Multi-core Video Analytics Engine (MVE™) V2

API User's Guide

V0.6

Sept 28, 2015

eUTecUs

www.eutecus.com
info@eutecus.com
Eutecus, Inc. , 1936 University Ave., Suite 360 , Berkeley, CA 94704
Phone: (510) 540-9603
Fax: (510) 649-7808

Table of Contents

List of Examples

1. API Overview

1.1. Introduction

The MVE™ Application Programing Interface is a cross platform API that can control and configure an MVE™ device. From the MVE™ API's point of view, the MVE™ device (hereinafter the device) functions as a server, so it listens on different TCP ports waiting for connections from a remote client (hereinafter the host) and provides its services through these ports.

1.2. Supported Platforms

The MVE™ API has been implemented in C++ and has been compiled and tested on Windows and Linux platforms. More generally, it is a cross platform API that can be used on any platform where a suitable C++ compiler, thread support libraries and BSD socket networking are available. The API package includes binary and header files. The header files are the same for all platforms, and these combined is the unified interface of the MVE™ API. The binary file includes the precompiled C++ codes for the given platform, and it has to be linked to or dynamically loaded by the application.

1.2.1. Windows

The MVE™ API is compiled with MS Visual Studio 2013 for Microsoft Windows 7. The `CrossAPI_msvc12_x86.lib` must be linked to the application. Make sure that the include and library paths are set properly to point to the include and lib folders, respectively, of the MVE™ API package.

1.2.2. Linux

The MVE™ API is compiled with GNU GCC's g++ (version 4.7-4.9, depending on the project) for Linux. The `libCrossAPI.a` has to be linked and the pthread library has to be added to the project for a successful compilation. Make sure that the include and library paths are set properly to point to the include and lib folders of the MVE™ API package.

2. Low-Level MVE™ API

The low-level MVE™ API provides the basic services for an MVE™ device. The interface is declared in the Bii_*.h header files, named after the service name. The implemented services are: the Firmware Update Service and the Status Service.

2.1. Firmware Update Service

2.1.1. Firmware Update

The facilities in Bii_FirmwareUpdate.h allows updating the device firmware by calling the `Bii_FirmwareUpdate` function, which requires

1. the device IP or domain name address,
2. the path to the firmware directory,
3. a callback function which gets called regularly with the current status of the firmware update,
4. a firmware update target (optional, defaults to Application firmware update)
5. send/receive timeout (optional, defaults to 2000 ms)

Both the callback function prototype (`Bii_FirmwareUpdateStatus_Callback`) and a helper function to convert the update stage enum to a textual representation (`Bii_UpdateStage2String`) is available in the Bii_FirmwareUpdate.h header file.

2.1.2. Compute Firmware Hash

A device firmware consists of several parts. The hashes of these parts of a firmware stored in the filesystem can be computed by the `Bii_FirmwareHash` function, which returns with an `std::map` containing the firmware part name and its hash. This can be used, for example, to compare the firmware on the device and on storage. To compute the hashes of the current firmware on a device, please refer to the `Bii_StatusServiceGetSystemHash` function of the Status Service.

2.2. Status Service

System status retrieval, reboot, timestamping, version and device ID fetch and firmware hash functionality is provided via the status service API defined in the Bii_StatusService.h header.

Upon successfully connecting to the status service via the `Bii_StatusServiceConnect` function (passing the device address, communication port and an optional timeout value), an opaque handle object is returned which shall be used for subsequent service requests. After the last request, the connection shall be closed by calling `Bii_StatusServiceDisconnect`.

Timestamping, reboot, status-, version- and hardware ID retrieval functions are available via the `Bii_StatusServiceTimeStamp` , `Bii_StatusServiceReboot`, `Bii_StatusServiceGetStatus`, `Bii_StatusServiceGetVersion` and `Bii_StatusServiceGetHardwareID` functions, respectively.

The hashes of the current device firmware parts can be fetched by the `Bii_StatusServiceGetSystemHash` function, which returns with an `std::map` containing the firmware part name and its

hash. This can be used to compare the firmware on the device and on storage. To compute the hashes of a firmware stored in the filesystem, please refer to the `Bii_FirmwareHash` function of the Firmware Update Service.

2.3. Sample Firmware Update Application With Version Check

Utilizing the status service's facilities, it is possible to write an automatic firmware update application which only updates the firmware if it does not match with the firmware in the MVE™ device:

```
#include <iostream>

#include "Bii_FirmwareUpdate.h"
#include "Bii_StatusService.h"

void BIIAPI UpdateStatus(Bii_UpdateStage stage, unsigned int percentage)
{
  std::cout << "Updating firmware: " << Bii_UpdateStage2String(stage) << " - "
            << percentage << '%' << std::endl;
}

int main(int argc, char * argv[])
{
  if (argc != 3) {
    std::cerr << "Usage: " << argv[0] << " IP_ADDRESS firmware_folder" << std::endl;
    return -1;
  }

  std::cout << "Validating firmware version numbers..." << std::endl;
  bool firmware_update_required = true;

  std::cout << "Connecting to secondary device status interface: "
            << argv[1] << ':' << 12011 << std::endl;
  void * StatusServiceHandle = Bii_StatusServiceConnect(argv[1], 12011, 2);
  if (nullptr == StatusServiceHandle) {
    std::cerr << "Error connecting secondary device. Status port is unavailable." << std::endl;
  }
  else {
    std::cout << "Get FPGA firmware hash..." << std::endl;
    auto SystemHashMap = Bii_StatusServiceGetSystemHash(StatusServiceHandle);
    Bii_StatusServiceDisconnect(StatusServiceHandle);
    if (2 > SystemHashMap.size()) {
      std::cout << "Get FPGA firmware hash...FAILED (" << SystemHashMap.size() << ')' << std::endl;
    } else {
      std::cout << "Get FPGA firmware hash...DONE:" << std::hex << std::endl;
      for (auto const & hash : SystemHashMap) {
        std::cout << hash.first << ' ' << hash.second << std::endl;
      }
      std::cout << std::dec;
    }

    std::cout << "Get firmware file hash from '" << argv[2] << "'..." << std::endl;
    auto FirmwareFileHash = Bii_FirmwareHash(argv[2]);
    if (FirmwareFileHash.empty()) {
      std::cerr << "Get firmware file hash from '" << argv[2]
                << "'...FAILED: No firmware found" << std::endl;
    }
    else {
      std::cout << "Get firmware file hash from '" << argv[2] << "'...DONE:" << std::endl;
      std::cout << "Firmware Hash : " << std::hex << std::endl;
      for (auto const & hash : FirmwareFileHash) {
        std::cout << hash.first << ' ' << hash.second << std::endl;
      }
      std::cout << std::dec;
```

```
    }

    if ((2 <= SystemHashMap.size()) && (SystemHashMap == FirmwareFileHash)) {
      std::cout << "FPGA firmware is matching with firmware in '" << argv[2]
                << "', no need to update." << std::endl;
      firmware_update_required = false;
    }
  }

  if (firmware_update_required) {
    std::cout << "FPGA firmware update is needed." << std::endl;
    bool retval = Bii_FirmwareUpdate(argv[1], argv[2], UpdateStatus);
    std::cout << "Firmware update was " << (retval ? "SUCCESSFUL" : "UNSUCCESSFUL") << std::endl;
  }

  return 0;
}
```

3. High-Level MVE™ API

The high-level MVE™ API provides complex control and configuration options for the MVE™ device. This API makes it easy to set the algorithmic parameters for the MVE™ or receive the metadata stream from the device. The high-level MVE™ API interface is declared in the MVE_*.h header files.

3.1. Control Service

The Control Service is declared in the MVE_Control.h header file. The functionality is provided via an opaque handler object.

The `MVE_Init` function initializes (and returns) the high-level MVE™ API handler object without starting analytics on the device thus allowing off-line parameter tuning and/or parameter set-up.

The `MVE_CreateParameter` function creates a new parameter and can only be used in off-line mode.

The `MVE_GetParamValue` and `MVE_SetParamValue` functions get and set a parameter value, respectively. The `MVE_GetParameter` function returns a description of a parameter. These functions can be used in both on-line and off-line parameter tuning.

The `MVE_Start` function starts the video analytics on the device with the parameters stored in the handler object (and optionally pre-tuned off-line).

The `MVE_Close` function shall be used to close an established connection to an MVE™ device.

The connection state can be checked by the `MVE_IsConnectionOK` function, which checks both the host-to-device and the device-to-host connection state. Note that this process is a passive check (i.e. not an echo-reply check), so it only knows about already encountered errors.

3.1.1. Single Command Configuration

If off-line parameter tuning is not needed, the high-level MVE™ API provides a very simple interface to configure an application on the MVE™ device with only one function call. Only the `MVE_Run` function has to be called, and it loads the parameters from storage, establishes the connection with the device, uploads all the parameters and starts video analytics.

This function requires the following information to establish a connection to an MVE™ device:

- IP address (or domain name) of the device which contains MVE™,
- TCP port numbers where the MVE™ device is listening,
- user name and password for the authentication process,
- pointer to a callback function that handles the incoming messages.

The IP address can be found in the user's manual of the given device, or the network administrator can provide this information, if the IP assignments are managed by a DHCP server. The TCP port numbers are 12000 and 12001. There is one TCP port for the device-to-host and one for the host-to-device communication. The user name and the password are specified in the user's manual of the device.

The API sends all the messages that come from the MVE™ device to the application via a callback function. This callback function needs to be implemented at the application level, and the prototype of this function can be found in Bii_HostData.h as `Bii_Callback`.

The last two parameters specify the ini files that contain the application parameters for all channels.

The usage of the `MVE_Run` function is shown in the following example:

```
void BIIAPI ProcessInformation(unsigned int message, void* wParam, void* lParam) {
            // Message handling needs to be implemented here
}

//...

char *IniFiles[] = {"ParametersCH1.ini",
                    "ParametersCH2.ini",
                    "ParametersCH3.ini",
                    "ParametersCH4.ini"};

void * RunHandle = MVE_Run("192.168.0.90", 12000, 12001,
                           "User", "Password", ProcessInformation, 4, IniFiles);
```

Example 3.1. Single Command Configuration

3.1.2. Control Messages

The device sends control messages to inform the host about an event. Typically, the first event that will trigger the `ProcessInformation` callback function is the `MSG_CONNECT`. This message informs the host that the connection with the MVE™ device has been established.

The following control messages are supported by the high-level MVE™ API:

- `BII_MSG_CONNECT`: Connection established message from the device
- `BII_MSG_DISCONNECT`: Disconnect request by the host and disconnect notification by the device

3.2. Parameter Handling

Parameter handling is the most important part of the high-level MVE™ API. This provides an interface to set the MVE™ algorithmic parameters and the device configuration parameters.

3.2.1. Synchronizing the Parameter Structure with the MVE™ device

Synchronizing the parameter structure is an essential part of the parameter handling. Without this synchronization the parameters cannot be set to the MVE™ device. The synchronization process is fully automatic and handled by the MVE™ API.

The MVE™ API will fill out the parameter description structure during the configuration process. The MVE™ device provides all the information about the parameters like type, address and range, and the MVE™ API responds with the parameter values based on the ini file that was set with the `MVE_Run` or `MVE_Init` functions.

The parameter description structure contains the path of the parameter, the type, the address in the device's memory for the parameter update, and the parameter value. This structure has to be initialized in or-

der to modify a parameter value. The structure can be initialized based on the XML description, which will be available in future releases.

3.2.2. Set/Get Parameter Values

The parameter values can be set, changed, or modified, after the parameter description structure was initialized. There are two ways to initialize this structure: by synchronizing it to the MVE™ device, or using the XML based parameter description (available in future releases).

The following functions can be used to access the parameters:

- The `MVE_SetParamValue` function, which can set the new parameter value, requires the path of the parameter and the new parameter value with the same type as the original parameter. The different types of parameters are supported by functions with different argument types. The enumeration type parameters are used as integer parameters, and the string based enumeration handling will be available in future releases.
- The `MVE_GetParamValue` function requires the path of the parameter and returns with the value of the parameter. The different parameter types are supported by functions with different return types. The enumeration type support is the same as for the set function.

3.3. Metadata Handling

Metadata handling is a callback oriented one-directional interface originating in the MVE™ V2 metadata capable device.

Based on the analyzed video frame, metadata is grouped into packages. Each metadata package has a header describing the frame (timestamp, etc.) and consists of metadata containers about that single frame only. All metadata about a certain frame is sent in a single metadata package.

The metadata handling interface is declared in the MVE_MetadataV2*.h header files.

3.3.1. Opening a Metadata Stream

The top-level metadata streaming class is the `TMetadataV2Capturer` class (declared in MVE_MetadataV2Capture.h). An object of this class is created by the `MVE_MetadataV2_Capturer_Create` function.

Metadata handling shall be implemented in a callback function. It is recommended to register the callback function with `TMetadataV2Capturer::RegisterPackageHandler` before connecting to the MVE™ device and starting the metadata streaming, not to lose any metadata package. The callback will be called with the metadata package header object, `TMetadataV2Header`, and an `std::vector` consisting of the metadata containers in the package. The signature of the callback function is the following:

```
void (TMetadataV2Header const & package_header,
      std::vector<std::shared_ptr<TMetadataV2Container>> && containers)
```

To start the metadata streaming, pass the IP address (or domain name) of the MVE™ device and optionally the metadata port to the `TMetadataV2Capturer::StartCapture` function. The registered metadata package callback function will be called each time a metadata package is received.

The IP address of the device can be found in the user's manual of the device, or the network administrator can provide this information, if the IP address range is managed by a DHCP server.

Refer to the basic metadata capture setup in the sample metadata capture example (Section 3.3.5, Sample Metadata Capture Application)

3.3.2. Closing a Metadata Stream

To close a metadata connection simply call the `TMetadataV2Capturer::StopCapture` function.

Note

It is important that the application has to handle the meta data packages until the metadata connection towards the device is fully closed, otherwise the data packages will be lost. (The MVE™ device will drop metadata packages if it detects that the communication channel is stalled.)

3.3.3. Metadata Connection Check

The metadata connection can be checked by the `IsConnected` and the `IsHandlerRunning` functions. If the handler is running, its termination can be waited for by the `WaitForHandlerFinish` function.

3.3.4. Metadata Package Handling

The metadata packages are provided to the application through the callback function that were registered with the metadata streamer object.

The callback function can access the data structures that contain all the elements of the metadata containers. The content of the metadata containers are described in the Metadata Streaming Specification document.

3.3.5. Sample Metadata Capture Application

```
void SignalHandler(int)
{
  printf("SIGINT signal has been received.\n");
  if(MetaDataHandle){
    printf("Sending metadata capturing stop message...\n");
    MetaDataHandle->StopCapture();
  }
}

void PackageHandler(TMetadataV2Header const & h,
                    std::vector<std::shared_ptr<TMetadataV2Container>> && containers)
{
  std::cout << h << '\n';
  for (auto const & c : containers) {
    std::cout << '\t' << *c << '\n';
  }
}

std::unique_ptr<TMetadataV2Capturer> MetaDataHandle = MVE_MetadataV2_Capturer_Create();

int main(int argc, char* argv[])
{
  signal(SIGINT,SignalHandler);

  if (argc != 3) {
    printf("Usage: %s ip_address port\n", argv[0]);
    return 1;
  }

  MetaDataHandle->RegisterPackageHandler(PackageHandler);

  std::cout << "MVE Metadata capture start" << std::endl;
  MetaDataHandle->StartCapture(argv[1], atoi(argv[2]));

  while (MetaDataHandle->IsConnected()) {
    #ifdef WIN32
      Sleep(10);
    #else //LINUX
      usleep(10000);
    #endif
  }

  std::cout << "MVE Metadata capture stop" << std::endl;

  return 0;
}
```

Example 3.2. Sample Metadata Capture Application