81 3 3511 0659
Fax. +81 3 35110662
e-mail: productsupport@andor.com

China

Andor Technology
Rm 1213
Building B
Luo Ke Time Square
No. 103 Huizhongli
Chaoyang District
Beijing 100101
China
Tel. +86-10-5129-4977
Fax. +86-10-6445-5401
e-mail: productsupport@andor.com

1.5 - Installation

The following sections will describe how to install the software and hardware in order to make your Mosaic III system ready to use.

1.5.1 – Windows Installation

Ensure that you do not install the PCI Express card before running any of the software installations.

1. Installation of SDK3 and PCI Express card drivers:

[Note: You must have administrator access on your PC to perform the installation.]

- Run the setup_mosaic3.exe file on the cd or from download.
- Select the installation directory or accept the default when prompted by the installer.
- Click on the Install button to confirm and continue with the installation.
- During the installation a number of other windows will pop up as the device drivers and SDK3 are installed. Click on the Finish button when prompted.

2. PCI Express Card Installation:

- Shut down your PC and remove the power cable.
- Take adequate Anti-static precautions to prevent damage to the PCI-E card or the PC
- Install the PCI Express card into a free PCI Express slot on your motherboard.
- Power on the PC.

1.6 – Getting Started

This section will demonstrate how to create a basic SDK3 application that will test the software & hardware installation, and will help to confirm communication between PC and Mosaic.

1.6.1 – Windows Getting Started

Running the examples that came with the installation

In the installation directory there is an examples directory. In that directory there are few further directories, such as 'framedisplay' and 'serialnumber'. These directories contain the executables and the required dll's to initialise and communicate with the Mosaic.
The frame display example will initialise and expose a rectangle mask on the device for a limited time.
The serial number example will initialise and print out the serial number of the device.
The sequence example will initialise, upload and expose sequence of frames.
The grayscale example will initialise, upload and expose sequence of frames to simulate the grayscale effect.

Creating your own applications

With this installation you can create an application with a Embarcadero or Microsoft compatible compiler. Perform the following steps to create your application.

1. Create a simple console application with either Embarcadero C++ Builder or Microsoft Visual Studio.
2. Add the SDK3 installation directory to the include path for the project. Eg. C:\Program Files\Andor Mosaic 3\SDK
3. Add the appropriate library from the SDK3 installation directory to your project.
    - atcore.lib for the Embarcadero compiler

- atcorem.lib for the Microsoft compiler

4. Copy all the DLL's from the SDK3 installation directory to the directory that the executable is going to run from.
5. Type or copy the code shown below into your projects main file.
6. Compile and run the program. The program should initialise the first device found and display it's serial number.
7. If the serial number is not displayed then follow the comments in the code listing for hints on tracking down any issues.

NOTE: It is assumed that there is one Mosaic III device attached and unspecified number of other devices that support SDK3.

```
#include "atcore.h"
#include <iostream>
using namespace std;

int main(int argc, char* argv[])
{
  int i_retCode;
  AT_H MosaicHndl = -1;
  i_retCode = AT_InitialiseLibrary();
  if (i_retCode != AT_SUCCESS) {
    //error condition, check atdebug.log file
  }
  AT_64 iNumberDevices = 0;
  i_retCode  = AT_GetInt(AT_HANDLE_SYSTEM, L"DeviceCount", &iNumberDevices);
  if (iNumberDevices <= 0) {
    // No devices found, check all redistributable binaries
    // have been copied to the executable directory or are in the system path
    // and check atdebug.log file
  }
  else {
    AT_H Hndl;
    for (int i = 0; i < iNumberDevices; i++) {
      i_retCode = AT_Open(i, &Hndl);
      if (i_retCode != AT_SUCCESS) {
        //error condition - check atdebug.log
      }

      AT_WC szDeviceType[64];
      AT_GetString(Hndl, L"DeviceType", szDeviceType, 64);
      if (wcscmp(szDeviceType, L"Mosaic III") == 0) {

        wcout << L"Mosaic III device found" << endl;
        AT_WC szValue[64];
        i_retCode= AT_GetString(Hndl, L"SerialNumber", szValue, 64);
        if (i_retCode == AT_SUCCESS) {
          //The serial number of the device is szValue
          wcout << L"The serial number is " << szValue << endl;
        }
        else {
         //Serial Number feature was not found, check the error code for
information
        }
        AT_Close(Hndl);
        break;
      }
      AT_Close(Hndl);
    }
  }
  AT_FinaliseLibrary();
  return 0;
}
```

SECTION 2 – API

2.1 – Overview

The SDK3 API can be divided into several sets of functions, each controlling a particular aspect of Mosaic device. There are sections in the API for opening a handle to a device, for buffer management and for accessing the features that every device exposes. Each feature that a device exposes to the user has a particular type that represents how that feature is controlled. The feature types are:

- Integer
- Floating Point
- Boolean
- Enumerated
- Command
- String

For example, the ExposureTime feature is of type Floating Point, and the Expose feature is of type Command. Each of these feature types, the management of multiple devices and buffer management are described in the sections below. The character type used by the API is a 16 bit wide character defined by the AT_WC type, which is used to represent all feature names, enumerated options and string feature values.

Wide Characters

An example of converting wide character strings to char strings can be found in the appendix.

2.2 - Function Listing

```
int AT_InitialiseLibrary();
int AT_FinaliseLibrary();

int AT_Open(int DeviceIndex, AT_H* Handle);
int AT_Close(AT_H Hndl);

typedef int (*FeatureCallback)(AT_H Hndl, AT_WC* Feature, void* Context);
int AT_RegisterFeatureCallback(AT_H Hndl, AT_WC* Feature, FeatureCallback EvCallback, void* Context);
int AT_UnregisterFeatureCallback(AT_H Hndl, AT_WC* Feature, FeatureCallback EvCallback, void* Context);

int AT_IsImplemented(AT_H Hndl, AT_WC* Feature, AT_BOOL* Implemented);
int AT_IsReadOnly(AT_H Hndl, AT_WC* Feature, AT_BOOL* ReadOnly);
int AT_IsReadable(AT_H Hndl, AT_WC* Feature, AT_BOOL* Readable);
int AT_IsWritable(AT_H Hndl, AT_WC* Feature, AT_BOOL* Writable);

int AT_SetInt(AT_H Hndl, AT_WC* Feature, AT_64 Value);
int AT_GetInt(AT_H Hndl, AT_WC* Feature, AT_64* Value);
int AT_GetIntMax(AT_H Hndl, AT_WC* Feature, AT_64* MaxValue);
int AT_GetIntMin(AT_H Hndl, AT_WC* Feature, AT_64* MinValue);

int AT_SetFloat(AT_H Hndl, AT_WC* Feature, double Value);
int AT_GetFloat(AT_H Hndl, AT_WC* Feature, double* Value);
int AT_GetFloatMax(AT_H Hndl, AT_WC* Feature, double* MaxValue);
int AT_GetFloatMin(AT_H Hndl, AT_WC* Feature, double* MinValue);

int AT_SetBool(AT_H Hndl, AT_WC* Feature, AT_BOOL Value);
int AT_GetBool(AT_H Hndl, AT_WC* Feature, AT_BOOL* Value);

int AT_SetEnumIndex(AT_H Hndl, AT_WC* Feature, int Value);
int AT_SetEnumString(AT_H Hndl, AT_WC* Feature, AT_WC* String);
int AT_GetEnumIndex(AT_H Hndl, AT_WC* Feature, int* Value);
int AT_GetEnumCount(AT_H Hndl, AT_WC* Feature, int* Count);
int AT_IsEnumIndexAvailable(AT_H Hndl, AT_WC* Feature, int Index, AT_BOOL* Available);
int AT_IsEnumIndexImplemented(AT_H Hndl, AT_WC* Feature, int Index, AT_BOOL* Implemented);
int AT_GetEnumStringByIndex(AT_H Hndl, AT_WC* Feature, int Index, AT_WC* String, int StringLength);

int AT_Command(AT_H Hndl, AT_WC* Feature);

int AT_SetString(AT_H Hndl, AT_WC* Feature, AT_WC* Value);
int AT_GetString(AT_H Hndl, AT_WC* Feature, AT_WC* Value, int StringLength);
int AT_GetStringMaxLength(AT_H Hndl, AT_WC* Feature, int* MaxStringLength);

int AT_QueueBuffer(AT_H Hndl, AT_U8* Ptr, int PtrSize);
int AT_WaitBuffer(AT_H Hndl, AT_U8** Ptr, int* PtrSize, unsigned int Timeout);
int AT_Flush(AT_H Hndl);
```

2.3 – API Description

2.3.1 – Library Initialisation

The first API function call made by any application using SDK3 must be:

```
AT_InitialiseLibrary()
```

This allows the SDK to setup its internal data structures and to detect any device that are attached. AT_InitialiseLibrary takes no parameters.

Before your application closes or when you no longer wish to access the API you should call the function :

```
AT_FinaliseLibrary()
```

This will clean up any data structures held internally by the SDK.

.

2.3.2 – Opening a Device Handle

To access the features provided by a device and to expose masks you must first open a handle. A device handle, represented by the data type AT_H, is a reference to the particular device that you wish to control and is passed as the first parameter to most other functions in the SDK. In multi-device environments the handle becomes particularly useful as it allows devices to be controlled simultaneously in a thread safe manner. To open a handle to a device you should pass the index of the device, that you wish to access, to the function :

```
AT_Open(int DeviceIndex, AT_H* Handle)
```

The handle will be returned in the Handle parameter which is passed by address. To open the first device you should pass a value of 0 to the DeviceIndex parameter, for the second device pass a value of 1 etc.

Once you have finished with the device it should be closed using the function :

```
AT_Close(AT_H Handle)
```

The only parameter to this function is the handle of the device that you wish to release.

System Handle

There are some features of the system that are not connected to a specific device but are global properties. For example, the DeviceCount feature stores a count of the number of devices that are currently connected. To access these features you do not need to open a handle to a device, instead you should use the system handle represented by the constant AT_HANDLE_SYSTEM. You do not need to retrieve this handle using the AT_Open function; it is predefined and can be used immediately after the AT_InitialiseLibrary function has completed.

Thread Safety

SDK3 is thread safe when accessing different devices on different threads. However you should not use multiple threads at once to access the features of the same device. The exception to this is that the AT_WaitBuffer and AT_QueueBuffer functions can be called simultaneously on separate threads, with the same device handle, during acquisition.

2.3.3 – Integer Features

Integer features are those that can be represented by a single integer value. To set an Integer feature use the function :

```
AT_SetInt(AT_H Hndl, AT_WC* Feature, AT_64 Value)
```

The first parameter is the handle to the device that is exposing the desired feature; the second parameter is a wide character string indicating the name of the feature that you wish to modify. The full list of feature strings is available in the Feature Reference section. The third parameter contains the value that you want to assign to the feature. The function will return a value indicating whether the function successfully applied the value. The Error Codes section lists the possible error codes that can be returned from the Integer Type functions.

To get the current value for an Integer feature use the function :

```
AT_GetInt(AT_H Hndl, AT_WC* Feature, AT_64 * Value)
```

The first two parameters are the device handle and the feature name, the same as those passed to the AT_SetInt function. The third parameter is the address of the variable into which you want to store the Integer value.

Integer features can sometimes be restricted in the range of values that they can be set to. This range of possible values can be determined by using the functions

```
AT_GetIntMax(AT_H Hndl, AT_WC* Feature, AT_64 * MaxValue)
AT_GetIntMin(AT_H Hndl, AT_WC* Feature, AT_64 * MinValue)
```

These functions work similarly to the AT_GetInt function except that the third parameter returns either the highest allowable value or the lowest allowable value. Using these maximum and minimum values you can check which values are allowed by AT_SetInt without having to monitor its return code. This can be useful, for example, when you wish to limit the range of possible values that the user can enter in a GUI application.

2.3.4 – Floating Point Features

Floating point type features work in a similar way to Integer features, in that they have a Set function and Get function and GetMin and GetMax functions. Floating Point features represent those features that are expressed with a value that contains a decimal point. As an example, the ExposureTime feature is exposed through the Floating Point functions. The functions are :

```
AT_SetFloat(AT_H Hndl, AT_WC* Feature, double Value)
AT_GetFloat(AT_H Hndl, AT_WC* Feature, double * Value)
AT_GetFloatMax(AT_H Hndl, AT_WC* Feature, double * MaxValue)
AT_GetFloatMin(AT_H Hndl, AT_WC* Feature, double * MinValue)
```

The first parameter to each of these functions is the device handle, the second parameter is the name of the feature and the third parameter contains either the value that you wish to set or the address of a variable that will return the current value, maximum or minimum of the feature. The list of possible error codes is described in the Error Codes section.

2.3.5 – Boolean Features

Boolean features can only be set to one of two possible values, representing the logical states true and false. True is represented by the value AT_TRUE and false by the value AT_FALSE. To change the state of a boolean feature use the function :

```
AT_SetBool(AT_H Hndl, AT_WC* Feature, AT_BOOL Value)
```

The first parameter is a handle to the device being used, the second parameter is the string descriptor of the feature to change and the third parameter is the value. So to enable a boolean feature pass a value of AT_TRUE, to disable a boolean feature, pass a value of AT_FALSE in the third parameter. To retrieve the current state of a boolean feature, use the function :

```
AT_GetBool(AT_H Hndl, AT_WC* Feature, AT_BOOL* Value)
```

For this function the third parameter contains the address of the variable into which you want the state stored. A value of AT_FALSE means the feature is disabled, while a value of AT_TRUE means the feature is enabled.

2.3.6 – Enumerated Features

Enumerated features are used to represent those features that can be assigned one value from a set of possible options. For example, the triggering mode that you wish to use with the device is set using the TriggerMode enumerated feature. The triggering mode setting can be chosen from a number of options, for example, internal, hardware or software external. The enumerated feature functions allow you to :

- Determine how many options are available
- Select which option you wish to use
- Retrieve a human readable representation of each option.

Enumerated options can be set either by their text value or by index, using the functions :

```
AT_SetEnumIndex(AT_H Hndl, AT_WC* Feature, int Index)
AT_SetEnumString(AT_H Hndl, AT_WC* Feature, AT_WC* String)
```

The first function changes the current item to the one that lies at the position specified by the Index parameter, where an Index of 0 is the first item. The second function lets you specify the string descriptor for the particular option that you wish to use. String Descriptors for all features are described in the Feature Reference section. As for all feature access functions the first two parameters are the device handle and the string descriptor of the feature that you wish to modify. The choice of which function to use will depend on your particular application and they can both be used in the same program.

Enumerated Indexes

The particular index that maps to an enumerated option may be different across SDK versions and across different devices. To ensure best compatibility for your application you should use strings wherever possible and avoid assuming that a specific option is found at a particular index.

To find out which option is currently selected for an enumerated feature, you can use the function :

```
AT_GetEnumIndex(AT_H Hndl, AT_WC* Feature, int* Value)
```

The third parameter is the address of the variable where you want the currently selected index stored.

To find out how many options there are available for the feature, use the function :

```
AT_GetEnumCount(AT_H Hndl, AT_WC* Feature, int* Count)
```

The third parameter, on return, will contain the number of possible options. If you attempt to select an option using AT_SetEnumIndex with an index either below zero or above or equal to this count an error will be returned.

You can retrieve the string descriptor for any option by calling the function :

```
AT_GetEnumStringByIndex(AT_H Hndl, AT_WC* Feature, int Index, AT_WC* String,
                       int StringLength)
```

The third parameter is the index of the option that you want to receive the descriptor for, the fourth parameter is a user allocated buffer to receive the descriptor and the fifth parameter is the length of the allocated buffer.

Enumerated Index Availability

In some situations one or more of the options listed for an enumerated feature may be either permanently or temporarily unavailable. An option may be permanently unavailable if the device does not support this option, or temporarily unavailable if the current value of other features do not allow this option to be selected. To find out which options are available you can use the functions :

```
AT_IsEnumIndexAvailable(AT_H Hndl, AT_WC* Feature, int Index, AT_BOOL* Available)
AT_IsEnumIndexImplemented(AT_H Hndl, AT_WC* Feature, int Index,
                                  AT_BOOL* Implemented)
```

Both functions take the index of the option that you want to interrogate in the third parameter. The first function determines if the option is only temporarily unavailable, the second function determines if the feature is permanently unavailable. The fourth parameter returns the availability status, a value of AT_FALSE means unavailable, and a value of AT_TRUE means the option is available. If you try to select an option that is unavailable using either of the set functions then an error will be returned.

2.3.7 – Command Features

Command features are those that represent a single action. For example, to start the device exposing a mask you will use the Command feature Expose. These commands do not require any extra parameters and are simply called by passing the string descriptor of the command to the function :

```
AT_Command(AT_H Hndl, AT_WC* Feature)
```

2.3.8 – String Features

String features are those that can be represented by 1 or more characters. An example of a String feature is the Serial Number of the device. In many cases these features are read only but if they are writable, you can set the value using the function :

```
AT_SetString(AT_H Hndl, AT_WC* Feature, AT_WC* Value)
```

The first parameter contains the device handle, the second parameter is the string descriptor of the feature, and the third parameter is the character string that you want to assign to the feature.

To retrieve the value of a String feature use the function :

```
AT_GetString(AT_H Hndl, AT_WC* Feature, AT_WC* Value, int StringLength)
```

In this case the third parameter should be a caller allocated character string that will be used to receive the string. The fourth parameter is the length of the caller allocated buffer. To determine what length of string that should be allocated to receive the string value you can use the function :

```
AT_GetStringMaxLength(AT_H Hndl, AT_WC* Feature, int* MaxStringLength)
```

The maximum length of the String feature will be returned in the third parameter.

2.3.9 – Buffer Management

Queueing

The input queue, which is written into by the application and read from by the SDK, is used to store the memory buffer to fill in by the user with any mask data to be exposed on the device. This queue is accessed using the function :

```
AT_QueueBuffer(AT_H Hndl, AT_U8 * Ptr, int PtrSize)
```

The first parameter contains the handle to the device. The second parameter is the address of an application allocated buffer large enough to store a single image. The PtrSize parameter should contain the size of the buffer that is being queued. The required size of the buffer can be obtained by reading the value of the ImageSizeBytes integer feature.

2.3.10 – Feature Access Control

The individual access rights of features can be determined using a set of functions that apply to all features, independent of their type. The four access characteristics of a feature, are :

- Whether a feature is implemented by a device.
- Whether a feature is read only.
- Whether a feature can currently be read.
- Whether a feature can currently be modified.

The first two access rights are permanent characteristics of the feature, the second two access rights may change during the running of the program. For example, if other features are modified in such a way as to affect this feature. If a feature is not implemented by a device then any attempt to access that feature will return the error code `AT_ERR_NOTIMPLEMENTED`. Any attempt to modify a read only feature will result in the error code `AT_ERR_READONLY`. If a feature cannot be currently read then any attempt to get the current value will return the `AT_ERR_NOTREADABLE` error code and any attempt to modify a value that cannot currently be written to will return the error code `AT_ERR_NOTWRITABLE`. To determine if a feature has been implemented use the function :

```
AT_IsImplemented(AT_H Hndl, AT_WC* Feature, AT_BOOL* Implemented)
```

The first two parameters are, as usual, the handle to the device and the string descriptor of the feature. The third parameter is an output parameter which returns with a value indicating whether the feature is implemented or not. If Implemented contains the value AT_TRUE, on return from the function then the feature is implemented, if it returns with the value AT_FALSE, then the feature is not implemented.

To determine if a feature is read only, use the function :

```
AT_IsReadOnly(AT_H Hndl, AT_WC* Feature, AT_BOOL* ReadOnly)
```

This function works in a similar way to AT_IsImplemented, that is, if the ReadOnly parameter returns with the value AT_TRUE then the feature is read only, a value of AT_FALSE indicates that it can be modified. SerialNumber is an example of a feature that is read only.

To determine if a feature is currently readable, use the function :

```
AT_IsReadable(AT_H Hndl, AT_WC* Feature, AT_BOOL* Readable)
```

The Readable parameter indicates whether the feature is currently readable in the same manner as the ReadOnly parameter to the function AT_IsReadOnly.

To determine if a feature is currently writable use the function :

```
AT_IsWritable(AT_H Hndl, AT_WC* Feature, AT_BOOL* Writable)
```

An example of the use of this function is to allow a GUI application to disable access to features when they cannot be modified.

2.3.11 – Feature Notifications

Sometimes a feature may change its value or its other characteristics, not as a direct result of the user modifying the feature, but indirectly through modification of a separate feature.

To allow the application to receive notification when this type of indirect change occurs, there are functions provided in the API that allow the application to create a callback function and attach it to a feature. Whenever the feature changes in any way, this callback will be triggered, allowing the application to carry out any actions required to respond to the change. For example, if an application provides a GUI interface that allows users to modify features, then the callback can update the GUI with any changes. This facilitates use of the Observer design pattern in your application. The callback will also be triggered if the feature is modified directly by the application.

The definition of the callback function implemented by the application should be in the following format :

```
int AT_EXP_CONV MyCallback(AT_H Hndl, const AT_WC* Feature, void* Context)
{
      // Perform action
}
```

There are three parameters sent to the function that allow the application to determine the reason for the call-back. The first parameter indicates which device caused the call-back. By using this parameter you can make use of the same call-back function for multiple devices. The second parameter holds the string descriptor of the feature that has been modified in some way, and allows the same call-back function to be used with multiple features. The final parameter is an application defined context parameter that was passed in as a parameter at the time that the call-back function was registered. The Context parameter is not parsed in any way by the SDK and can be used to store any information that the application wishes.

Note that the AT_EXP_CONV modifier must be present and ensures that the correct calling convention is used by the SDK.

To register the call-back function, use the function :

```
AT_RegisterFeatureCallback(AT_H Hndl, AT_WC* Feature, FeatureCallback EvCallback,
                           void* Context)
```

The first parameter is the device handle, the second parameter is the string descriptor of the feature that you wish to receive notifications for. The third parameter is your call-back function that you have defined as described above and the fourth parameter in the Context parameter that will be passed to the call-back each time it is called. Whenever the application no longer requires notifications for a particular feature, it should release the call-back by calling the function :

```
AT_UnregisterFeatureCallback(AT_H Hndl, AT_WC* Feature, FeatureCallback EvCallback,
                             void* Context)
```

The same parameters should be passed to this function as were passed to the AT_RegisterFeatureCallback.

You need to register a call-back individually for each feature that you are interested in, but the same call-back function can be used for all or some features, or, a separate call-back function can be provided for each feature.

Notes on implementing call-backs

A call-back should complete any work required in the minimal amount of time as it holds up the thread that caused the call-back. If possible the application should delegate any work to a separate application thread if the action will take a significant amount of time.

The call-back function should not attempt to modify the value of any feature as this can cause lockup.

Find below the available return codes and their values for each feature type and the buffer control functions.

Device Connection	
AT_SUCCESS (0)	Function call has been successful
AT_ERR_NONINITIALISED (1)	Function called with an uninitialised handle
AT_ERR_CONNECTION (10)	Error connecting to or disconnecting from hardware
AT_ERR_NULL_HANDLE (21)	Null device handle passed to function
AT_ERR_DEVICEINUSE (38)	Function failed to connect to a device because it is already being used

String Feature	
AT_SUCCESS (0)	Function call has been successful
AT_ERR_NOTIMPLEMENTED (2)	Feature has not been implemented for the chosen device
AT_ERR_READONLY (3)	Feature is read only
AT_ERR_NOTWRITABLE (5)	Feature is currently not writable
AT_ERR_NOTREADABLE (4)	Feature is currently not readable
AT_ERR_ EXCEEDEDMAXSTRINGLENGTH (9)	String value provided exceeds the maximum allowed length
AT_ERR_COMM (17)	An error has occurred while communicating with hardware

Integer Feature	
AT_SUCCESS (0)	Function call has been successful
AT_ERR_OUTOFRANGE (6)	Value is outside the maximum and minimum limits
AT_ERR_NOTIMPLEMENTED (2)	Feature has not been implemented for the chosen device
AT_ERR_READONLY (3)	Feature is read only
AT_ERR_NOTWRITABLE (5)	Feature is currently not writable
AT_ERR_NOTREADABLE (4)	Feature is currently not readable
AT_ERR_COMM (17)	An error has occurred while communicating with hardware

Float Feature	
AT_SUCCESS (0)	Function call has been successful
AT_ERR_OUTOFRANGE (6)	Value is outside the maximum and minimum limits
AT_ERR_NOTIMPLEMENTED (2)	Feature has not been implemented for the chosen device
AT_ERR_READONLY (3)	Feature is read only
AT_ERR_NOTWRITABLE (5)	Feature is currently not writable
AT_ERR_NOTREADABLE (4)	Feature is currently not readable
AT_ERR_COMM (17)	An error has occurred while communicating with hardware

Boolean Feature	
AT_SUCCESS (0)	Function call has been successful
AT_ERR_OUTOFRANGE (6)	The value passed to the function was not a valid boolean value i.e. 0 or 1.
AT_ERR_NOTIMPLEMENTED (2)	Feature has not been implemented for the chosen device
AT_ERR_READONLY (3)	Feature is read only
AT_ERR_NOTWRITABLE (5)	Feature is currently not writable
AT_ERR_NOTREADABLE (4)	Feature is currently not readable
AT_ERR_COMM (17)	An error has occurred while communicating with hardware

Enumerated Feature	
AT_SUCCESS (0)	Function call has been successful
AT_ERR_OUTOFRANGE (6)	The index passed to the function was either less than zero or greater than or equal to the number of implemented options.
AT_ERR_NOTIMPLEMENTED (2)	Feature has not been implemented for the chosen device
AT_ERR_READONLY (3)	Feature is read only
AT_ERR_NOTWRITABLE (5)	Feature is currently not writable
AT_ERR_NOTREADABLE (4)	Feature is currently not readable
AT_ERR_ INDEXNOTAVAILABLE (7)	Index is currently not available
AT_ERR_ INDEXNOTIMPLEMENTED (8)	Index is not implemented for the chosen device
AT_ERR_STRINGNOTAVAILABLE(18)	Index / String is not available
AT_ERR_ STRINGNOTIMPLEMENTED (19)	Index / String is not implemented for the chosen device
AT_ERR_COMM (17)	An error has occurred while communicating with hardware

Command Feature	
AT_SUCCESS (0)	Function call has been successful
AT_ERR_NOTIMPLEMENTED (2)	Feature has not been implemented for the chosen device
AT_ERR_NOTWRITABLE (5)	Feature is currently not executable
AT_ERR_COMM (17)	An error has occurred while communicating with hardware

Buffer Control	
AT_SUCCESS (0)	Function call has been successful
AT_ERR_TIMEDOUT (13)	The AT_WaitBuffer function timed out while waiting for data arrive in output queue **Not supported in Mosaic III SDK**
AT_ERR_BUFFERFULL (14)	The input queue has reached its capacity

	Not supported in Mosaic III SDK
`AT_ERR_INVALIDSIZE (15)`	The size of a queued buffer did not match the frame size **Not supported in Mosaic III SDK**
`AT_ERR_INVALIDALIGNMENT (16)`	A queued buffer was not aligned on an 8-byte boundary **Not supported in Mosaic III SDK**
`AT_ERR_HARDWARE_OVERFLOW (100)`	The software was not able to retrieve data from the card or device fast enough to avoid the internal hardware buffer bursting. **Not supported in Mosaic III SDK**
`AT_ERR_COMM (17)`	An error has occurred while communicating with hardware

SECTION 3 - Features

3.1 Device Support

SDK3 currently supports the following Andor products:

- Neo camera
- Zyla camera
- Mosaic III device.

This document describes the features available only for Mosaic III so for any other Andor product refer to the appropriate manual. Mosaic III product features are outlined below.

3.2 Feature Reference

Feature	Type	Description
Abort	Command	Aborts the currently displayed mask by switching all the micromirrors off. It's recommended to execute that command to make sure no mask is displayed on the device prior to displaying another mask.
ClearFrameMemory	Command	Deletes all masks stored in the frame memory.
ClearSequenceMemory	Command	Deletes all sequence events stored in the sequence memory.
ControllerFirmwareVersion	String	Returns the controller card firmware version.
DeviceCount	Integer	Returns the number of devices detected.
DeviceType	String	Returns the device model.
Expose	Command	Start exposing a frame or sequence of frames on the device. The command will fail if the device is currently displaying a mask.
ExposureTime	Floating Point	Time in seconds a single frame is exposed onto the device in Live/FrameMemory mode.
ImageSizeBytes	Integer	Returns the buffer size in bytes required to store the data for one frame. The buffer size varies for different PixelEncoding options so the user should read the current ImageSizeBytes value each time the PixelEncoding changes.
InvertMode	Boolean	If the feature is set to true then all bits of a mask will be inverted before gets sent to the device. This feature has no effect on the existing masks in the frame memory.
IsExposing	Boolean	Returns whether or not the device is exposing a mask.
FirmwareVersion	String	Returns the device firmware version.
FrameCount	Integer	Specifies the number of frames for a single sequence event upload.
FrameCycleCount	Integer	Specifies the number of repeats for a single sequence event upload.
FrameIndex	Integer	The current mask index in the frame memory. The value of the feature is also used to specify a start frame index for a single sequence event upload.
FrameIndexSource	Enumerated	Configures the frame index source to display frames from the frame memory. The feature can only be used in FrameMemory mode (see OperationMode feature for details). <u>Options:</u> • **Software** (default value) Index of the current mask to display from the frame memory onto the device is specified by FrameIndex feature • **Hardware** Index of the current mask to display from the frame memory onto the device is specified by an external electric switch

FrameMemoryCount	Integer	Returns total number of masks stored in the frame memory.
GrayScaleExposureTime	Floating Point	The cycle time during a single grayscale sequence is exposed on the device in seconds when displayed in FrameSequence/ContinuousFrameSequence mode.
GrayScaleSequenceEncoding	Enumerated	Configures the number of bit planes in the user buffer when uploading a grayscale sequence. A single byte in the user buffer can contain up to 8 bit planes which then can be uploaded to the sequence memory by executing UpdateGrayScaleImage command. The planes will be uploaded staring from the left (Bit8, Bit6, Bit4 depending on selected option) towards the least significant bit (lsb). A single byte sequence encoding: Bit8 Bit7 Bit6 Bit5 Bit4 Bit3 Bit2 Bit1 [msb | | | | | | | lsb] <u>Options:</u> • **4BitDIP** A single byte in the user buffer contains 4 bit planes (Bit4, Bit3, Bit2,Bit1) that will be uploaded for a grayscale sequence. • **6BitDIP** A single byte in the user buffer contains 6 bit planes (Bit6, Bit5, Bit4, Bit3, Bit2,Bit1) that will be uploaded for a grayscale sequence • **8BitDIP** (default value) A single byte in the user buffer contains 8 bit planes (Bit8, Bit7, Bit6, Bit5, Bit4, Bit3, Bit2,Bit1) that will be uploaded for a grayscale sequence
OperationMode	Enumerated	Configures the device operating mode. <u>Options:</u> • **Live** (default value) A mask is displayed directly from the user defined buffer onto the device. • **FrameMemory** A mask which number is specified by the current value of FrameIndex feature is displayed from the frame memory onto the device • **FrameSequence** Sequence events are displayed from the sequence memory onto the device. • **ContinuousFrameSequence** Sequence events are displayed from the sequence memory onto the device. Masks defined in the sequence are displayed in a loop for unlimited length of time until the Abort command is executed.
OverlapMode	Boolean	If the feature is set to true then there is no mirrors off period between displaying frames in FrameSequence/ContinuousFrameSequence mode. The current SequenceGapTime value is ignored if OverlapMode is set to true.
PixelEncoding	Enumerated	Configures the format of data stream. <u>Options:</u> • **Mono1** (default value) A single byte in the user buffer stores a single pixel value.

		• **Mono1Packed** A single byte in the user buffer stores 8 consecutive pixel values. This option is recommended in LiveMode for the fast frame rate streaming directly from a PC onto the device.
SensorHeight	Integer	Returns the height of the sensor as a number of micromirrors.
SensorWidth	Integer	Returns the width of the sensor as a number of micromirrors.
SerialNumber	String	Returns the device serial number.
SequenceEventCount	Integer	Returns total number of sequence events stored in the sequence memory.
SequenceExposureTime	Floating Point	The time a frame is exposed on the device in seconds when displayed in FrameSequence/ContinuousFrameSequence mode.
SequenceLoopCount	Integer	Number of sequence repeats when displaying masks in FrameSequence/ContinuousFrameSequence mode.
SequenceLoopLength	Integer	Number of events in the sequence memory to iterate through when displaying masks in FrameSequence/ContinuousFrameSequence mode.
SequenceGapTime	Floating Point	The wait time (all mirrors off) in seconds between exposing the current and the next mask in FrameSequence or ContinuousFrameSequence mode. The wait time is ignored if OverlapMode value is set to true.
SequenceStartIndex	Integer	The position in the sequence memory where the sequence starts from in FrameSequence/ContinuousFrameSequence mode.
SoftwareVersion	String	Returns the version of the SDK.
TriggerMode	Enumerated	Allows the user to configure the device trigger mode at a high level. <u>Options:</u> • **InternalExpose** (default value) A mask is exposed for limited time specified by ExposureTime feature. The exposure is triggered by executing Expose command feature. This mode is available for all OperationMode options. • **InternalSoftware** A mask is exposed for unlimited time. The exposure is triggered by executing Expose command feature. Sending Abort command stops the current mask exposure. This mode is available for Live/FrameMemory option only. • **ExternalExpose** A mask/sequence is exposed for limited time specified by ExposureTime feature**.** The mask exposure is triggered by a signal sent by an external hardware trigger. This mode is available for all OperationMode options. • **ExternalSequenceStart** A defined sequence of masks is exposed from the sequence memory**.** The exposure is triggered by a signal sent by an external hardware trigger. This mode is available for FrameSequence/ContinuousFrameSequence option only. • **ExternalBulb** A mask is exposed by a signal sent by an external hardware trigger for as long as this signal is 'High'. This mode is available for all OperationMode options.
UploadFrame	Command	Uploads a single mask from the user buffer to the frame memory.

UploadGrayScaleImage	Command	Uploads grayscale sequence of bit planes to the frame memory and the sequence memory. The number of bit planes is specified by GrayScaleSequenceEncoding feature. Each bit plane is uploaded to the frame memory and then to the sequence memory with a different exposure time for each bit plane to simulate the grayscale effect.
UploadSequenceEvent	Command	Uploads a single sequence event to the sequence memory.

SECTION 4 - Tutorial

This section will explain how to develop a program to use the API to communicate with a Mosaic device. It will point out the critical parts that the program must have and provide a foundation to build more complicated programs in the future.

The first thing that must be done is to add the appropriate library file to the project. During the SDK3 installation a Borland library (atcore.lib) and a Microsoft library (atcorem.lib) are made available. You should also add the include path of the SDK3 installation directory to your project. The atcore.h file from this directory will need to be included in the main project file.

The very first API call must be AT_InitialiseLibrary, and the very last call must be AT_FinaliseLibrary. These functions will prepare the API for use and free resources when no longer needed.

```
AT_InitialiseLibrary( );
```

```
AT_FinaliseLibrary( );
```

Every function will return an error code when called. It is recommended that a user check every return code before moving on to the next statement. If the AT_InitialiseLibrary function call fails there is no point continuing on with the program as every proceeding function call will also fail.
Every function call will return an Integer and each return code that could possibly be returned is listed in the atcore.h file and documented in the Error Codes section.

So now the program becomes:

```
int i_returnCode;
i_returnCode = AT_InitialiseLibrary( );
if (i_returnCode == AT_SUCCESS) {
  //continue with program
}
i_returnCode = AT_ FinaliseLibrary( );
if (i_returnCode != AT_SUCCESS) {
  //Error FinaliseLibrary
}
```

Error Checking

It is highly recommended that you check return codes from every function in case of error. For the purpose of this tutorial the error checking will be kept to a minimum to reduce the length of the program.

The next stage of the program is to get a handle to the device that is to be controlled. This is done with the AT_Open function and there is a corresponding AT_Close function to release the device handle. A device index is passed into the AT_Open function specifying which device you wish to open.

Now the program is :

```
int i_returnCode;
AT_H Hndl;
int i_cameraIndex = 0;
i_returnCode = AT_InitialiseLibrary( );
if (i_returnCode == AT_SUCCESS) {
  i_returnCode = AT_Open ( i_cameraIndex, &Hndl );
  if (i_returnCode == AT_SUCCESS) {
    //continue on with program
    //.........
    i_returnCode = AT_Close ( Hndl );
    if (i_returnCode != AT_SUCCESS) {
      // error closing handle
    }
  }
}
i_returnCode = AT_FinaliseLibrary( );
if (i_returnCode != AT_SUCCESS) {
  //Error FinaliseLibrary
}
```

From here on example code will assume that we have successfully got a handle to our device, following on from the previous code. Now we will add code to the program to modify and view the “ExposureTime” setting of the device.

It is recommended that any features that need to set or view that they should have their accessibility checked first. Every feature can be checked for being implemented (AT_IsImplemented), writable (AT_IsWritable), readable (AT_IsReadable) and read only(AT_IsReadOnly)

For this tutorial it is assumed that the ‘ExposureTime’ feature is writable and readable, to reduce code complexity.

```
double d_newExposure = 0.02;
i_returnCode = AT_SetFloat ( Hndl,  L"ExposureTime", d_newExposure);
if (i_returnCode == AT_SUCCESS) {
  //it has been set

}
```

NOTE: It is recommended that when using the AT_SetFloat functions that an AT_GetFloat is done afterwards to get the actual value that the device will use as it may not be exactly what was set.

All other settings we will take as default.

This stage will show how to get data from a single acquisition. In preparation the program must allocate memory to store the acquired image in. To get the size of memory to be declared use the integer “ImageSizeBytes” feature. No error checking will be shown in the following example to help with clarity but it is recommended that in final code all return codes are checked.

The next stage is to let the SDK know what memory to use for the upcoming frame exposure. This is done with the AT_QueueBuffer API function call. Only one buffer can be queued to the SDK before a frame exposure starts.
In case multiple buffers are queued then the last buffer added to the queue will be used to expose a frame.

```
AT_QueueBuffer(Hndl,  pucAlignedBuffer, ImageSizeBytes);
```

Now start the mask exposure with the Command “Expose”.

```
AT_Command(Hndl, L"Expose");
```

The mask will only be displayed if nothing is currently exposed on device. The command to stop the current exposure is "Abort".

```
AT_Command(Hndl, L"Abort");
```

The final code for this tutorial can be seen in section “Code Listing for Tutorial” in the Appendix

4.1 – Main Functionality

4.1.1 – Frame Memory

One of the operating modes of Mosaic 3 is the frame memory. It allows to store up to 139 masks in the Mosaic 3's internal RAM memory for as long as the device is connected to the power.

In order to store a single mask to the frame memory UploadFrame command should to be called.

```
AT_Command(Handle, L"UploadFrame");
```

The command will copy the current mask from the user defined buffer to the first position available in the memory starting from 0 up the maximum value of the FrameMemoryCount feature. The function returns AT_ERR_NOTWRITABLE if the number of uploaded masks reached the memory limit. The full list of the return codes for that feature can be found at the end of this paragraph.

The frame memory allows 2 operations:

- Uploading a frame using UploadFrame feature
- Clear the whole frame memory using ClearFrameMemory feature

In order to clear the frame memory ClearFrameMemory command should to be called.

```
AT_Command(Handle, L"ClearFrameMemory");
```

The ClearFrameMemory command will return AT_ERR_NOTWRITABLE if the device is currently exposing a frame or the sequence memory is not empty. The sequence memory has to be cleared first as the uploaded sequence events refer to indexes of masks stored in the frame memory.

The uploaded masks can be transferred from the frame memory and displayed on the device. An individual mask can be accessed in the memory by specifying its index using FrameIndex feature.

In order to use the frame memory to display masks a user would need to set OperationMode feature to FrameMemory mode as following:

```
AT_SetEnumString(Handle, L"OperationMode", L"FrameMemory");
```

The following code will set the current mask index to the first index and display the mask at that index using Expose feature:

```
AT_SetEnumString(Handle, L"OperationMode", L"FrameMemory");
```

```
AT_SetInt(Handle, L"FrameIndex", 0);
AT_Command(Handle, L"Expose");
```

NOTE: FrameIndex feature uses zero-based numbering system.

The FrameIndex command will return error code AT_ERR_NOTWRITABLE if a user tries to access an index of a mask that hasn't been uploaded to the frame memory yet. The user can check how many masks are currently in the frame memory using FrameMemoryCount as following:

```
AT_GetInt(Handle, L"FrameMemoryCount", &frameMemoryCount);
```

Find below the available return codes and their values for UploadFrame feature.

UploadFrame	
AT_SUCCESS (0)	Function call has been successful
AT_ERR_NOTWRITABLE (5)	The number of uploaded masks reached the memory limit
AT_ERR_NOTWRITABLE (5)	Feature is temporarily not writable because a mask is currently being exposed on the device
AT_ERR_COMM (17)	An error has occurred while communicating with hardware

4.1.2 – Sequence Memory

The sequence memory allows a user to create and store sequence events in the memory for as long as the device is connected to the power.

In order to upload a new sequence event to the sequence memory the UploadSequenceEvent command should to be called:

```
AT_Command(Handle, L" UploadSequenceEvent");
```

UploadSequenceEvent command creates a new sequence event and appends it at the first position available in the sequence memory starting from 0 up the memory limit indicated by the maximum value of SequenceEventCount feature.

In order to upload a new sequence the following features have to be set:

- FrameIndex
- FrameCount
- FrameCycleCount
- SequenceExposureTime
- SequenceGapTime

These features have to be set up individually for each sequence, for example:

```
AT_SetInt(Handle, L"FrameIndex", 0);
AT_SetInt(Handle, L"FrameCount", 1);
AT_SetInt(Handle, L"FrameCycleCount", 1);
AT_SetFloat(Handle, L"SequenceExposureTime", 0.5);
AT_SetFloat(Handle, L"SequenceGapTime", 0.5);
AT_Command(Handle, L"UploadSequenceEvent");
```

The code above will create a single sequence event in the sequence memory using one frame at index 0 of the frame memory with repeat count equal to 1. The function returns AT_ERR_NOTWRITABLE if the number of uploaded sequence events reached the memory limit. The full list of the return codes for that feature can be found at the end of this paragraph.

The sequence memory allows 2 operations:

- Uploading a sequence event using UploadSequenceEvent feature
- Clear the whole sequence memory using ClearSequenceMemory feature

In order to clear the sequence memory ClearSequenceMemory command should to be called.

```
AT_Command(Handle, L"ClearSequenceMemory");
```

The ClearSequenceMemory command will return AT_ERR_NOTWRITABLE is the device is currently exposing a frame or a sequence.

In order to run a sequence the OperationMode feature has to be set to FrameSequence or ContinuousFrameSequence and then Expose command should to be called. It will run a sequence or series of sequences based the selected option of the following features:

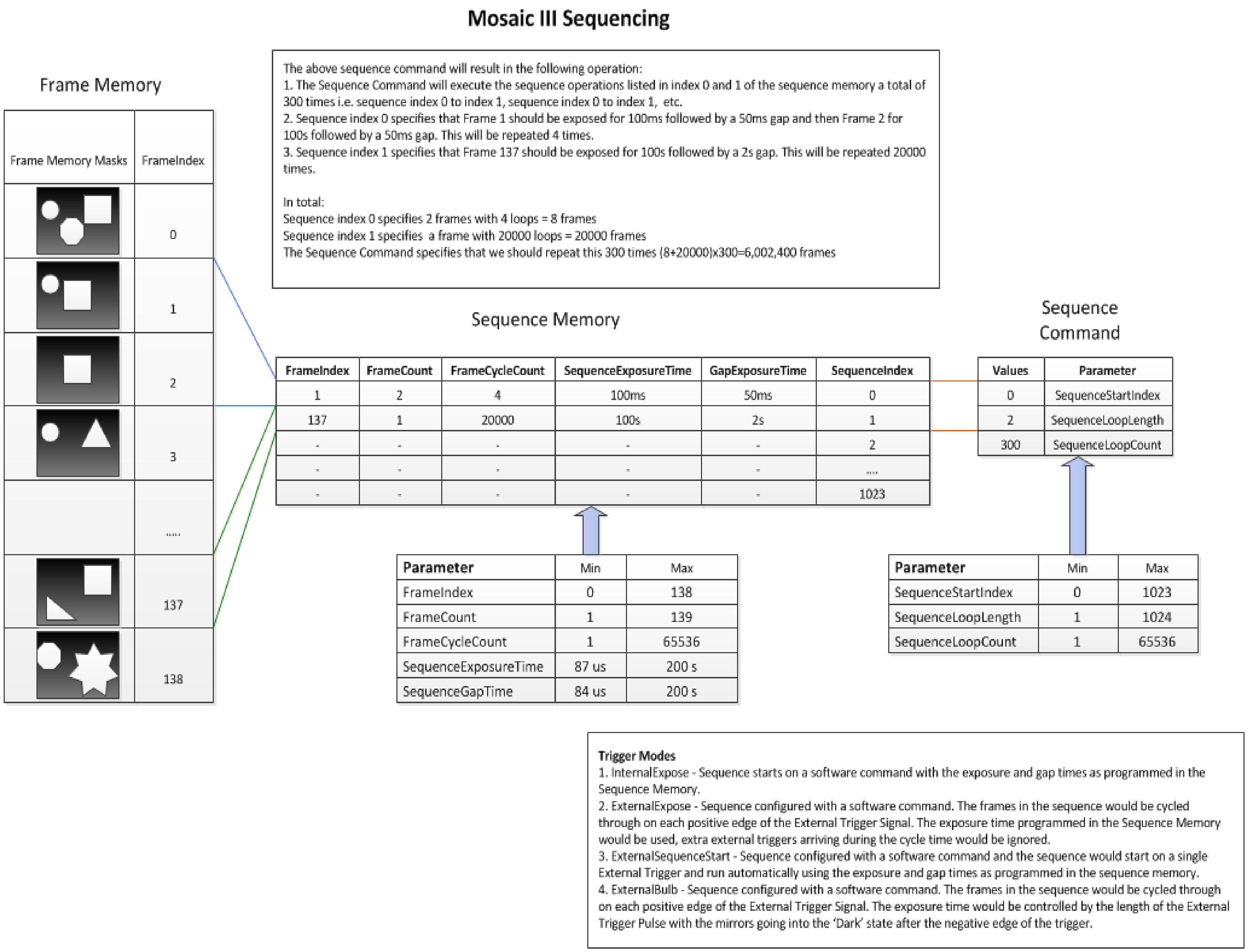
- SequenceStartIndex
- SequenceLoopCount
- SequenceLoopLength

Below is a code example to run a single sequence at index 0 of the sequence memory once.

```
AT_SetEnumString(Handle, L"OperationMode", L"FrameSequence");
AT_SetInt(Handle, L"SequenceStartIndex", 0);
AT_SetInt(Handle, L"SequenceLoopCount", 1);
AT_SetInt(Handle, L"SequenceLoopLength", 1);
AT_Command(Handle, L"Expose");
```

NOTE: SequenceStartIndex feature uses zero-based numbering system.

The diagram below shows how the frame sequencing works in Mosaic III.



Mosaic III Sequencing

The above sequence command will result in the following operation:
1. The Sequence Command will execute the sequence operations listed in index 0 and 1 of the sequence memory a total of 300 times i.e. sequence index 0 to index 1, sequence index 0 to index 1, etc.
2. Sequence index 0 specifies that Frame 1 should be exposed for 100ms followed by a 50ms gap and then Frame 2 for 100s followed by a 50ms gap. This will be repeated 4 times.
3. Sequence index 1 specifies that Frame 137 should be exposed for 100s followed by a 2s gap. This will be repeated 20000 times.

In total:
Sequence index 0 specifies 2 frames with 4 loops = 8 frames
Sequence index 1 specifies a frame with 20000 loops = 20000 frames
The Sequence Command specifies that we should repeat this 300 times (8+20000)x300=6,002,400 frames

Frame Memory

Frame Memory Masks	FrameIndex
	0
	1
	2
	3
	.....
	137
	138

Sequence Memory

FrameIndex	FrameCount	FrameCycleCount	SequenceExposureTime	GapExposureTime	SequenceIndex
1	2	4	100ms	50ms	0
137	1	20000	100s	2s	1
-	-	-	-	-	2
-	-	-	-	-	....
-	-	-	-	-	1023

Parameter	Min	Max
FrameIndex	0	138
FrameCount	1	139
FrameCycleCount	1	65536
SequenceExposureTime	87 us	200 s
SequenceGapTime	84 us	200 s

Sequence Command

Values	Parameter
0	SequenceStartIndex
2	SequenceLoopLength
300	SequenceLoopCount

Parameter	Min	Max
SequenceStartIndex	0	1023
SequenceLoopLength	1	1024
SequenceLoopCount	1	65536

Trigger Modes
1. InternalExpose - Sequence starts on a software command with the exposure and gap times as programmed in the Sequence Memory.
2. ExternalExpose - Sequence configured with a software command. The frames in the sequence would be cycled through on each positive edge of the External Trigger Signal. The exposure time programmed in the Sequence Memory would be used, extra external triggers arriving during the cycle time would be ignored.
3. ExternalSequenceStart - Sequence configured with a software command and the sequence would start on a single External Trigger and run automatically using the exposure and gap times as programmed in the sequence memory.
4. ExternalBulb - Sequence configured with a software command. The frames in the sequence would be cycled through on each positive edge of the External Trigger Signal. The exposure time would be controlled by the length of the External Trigger Pulse with the mirrors going into the 'Dark' state after the negative edge of the trigger.

Find below the available return codes and their values for UploadSequenceEvent feature.

UploadSequenceEvent	
`AT_SUCCESS (0)`	Function call has been successful
`AT_ERR_NOTWRITABLE (5)`	The number of uploaded sequence events reached the memory limit
`AT_ERR_NOTWRITABLE (5)`	Feature is temporarily not writable because a mask is currently being exposed on the device
`AT_ERR_NOTWRITABLE (5)`	Adding new sequence event is not possible due to incorrect parameters (FrameIndex, FrameCount or FrameCycleCount)
`AT_ERR_COMM (17)`	An error has occurred while communicating with hardware

4.1.3 –Grayscale Sequencing

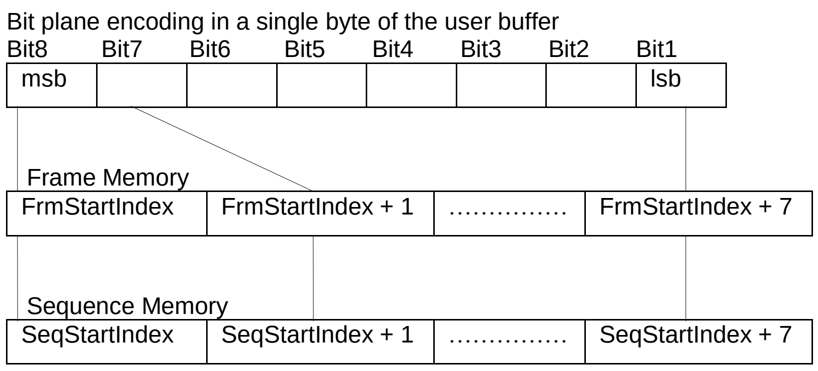
The Grayscale Sequencing functionality allows a user to display a sequence of masks to achieve grayscale effect. The individual bit planes for the grayscale sequence have to be stored in the frame memory and the appropriate sequence events created in the sequence memory.

In order to upload a grayscale sequence the UploadGrayScaleImage command should to be called

```
AT_Command(Handle, L"UploadGrayScaleImage");
```

It uploads the grayscale sequence of bit planes to the frame memory and the sequence memory. The number of bit planes to upload is specified by GrayScaleSequenceEncoding feature. Each single byte in the user buffer can contain the 8-bit, 6-bit, or 4-bit planes. All bytes in the user buffer must have the same number of bit planes for a single grayscale sequence upload. The planes in a single byte will be uploaded staring from the left (Bit8, Bit6, Bit4 depending on GrayScaleSequenceEncoding selected option) towards the least significant bit (lsb).

The diagram below shows how the bit planes are uploaded from the user buffer to the frame and sequence memory for GrayScaleSequenceEncoding set to 8BitDIP



For each bit plane uploaded to the frame memory there is a sequence event created in the sequence memory. It contains an index of a single bit plane in the frame memory and a calculated exposure time that the bit plane will be exposed at when running the sequence of them to simulate the grayscale effect. The exposure time is calculated internally be the SDK based on the overall exposure for the grayscale sequence which can be set by the GrayScaleExposureTime feature.

The following code example uploads 8 bit planes from the user buffer to the frame memory and creates 8 sequence events in the sequence memory to simulate a single grayscale image. The planes will be appended to the frame memory and the sequence events will be appended to the sequence memory. The user will have make sure there is enough free space available for uploaded bit planes otherwise the UploadGrayScaleImage command returns AT_ERR_NOTWRITABLE error code.
The exposure time for a single grayscale sequence cycle is set to 0.2 seconds.

```
AT_SetFloat(Handle, L"GrayScaleExposureTime", 0.2);
AT_SetEnumString(Handle, L"GrayScaleSequenceEncoding", L"8BitDIP");
AT_Command(Handle, L"UploadGrayScaleImage");
```

In order to display the grayscale sequence the following features have to be set:

- SequenceStartIndex set to the index of the first bit plane in the sequence memory
- SequenceLoopLength set to the number of bit planes to cycle through
- SequenceLoopCount set to the number of grayscale sequence repeats if the OperationMode set to FrameSequence. This setting is ignored if a sequence is displayed in ContinuousFrameSequence mode
- OperationMode set to FrameSequence/ContinuousFrameSequence
- Expose command executed

Below is a code example of displaying the grayscale image:

```
AT_SetInt(Handle, L"SequenceStartIndex", 0);
AT_SetInt(Handle, L"SequenceLoopLength", 8);
AT_SetEnumString(Handle, L"OperationMode", L"ContinuousFrameSequence");
AT_Command(Handle, L"Expose");
```

It would cycle through 8 consecutive sequence events repeatedly starting from the first index in the sequence memory giving the impression of the grayscale image displayed on the device.
In order to stop running the sequence the Abort command needs to be executed.

4.1.4 – Trigger Modes

The TriggerMode feature allows the user to configure the device trigger mode for exposing a frame or sequence of frames.

The list of all modes for TriggerMode feature:

- InternalExpose
- InternalSoftware
- ExternalExpose
- ExternalSequenceStart
- ExternalBulb

The availability of each mode depends on the currently selected OperationMode option. If the trigger mode is not implemented by the device for the selected OperationMode option then any attempt to access that trigger mode will return the error code AT_ERR_STRINGNOTAVAILABLE (if calling `AT_SetEnumString`) or AT_ERR_INDEXNOTAVAILABLE (if calling `AT_SetEnumIndex`), e.g. if the user tries to set the TriggerMode feature to ExternalSequenceStart in Live or FrameMemory option of OperationMode feature then the error code AT_ERR_STRINGNOTAVAILABLE will be returned. Refer to the Feature Reference section for information on all available modes.

The InternalExpose mode is a default TriggerMode setting for all available options of the OperationMode feature. If a user changes the current TriggerMode value it will only affect the currently selected option of OperationMode feature, e.g. if the TriggerMode is changed to InternalSoftware the TriggerMode is still set to InternalExpose for any other OperationMode option. Any changes to TriggerMode will only be applicable to the current OperationMode value so if the OperationMode changes to the different mode the TriggerMode value also changes to the value used when the new selected OperationMode option was used last time.
In order to make sure the correct trigger mode will be used it is recommended to set the TriggerMode explicitly every time the OperationMode changes, e.g.:

```
AT_SetEnumString(Handle, L"OperationMode", L"FrameSequence");
AT_SetEnumString(Handle, L"TriggerMode", L"ExternalSequenceStart");
```

In Live or FrameSequence/ContinuousFrameSequence mode to expose a frame or a sequence of frames the Expose command should to be called for all available TriggerMode options.

```
AT_Command(Handle, L"Expose");
```

Depending on the selected TriggerMode option the Expose command will expose a frame immediately if the TriggerMode is set to one of the internal modes (InternalExpose or InternalSoftware) or the device will be waiting for an external signal to expose a frame if the TriggerMode is set to one of the external modes (ExternalExpose, ExternalSeqneceStart or ExternalBulb).

In FrameMemory mode the Expose command works slightly different. It should only be called if the trigger mode is set to one of the internal modes (InternalExpose or InternalSoftware). In the external trigger mode (ExternalExpose or ExternalBulb) the device doesn't need to wait for the execution of the Expose command

and it will expose a frame indicated by the current value of FrameIndex feature or an external electric switch (see FrameIndexSource feature for more details) when externally triggered.

4.2.1 – Initialise Library and Open Device

```
//InitialiseLibrary must be the first function call made by an application before
accessing other functions
AT_InitialiseLibrary();

//Declare an Andor Device Handle for referencing device later
AT_H Handle;

//Open the first device (Device 0)
AT_Open(0, &Handle);

//Close the device when finished using it
AT_Close(Handle);

//Call FinaliseLibrary when all access to API is complete
AT_FinaliseLibrary();
```

4.2.2 - Simple Single Frame Display

```
#include "atcore.h"
#include <iostream>
using namespace std;

int main(int argc, char* argv[])
{
  AT_InitialiseLibrary();
  AT_H Handle;
  AT_Open(0, &Handle);

  //Set the exposure time for this device to 5 seconds
  AT_SetFloat(Handle, L"ExposureTime", 5);

  //Get the sensor width
  AT_64 i64_sensorWidth;
  AT_GetInt(Handle, L"SensorWidth", &i64_sensorWidth);

  //Get the sensor height
  AT_64 i64_sensorHeight;
  AT_GetInt(Handle, L"SensorHeight", &i64_sensorHeight);

  //Get the number of bytes required to store one frame
  AT_64 ImageSizeBytes;
  AT_GetInt(Handle, L"ImageSizeBytes", &ImageSizeBytes);

  int BufferSize = static_cast<int>(ImageSizeBytes);

  //Allocate a memory buffer to store one frame
  unsigned char* UserBuffer = new unsigned char[BufferSize];

  //Pass this buffer to the SDK
  AT_QueueBuffer(Handle, UserBuffer, BufferSize);

  //Abort the currently displayed image on device (if any)
  AT_Command(Handle, L"Abort");

  //Clear the user buffer
  memset(UserBuffer, 0, BufferSize);

  //Populate UserBuffer with the rectangle shape
  for (int y = 100; y < i64_sensorHeight - 100; y++) {
    for (int x = 100; x < i64_sensorWidth - 100; x++) {
      UserBuffer[y*i64_sensorWidth + x] = 1;
    }
  }

  //Start exposing the rectangle shape
  AT_Command(Handle, L"Expose");

  AT_BOOL IsExposing;
  do {
    AT_GetBool(Handle, L"IsExposing", &IsExposing);
  } while (IsExposing == TRUE);

  //Application specific data processing goes here...

  //Free the allocated buffer
  delete [] UserBuffer;

  AT_Close(Handle);
  AT_FinaliseLibrary();

  return 0;
}
```

4.2.3 - Using a Feature

```
AT_InitialiseLibrary();
AT_H Handle;
AT_Open(0, &Handle);

AT_BOOL Implemented;
//To determine if Exposure time is implemented by the device
AT_IsImplemented(Handle, L"ExposureTime", &Implemented);

AT_BOOL ReadOnly;
//To determine if Exposure time a Read Only Feature
AT_IsReadOnly(Handle, L"ExposureTime", &ReadOnly);

if (Implemented==AT_TRUE) {
  //Get the Limits for Exposure Time
  double Min, Max;
  AT_GetFloatMin(Handle, L"ExposureTime", &Min);
  AT_GetFloatMax(Handle, L"ExposureTime", &Max);

  //Get the current accessibility
  AT_BOOL Writable, Readable;
  AT_IsWritable(Handle, L"ExposureTime", &Writable);
  AT_IsReadable(Handle, L"ExposureTime", &Readable);

  if (Readable==AT_TRUE) {
    //To get the current value of Exposure time in
    //microseconds
    double ExposureTime;
    AT_GetFloat(Handle, L"ExposureTime", &ExposureTime);
  }

  if (Writable==AT_TRUE) {
    //To set the value of Exposure Time to 100
    //microseconds
    AT_SetFloat(Handle, L"ExposureTime", 0.0001);
  }

}

AT_Close(Handle);
AT_FinaliseLibrary();
```

4.2.4 – Pixel Encoding

```
AT_InitialiseLibrary();
AT_H Handle;
AT_Open(0, &Handle);

//Get the sensor width
AT_64 i64_sensorWidth;
AT_GetInt(Handle, L"SensorWidth", &i64_sensorWidth);

//Get the sensor height
AT_64 i64_sensorHeight;
AT_GetInt(Handle, L"SensorHeight", &i64_sensorHeight);

//Set PixelEncoding  to Mono1Packed - A single byte in the user buffer stores 8
consecutive pixel values
AT_SetEnumString(Handle, L"PixelEncoding", L"Mono1Packed");

//Set TriggerMode to InternalSoftware - A frame is displayed for unlimited time
AT_SetEnumString(Handle, L"TriggerMode", L"InternalSoftware");

//Get the number of bytes required to store one frame
AT_64 ImageSizeBytes;
AT_GetInt(Handle, L"ImageSizeBytes", &ImageSizeBytes);

int BufferSize = static_cast<int>(ImageSizeBytes);

//Allocate a memory buffer to store one frame
unsigned char* UserBuffer = new unsigned char[BufferSize];

memset(UserBuffer, 0, ImageSizeBytes);

//Pass this buffer to the SDK
AT_QueueBuffer(Handle, UserBuffer, BufferSize);

//Abort the currently displayed image on device (if any)
AT_Command(Handle, L"Abort");

//Populate UserBuffer with the rectangle shape
for (int y = 100; y < i64_sensorHeight - 100; y++) {
  for (int x = 100; x < i64_sensorWidth - 100; x++) {
    //Pack the frame
    int i_index = (y*(int)i64_sensorWidth + x )/ 8;
    int i_shift = 7 - (x % 8);
    UserBuffer[i_index] |= ( 1 << i_shift );
  }
}

//Start exposing the rectangle shape
AT_Command(Handle, L"Expose");

//Application specific data processing goes here...
Sleep(5000);

//Abort the currently displayed image on device
AT_Command(Handle, L"Abort");

//Free the allocated buffer
delete [] UserBuffer;

AT_Close(Handle);
AT_FinaliseLibrary();
```

4.2.5 – Call-Backs

```
//This example will demonstrate how to setup, register and unregister a call-back
//Tests of the correct call-back context and a count of the number of updates
received are provided

AT_H Handle;
int g_iCallbackCount = 0;
int g_iCallbackContext = 0;

int AT_EXP_CONV Callback(AT_H Hndl, const AT_WC* Feature, void* Context)
{
  //Application specific call-back handling should go here
  g_iCallbackCount++;
  g_iCallbackContext = *reinterpret_cast<int*>(Context);
  return AT_CALLBACK_SUCCESS;
}

int main(int argc, char* argv[])
{
  AT_InitialiseLibrary();
  AT_Open(0, &Handle);

  //Set the call-back context, context values can be defined on per application
basis
  int i_callbackContext = 5;

  //Reset the call-back count
  //Only required for the purposes of this example to show the call-back has been
received
  g_iCallbackCount = 0;

  //Register a call-back for the given feature
  AT_RegisterFeatureCallback(Handle, L"ExposureTime", Callback,
(void*)&i_callbackContext);

  //Set the feature in order to trigger the call-back
  AT_SetFloat(Handle, L"ExposureTime", 0.0001);

  // Application specific code should go here
  //For this example we shall check that the call-back has been successful
  if (g_iCallbackCount==0 || g_iCallbackContext != i_callbackContext) {
    //Deal with failed call-back
  }

  //Unregister the call-back, no more updates will be received
  AT_UnregisterFeatureCallback(Handle, L"ExposureTime", Callback,
(void*)&i_callbackContext);

  AT_Close(Handle);
  AT_FinaliseLibrary();
}
```

4.2.6 – Frame Sequencing

```
#include "atcore.h"
#include <iostream>
using namespace std;

//This example will demonstrate how to setup and upload frames into Frame Memory
and sequence events to the Sequence Memory

AT_H Handle;
AT_64 i64_sensorWidth;
AT_64 i64_sensorHeight;
int returnCode;

unsigned char* UserBuffer;
int BufferSize;

void ClearFrameMemory()
{
  AT_Command(Handle, L"ClearSequenceMemory");
  AT_Command(Handle, L"ClearFrameMemory");
}

void CreateFrame(int frameIndex)
{
  memset(UserBuffer, 0, BufferSize );
  //Put a code to create a shape for an individual frame
  for (int y = 20*frameIndex; y < i64_sensorHeight - 20*frameIndex; y++) {
    for (int x = 20*frameIndex; x < i64_sensorWidth - 20*frameIndex; x++) {
      UserBuffer[y*i64_sensorWidth + x] = 1;
    }
  }
}

void UploadFrames()
{
  for (int i = 0; i < 10; i++)
  {
    //Call user-defined function
    CreateFrame(i);
    AT_Command(Handle, L"UploadFrame");
  }
}

void UploadSequenceEventEvents()
{
   AT_Command(Handle, L"ClearSequenceMemory");
   //Sequence 1 setup
   AT_SetInt(Handle, L"FrameIndex", 0);
   AT_SetInt(Handle, L"FrameCount", 1);
   AT_SetInt(Handle, L"FrameCycleCount", 1);
   AT_SetFloat(Handle, L"SequenceExposureTime", 0.5);
   AT_SetFloat(Handle, L"SequenceGapTime", 0.5);
   returnCode = AT_Command(Handle, L"UploadSequenceEvent");
   if (returnCode != 0)
   {
     cout << "Error occured during uploading a sequence event" << endl;
   }

   //Sequence 2 setup
   AT_SetInt(Handle, L"FrameIndex", 5);
   AT_SetInt(Handle, L"FrameCount", 2);
   AT_SetInt(Handle, L"FrameCycleCount", 3);
   AT_SetFloat(Handle, L"SequenceExposureTime", 0.5);
   AT_SetFloat(Handle, L"SequenceGapTime", 0.5);
   returnCode = AT_Command(Handle, L"UploadSequenceEvent");
   if (returnCode != 0)
   {
     cout << "Error occured during uploading a sequence event" << endl;
   }
}

void DisplayFrame(int frameIndex, double exposureTime)
{
  // Set OperationMode to FrameMemory
  AT_SetEnumString(Handle, L"OperationMode", L"FrameMemory");
  AT_SetInt(Handle, L"FrameIndex", frameIndex);
  AT_SetFloat(Handle, L"ExposureTime", exposureTime);
  AT_Command(Handle, L"Abort");
  returnCode = AT_Command(Handle, L"Expose");
  if (returnCode != 0)
  {
    cout << "Error occured during exposing a frame" << endl;
  }
  AT_BOOL IsExposing;
  do {
    AT_GetBool(Handle, L"IsExposing", &IsExposing);
  }
  while (IsExposing == TRUE);
}

void DisplaySequence()
{
  // Set OperationMode to FrameSequence
  returnCode = AT_SetEnumString(Handle, L"OperationMode", L"FrameSequence");
  returnCode = AT_SetInt(Handle, L"SequenceStartIndex", 0);
  returnCode = AT_SetInt(Handle, L"SequenceLoopCount", 2);
  returnCode = AT_SetInt(Handle, L"SequenceLoopLength", 2);

  returnCode = AT_Command(Handle, L"Abort");
  returnCode = AT_Command(Handle, L"Expose");
  if (returnCode != 0)
  {
    cout << "Error occured during exposing a frame" << endl;
  }
  AT_BOOL IsExposing;
  do {
    AT_GetBool(Handle, L"IsExposing", &IsExposing);
  }
  while (IsExposing == TRUE);
}

int main(int argc, char* argv[])
{
  AT_InitialiseLibrary();
  AT_Open(0, &Handle);

  //Get the sensor width
  AT_GetInt(Handle, L"SensorWidth", &i64_sensorWidth);

  //Get the sensor height
  AT_GetInt(Handle, L"SensorHeight", &i64_sensorHeight);

  AT_64 ImageSizeBytes;
  //Get the number of bytes required to store one frame
  AT_GetInt(Handle, L"ImageSizeBytes", &ImageSizeBytes);

  BufferSize = static_cast<int>(ImageSizeBytes);

  //Allocate a memory buffer to store one frame
  UserBuffer = new unsigned char[BufferSize];

  //Pass this buffer to the SDK
  AT_QueueBuffer(Handle, UserBuffer, BufferSize);

  //Clear existing frames in memory
  ClearFrameMemory();

  //Upload user-defined frames to Frame Memory
  UploadFrames();

  //Display frames 0..10
  for (int i = 0 ; i < 10; i++)
  {
    DisplayFrame(i, 1);
  }

  //Upload defined sequence events
  UploadSequenceEventEvents();

  //Display sequence
  DisplaySequence();

  delete [] UserBuffer;

  AT_Close(Handle);
  AT_FinaliseLibrary();
}
```

5.1 Function Listing

5.1.1 – AT_Open

```
int AT_Open(int DeviceIndex, AT_H* Handle)
```

Description

This function is used to open up a handle to a particular device. The DeviceIndex parameter indicates the index of the device that you wish to open and the handle to the device is returned in the Handle parameter. This Handle parameter must be passed as the first parameter to all other functions to access the features or to acquire data from the device.

5.1.2 – AT_Close

```
int AT_Close(AT_H Handle)
```

Description

This function is used to close a previously opened handle to a device. The Handle parameter is the handle that was returned from the AT_Open function. The function should be called when you no longer wish to access the device from your application usually at shutdown.

5.1.3 – AT_IsImplemented

```
int AT_IsImplemented(AT_H Hndl, AT_WC* Feature, AT_BOOL* Implemented)
```

Description

This function can be used to determine whether the device has implemented the feature specified by the Feature parameter. On return the Implemented parameter will contain the value AT_FALSE or AT_TRUE. In the case that the feature is implemented the value of Implemented will be AT_TRUE, otherwise it will be AT_FALSE.

5.1.4 – AT_IsReadOnly

```
int AT_IsReadOnly(AT_H Hndl, AT_WC* Feature, AT_BOOL* ReadOnly)
```

Description

This function can be used to determine whether the feature specified by the Feature parameter can be modified. On return the ReadOnly parameter will contain the value AT_FALSE or AT_TRUE. In the case that the feature can be modified the value of ReadOnly will be AT_TRUE, otherwise it will be AT_FALSE.

5.1.5 – AT_IsWritable

```
int AT_IsWritable(AT_H Hndl, AT_WC* Feature, AT_BOOL* Writable)
```

Description

This function can be used to determine whether the feature specified by the Feature parameter can currently be modified. On return the Writable parameter will contain the value AT_FALSE or AT_TRUE. In the case that the feature is currently writable the value of Writable will be AT_TRUE, otherwise it will be AT_FALSE. This function differs from the AT_IsReadOnly function in that a feature that is not writable may only be temporarily unavailable for modification because of the values of other features, whereas a feature that is read only is permanently un-modifiable.

5.1.6 – AT_IsReadable

```
int AT_IsReadable(AT_H Hndl, AT_WC* Feature, AT_BOOL* Readable)
```

Description

This function can be used to determine whether the feature specified by the Feature parameter can currently be read. On return the Readable parameter will contain the value AT_FALSE or AT_TRUE. In the case that the feature is currently readable the value of Readable will be AT_TRUE, otherwise it will be AT_FALSE. A feature may become unavailable for reading based on the value of other features.

5.1.7 – AT_RegisterFeatureCallback

```
int AT_RegisterFeatureCallback(AT_H Hndl, AT_WC* Feature,
                               FeatureCallback EvCallback, void* Context)
```

Description

To retrieve a notification each time the value or other properties of a feature changes you can use this function to register a callback function. The Feature that you wish to receive notifications for is passed into the function along with the function that you wish to get called. The fourth parameter is a caller defined parameter that can be used to provide contextual information when the callback is called. The callback function should have the signature shown below.

```
      int MyFunction(AT_H Hndl, AT_WC* Feature, void* Context)
```

When called, the Feature that caused the callback is returned which allows you to use a single callback function to handle multiple features. The context parameter is the same as that used when registering the callback and is sent unmodified. As soon as this callback is registered a single callback will be made immediately to allow the callback handling code to perform any Initialisation code to set up monitoring of the feature.

5.1.8 – AT_UnregisterFeatureCallback

```
int AT_UnregisterFeatureCallback(AT_H Hndl, AT_WC* Feature,
                                 FeatureCallback EvCallback, void* Context)
```

Description

This function is used to un-register a callback function previously registered using AT_RegisterFeatureCallback. The same parameters that were passed to the register function should be passed to this unregister function. Once this function is called, no more callbacks will be sent to this callback function for the specified Feature.

5.1.9 – AT_InitialiseLibrary

```
int AT_InitialiseLibrary()
```

Description

This function is used to prepare the SDK internal structures for use and must be called before any other SDK functions have been called.

5.1.10 – AT_FinaliseLibrary

```
int AT_FinaliseLibrary()
```

Description

This function will free up any resources used by the SDK and should be called whenever the program no longer needs to use any SDK functions. AT_InitialiseLibrary may be called again later by the same process if device control is again required.

5.1.11 – AT_SetInt

```
int AT_SetInt(AT_H Hndl, AT_WC* Feature, AT_64 Value)
```

Description

This function will modify the value of the specified feature, if the feature is of integer type. The function will return an error if the feature is read only or currently not writable or if the feature is either not an integer feature or is not implemented by the device.

5.1.12 – AT_GetInt

```
int AT_GetInt(AT_H Hndl, AT_WC* Feature, AT_64 * Value)
```

Description

The function will return the current value for the specified feature. The function will return an error if the feature is currently not readable or if the specified feature is either not an integer feature or is not implemented by the device.

5.1.13 – AT_GetIntMax

```
int AT_GetIntMax(AT_H Hndl, AT_WC* Feature, AT_64 * MaxValue)
```

Description

This function will return the maximum allowable value for the specified integer type feature.

5.1.14 – AT_GetIntMin

```
int AT_GetIntMin(AT_H Hndl, AT_WC* Feature, AT_64 * MinValue)
```

Description

This function will return the minimum allowable value for the specified integer type feature.

5.1.15 – AT_SetFloat

```
int AT_SetFloat(AT_H Hndl, AT_WC* Feature, double Value)
```

Description

This function will modify the value of the specified feature, if the feature is of float type. The function will return an error if the feature is read only or currently not writable or if the feature is either not a float type feature or is not implemented by the device.

5.1.16 – AT_GetFloat

```
int AT_GetFloat(AT_H Hndl, AT_WC* Feature, double * Value)
```

Description

The function will return the current value for the specified feature. The function will return an error if the feature is currently not readable or if the specified feature is either not a float type feature or is not implemented by the device.

5.1.17 – AT_GetFloatMax

```
int AT_GetFloatMax(AT_H Hndl, AT_WC* Feature, double * MaxValue)
```

Description

This function will return the maximum allowable value for the specified float type feature.

5.1.18 – AT_GetFloatMin

```
int AT_GetIntMin(AT_H Hndl, AT_WC* Feature, double * MinValue)
```

Description

This function will return the minimum allowable value for the specified float type feature.

5.1.19 – AT_SetBool

```
int AT_SetBool(AT_H Hndl, AT_WC* Feature, AT_BOOL Value)
```

Description

This function will set the value of the specified boolean feature. A value of AT_FALSE indicates false and a value of AT_TRUE indicates true. An error will be returned if the feature is read only, currently not writable, not a boolean feature or is not implemented by the device.

5.1.20 – AT_GetBool

```
int AT_GetBool(AT_H Hndl, AT_WC* Feature, AT_BOOL * Value)
```

Description

This function will return the current value of the specified boolean feature. If a value of AT_FALSE is returned then the feature is currently set to false. If a value of AT_TRUE is returned then the feature is currently set to true. An error will be returned if the feature is currently not readable, not a boolean feature or is not implemented by the device.

5.1.21 – AT_Command

```
int AT_Command(AT_H Hndl, AT_WC* Feature)
```

Description

This function will trigger the specified command feature to execute. An error will be returned if the feature is currently not writable, not a command feature or is not implemented by the device.

5.1.22 – AT_SetString

```
int AT_SetString(AT_H Hndl, AT_WC* Feature, AT_WC* Value)
```

Description

This function will set the value of the specified string feature. The string should be null terminated. An error will be returned if the feature is read only, currently not writable, not a string feature or is not implemented by the device.

5.1.23 – AT_GetString

```
int AT_GetString(AT_H Hndl, AT_WC* Feature, AT_WC* Value, int StringLength)
```

Description

This function will return the current value of the specified string feature. The length of the string in which you want the value returned must be provided in the fourth parameter and the string should include enough space for the null terminator. An error will be returned if the feature is currently not readable, not a string feature or is not implemented by the device.

5.1.24 – AT_GetStringMaxLength

```
int AT_GetStringMaxLength(AT_H Hndl, AT_WC* Feature, int* MaxStringLength)
```

Description

This function will return the maximum length of the specified string feature. This value can be used to determine what size of string to allocate when retrieving the value of the feature using the AT_GetString function.

5.1.25 – AT_SetEnumIndex

```
int AT_SetEnumIndex(AT_H Hndl, AT_WC* Feature, int Value)
```

Description

This function sets the currently selected index of the specified enumerated feature. The index is zero based and should be in the range 0 to Count-1, where Count has been retrieved using the AT_GetEnumCount function. An error will be returned if the feature is read only, currently not writable, the index is outside the allowed range, not an enumerated feature, or the feature is not implemented by the device. In some cases an index within the range may not be allowed if its availability depends on other features values, in this case an error will be returned if this index is applied.

5.1.26 – AT_SetEnumString

```
int AT_SetEnumString(AT_H Hndl, AT_WC* Feature, AT_WC* String)
```

Description

This function directly sets the current value of the specified enumerated feature. The String parameter must be one of the allowed values for the feature and must be currently available.

5.1.27 – AT_GetEnumIndex

```
int AT_GetEnumIndex(AT_H Hndl, AT_WC* Feature, int* Value)
```

Description

This function retrieves the currently selected index of the specified enumerated feature. The function will return an error if the feature is currently not readable or if the specified feature is either not an enumerated type feature or is not implemented by the device.

5.1.28 – AT_GetEnumCount

```
int AT_GetEnumCount(AT_H Hndl, AT_WC* Feature, int* Count)
```

Description

This function returns the number of indexes that the specified enumerated feature can be set to.

5.1.29 – AT_GetEnumStringByIndex

```
int AT_GetEnumStringByIndex(AT_H Hndl, AT_WC* Feature, int Index, AT_WC* String,
                           int StringLength)
```

Description

This function returns the text representation of the specified enumerated feature index. The index should be in the range 0... Count-1, where Count has been retrieved using the AT_GetEnumCount function. The length of the String parameter should be passed in to the fifth parameter.

5.1.30 – AT_IsEnumIndexAvailable

```
int AT_IsEnumIndexAvailable(AT_H Hndl, AT_WC* Feature, int Index,
                                  AT_BOOL* Available)
```

Description

This function indicates whether the specified enumerated feature index can currently be selected. The availability of enumerated options may depend on the value of other features.

5.1.31 – AT_IsEnumIndexImplemented

```
int AT_IsEnumIndexImplemented(AT_H Hndl, AT_WC* Feature, int Index,
                                    AT_BOOL* Implemented)
```

Description

This function indicates whether the device supports the specified enumerated feature index. For consistency across the device range, some enumerated features options may appear in the list even when they are not supported, this function will let you filter out these options.

5.1.32 – AT_QueueBuffer

```
int AT_QueueBuffer(AT_H Hndl, AT_U8* Ptr, int PtrSize)
```

Description

This function configures the area of memory into which frames to display on the device will be stored. The area of memory used to upload frames to Frame Memory should also be queued using this function. The PtrSize parameter should be equal to the size of a frame in number of bytes. This function has to be called before the frame exposure starts. Any buffer queued using this function should not be deallocated by the calling application until the program finishes its interaction with the device.

5.1.33 – AT_WaitBuffer

```
int AT_WaitBuffer(AT_H Hndl, AT_U8** Ptr, int* PtrSize, unsigned int Timeout)
```

AT_WaitBuffer is not used for Mosaic III device

5.1.34 – AT_Flush

```
int AT_Flush(AT_H Hndl)
```

AT_Flush is not used for Mosaic III device

Appendix

SDK3

Mosaic Feature Quick Reference

Feature	Type	Available Options
Abort	Command	Na
ClearFrameMemory	Command	Na
ClearSequenceMemory	Command	Na
ControllerFirmwareVersion	String	Na
DeviceCount	Integer	Na
DeviceType	String	Na
Expose	Command	Na
ExposureTime	Floating Point	Na
ImageSizeBytes	Integer	Na
InvertMode	Boolean	Na
IsExposing	Boolean	Na
FirmwareVersion	String	Na
FrameCount	Integer	Na
FrameCycleCount	Integer	Na
FrameIndex	Integer	Na
FrameIndexSource	Enumerated	Software, Hardware
FrameMemoryCount	Integer	Na
GrayScaleExposureTime	Floating Point	Na
GrayScaleSequenceEncoding	Enumerated	4BitDIP, 6BitDIP, 8BitDIP
OperationMode	Enumerated	Live, FrameMemory, FrameSequence, ContinuousFrameSequence
OverlapMode	Boolean	Na
PixelEncoding	Enumerated	Mono1, Mono1Packed
SensorHeight	Integer	Na
SensorWidth	Integer	Na
SerialNumber	String	Na
SequenceEventCount	Integer	Na
SequenceExposureTime	Floating Point	Na
SequenceLoopCount	Integer	Na
SequenceLoopLength	Integer	Na
SequenceGapTime	Floating Point	Na
SequenceStartIndex	Integer	Na
SoftwareVersion	String	Na
TriggerMode	Enumerated	InternalExpose, InternalSoftware, ExternalExpose, ExternalSequenceStart, ExternalBulb
UploadFrame	Command	Na
UploadGrayScaleImage	Command	Na
UploadSequenceEvent	Command	Na

ANDOR™ TECHNOLOGY

Function Quick Reference

```
int AT_InitialiseLibrary();
int AT_FinaliseLibrary();

int AT_Open(int DeviceIndex, AT_H* Handle);
int AT_Close(AT_H Hndl);

typedef int (*FeatureCallback)(AT_H Hndl, AT_WC* Feature, void* Context);
int AT_RegisterFeatureCallback(AT_H Hndl, AT_WC* Feature, FeatureCallback EvCallback, void* Context);
int AT_UnregisterFeatureCallback(AT_H Hndl, AT_WC* Feature, FeatureCallback EvCallback, void* Context);

int AT_IsImplemented(AT_H Hndl, AT_WC* Feature, AT_BOOL* Implemented);
int AT_IsReadOnly(AT_H Hndl, AT_WC* Feature, AT_BOOL* ReadOnly);
int AT_IsReadable(AT_H Hndl, AT_WC* Feature, AT_BOOL* Readable);
int AT_IsWritable(AT_H Hndl, AT_WC* Feature, AT_BOOL* Writable);

int AT_SetInt(AT_H Hndl, AT_WC* Feature, AT_64 Value);
int AT_GetInt(AT_H Hndl, AT_WC* Feature, AT_64 * Value);
int AT_GetIntMax(AT_H Hndl, AT_WC* Feature, AT_64 * MaxValue);
int AT_GetIntMin(AT_H Hndl, AT_WC* Feature, AT_64 * MinValue);

int AT_SetFloat(AT_H Hndl, AT_WC* Feature, double Value);
int AT_GetFloat(AT_H Hndl, AT_WC* Feature, double * Value);
int AT_GetFloatMax(AT_H Hndl, AT_WC* Feature, double * MaxValue);
int AT_GetFloatMin(AT_H Hndl, AT_WC* Feature, double * MinValue);

int AT_SetBool(AT_H Hndl, AT_WC* Feature, AT_BOOL Value);
int AT_GetBool(AT_H Hndl, AT_WC* Feature, AT_BOOL* Value);

int AT_SetEnumIndex(AT_H Hndl, AT_WC* Feature, int Value);
int AT_SetEnumString(AT_H Hndl, AT_WC* Feature, AT_WC* String);
int AT_GetEnumIndex(AT_H Hndl, AT_WC* Feature, int* Value);
int AT_GetEnumCount(AT_H Hndl, AT_WC* Feature, int* Count);
int AT_IsEnumIndexAvailable(AT_H Hndl, AT_WC* Feature, int Index, AT_BOOL* Available);
int AT_IsEnumIndexImplemented(AT_H Hndl, AT_WC* Feature, int Index, AT_BOOL* Implemented);
int AT_GetEnumStringByIndex(AT_H Hndl, AT_WC* Feature, int Index, AT_WC* String, int StringLength);

int AT_Command(AT_H Hndl, AT_WC* Feature);

int AT_SetString(AT_H Hndl, AT_WC* Feature, AT_WC* Value);
int AT_GetString(AT_H Hndl, AT_WC* Feature, AT_WC* Value, int StringLength);
int AT_GetStringMaxLength(AT_H Hndl, AT_WC* Feature, int* MaxStringLength);

int AT_QueueBuffer(AT_H Hndl, AT_U8 * Ptr, int PtrSize);
int AT_WaitBuffer(AT_H Hndl, AT_U8 ** Ptr, int* PtrSize, unsigned int Timeout);
int AT_Flush(AT_H Hndl);
```

SDK3

Code Listing for Tutorial

```
int main(int argc, char* argv[])
{
  int i_returnCode;
  AT_H Hndl;
  int i_cameraIndex = 0;
  i_returnCode = AT_InitialiseLibrary( );

  if (i_returnCode == AT_SUCCESS) {
    i_returnCode = AT_Open ( i_cameraIndex, &Hndl );
    AT_WC ExpFeatureName[] = L"ExposureTime";
    double d_newExposure = 0.02;
    i_returnCode = AT_SetFloat ( Hndl,  ExpFeatureName, d_newExposure);
    if (i_returnCode == AT_SUCCESS) {
      //it has been set
      double d_actualExposure;
      i_returnCode = AT_GetFloat ( Hndl,  ExpFeatureName, &d_actualExposure);
      if (i_returnCode == AT_SUCCESS) {
        //the actual exposure being used is d_actualExposure

        AT_64 ImageSizeBytes;
        AT_GetInt( Hndl, L"ImageSizeBytes", &ImageSizeBytes);
        //cast to prevent warnings
        int i_imageSize = static_cast<int>(ImageSizeBytes);

        unsigned char* gblp_Buffer = new unsigned char[i_imageSize+8];

        i_returnCode = AT_QueueBuffer(Hndl,  gblp_Buffer, i_imageSize);
        if (i_returnCode == AT_SUCCESS) {

          //Get the sensor width
          AT_64 i64_sensorWidth;
          i_returnCode = AT_GetInt(Hndl, L"SensorWidth", &i64_sensorWidth);
          if (i_returnCode == AT_SUCCESS) {

            //Get the sensor height
            AT_64 i64_sensorHeight;
            i_returnCode = AT_GetInt(Hndl, L"SensorHeight", &i64_sensorHeight);
            if (i_returnCode == AT_SUCCESS) {
              int i_index;
              //Populate UserBuffer with the rectangle shape
              for (int y = 100; y < i64_sensorHeight - 100; y++) {
                for (int x = 100; x < i64_sensorWidth - 100; x++) {
                  i_index = y*i64_sensorWidth + x;
                  gblp_Buffer [i_index] = 1;
                }
              }
            }
          }
          i_returnCode = AT_Command(Hndl, L"Expose");
          if (i_returnCode == AT_SUCCESS) {
            // Application specific data processing goes here...
          }
          AT_Command(Hndl, L"Abort");
        }
        delete [] gblp_Buffer;
      }

      i_returnCode = AT_Close ( Hndl );
      if (i_returnCode != AT_SUCCESS) {
        // error closing handle
      }
    }
  }

  i_returnCode = AT_FinaliseLibrary( );
  if (i_returnCode != AT_SUCCESS) {
    //Error FinaliseLibrary
  }
  return 0;
}
```

Conversion between char* and AT_WC*

The following code shows an example of how to convert a char* null terminated string to the equivalent wide character string.

```
#include “stdlib.h”
char szStr[512];
AT_WC wcszStr[512];
mbstowcs(wcszStr, szStr, 512);
```

and from wide character string to char*

```
#include “stdlib.h”
char szStr[512];
AT_WC wcszStr[512];
wcstombs(szStr, wcszStr, 512);
```