Horizon 2020 PROGRAMME *ICT-01-2014: Smart Cyber-Physical Systems*

This project has received funding from the European Union's Horizon 2020 research and innovation programme under Grant Agreement No 643924

D4.1
Demonstrator 1, EoT Application

1. DOCUMENT INFORMATION

Authors	C. Fedorczak (THALES) T. Larmoire (THALES) Stephan Krauß (DFKI) Alain Pagani (DFKI)
Responsible Author	Alain Pagani (DFKI) e-mail: alain.pagani@dfki.de

Keywords	Demonstrator 1 – Peephole Surveillance
WP/Task	WP4
Nature	Other
Dissemination Level	PU

2. DOCUMENT HISTORY

Person	Date	Comment	Version
Alain Pagani	06.06.2018	Delivered version	1.0

3. ABSTRACT

This document is a software description document that accompanies the deliverable D4.1 “Peephole Demonstrator EoT Application”. This deliverable is the software developed on the EoT Board in order to implement the Peephole surveillance demonstrator. The software is made available to the reviewers on a GitLab server. This document first provides a short description of the demonstrator and its features. It then describes the developed software that runs on the EoT board. The interface of this software is finally documented, with a list of all structures and functions available.

4. TABLE OF CONTENTS

5. INTRODUCTION

This document describes the software for the Peephole Surveillance demonstrator running on the EoT device. The Peephole Surveillance demonstrator is one of the possible uses of the EoT device, where the device is placed on the exterior part of the main door of an apartment or house. The software loaded in the EoT device performs an image analysis task and is able to detect if someone appears in front of the door. When such an event happens, a notification is sent to a companion device (smartphone or tablet).

The code is available in GitLab at the following address:

https://gitlab.com/espiaran/EoT/tree/DFKI/WorkPackage_4/Peephole

The reviewers will be able to access the private parts of the code on request.

6. SHORT DESCRIPTION OF THE DEMONSTRATOR

The demonstrator described in this document is a peephole surveillance system. Before leaving home, the user attaches the EoT-based device to the peephole and configures it through a smartphone app (see Figure 1). The device will continuously monitor for motion or faces in front of the door, sending alarms and pictures to the user's smartphone. The device does not need cables, since it functions with its own rechargeable battery. Furthermore, the device will also detect tampering events (e.g. attempts to cover the peephole) and generate alarms. Besides these instant alarms/notifications the devices also records video clips of the detected events and send them to the user on demand.

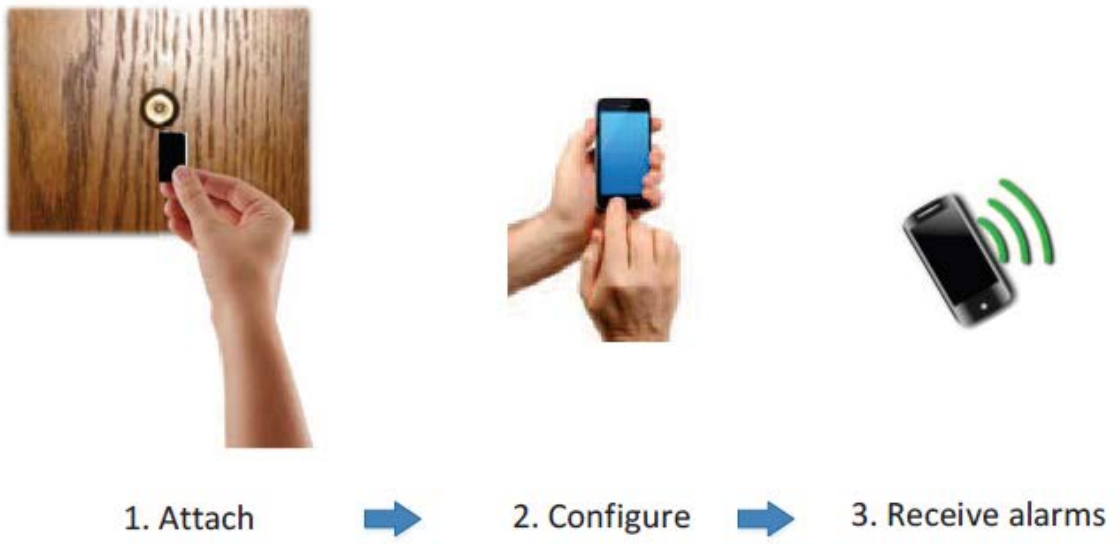

Figure 1: Setup of the peephole surveillance demonstrator.

Scenario 1: Surveillance
Once activated, the system monitors for activity (motion) at the door. If it detects some, the system starts recording at 30 frames per second and sends a motion alert to the user. The recording is sent to the user upon request. After some time without activity, the system stops recording.

Scenario 2: Tampering detection
This scenario is an extension of scenario 1, where a presence has already been detected and the user informed through a first alarm on the smartphone. Now the person in front of the door covers the peephole either blocking light completely or blurring the view significantly. The system detects this, raises a tampering alert and sends it to the user's smartphone. The user can again request the video footage of the event.

Scenario 3: Face detection
In this scenario, the user has configured the device to perform face detection via the mobile configuration app. If the system detects a face, the user is alerted and a picture of the face is sent to the user's smartphone.

The original requirement analysis document mentioned in addition two scenarios: Scenario 4: Face recognition, and Scenario 5: Instant two-way voice communication. These two scenarios were optional and were to be implemented only after the successful implementation of the first three scenarios. After the development of the three first scenarios, the allocated budget was already consumed, and the optional scenarios have not been implemented.

7. EOT APPLICATION SOFTWARE DESCRIPTION

In the following, we start by stating the demonstrator requirements. We then describe the software that runs on the EoT device. It consists of four modules: (1) camera and ISP, (2) event detection, (3) event recording, (4) network communication and the application logic.

7.1. Demonstrator requirements

A requirements analysis has been conducted for all demonstrators in the first phase of the project. The result of the analysis has been documented in a report that has been submitted as an annex for the mid-term review ("Annex 2").

For the Peephole Surveillance demonstrator, the following requirements have been identified. In the following table, we provide the name of each requirement, and indicate if this requirement has been covered by the implementation or not. In case the requirement was not implemented, we justify the modification of the development plan in the last column.

ID	Name	Description	Achieved
REQ001	Connect to WiFi hot spot	Connection to the local network/Internet	Yes, Broker on the EoT device
REQ002	Connect to the PC or Smatphone app	Connection to the app	Yes
REQ003	Module set-up	Set cloud address, login, password, application	Yes
REQ004	Start operation	Start the app on the module	Yes
REQ005	Stop operation	Stop the app on the module	Yes
REQ006	Status	Send the module status: • Active /Standby • Event detected or not • Last events • Memory used/remaining • Battery level • I/O status • Camera status (average light level)	Partially (Not implemented: Memory used, battery level, camera status – justification: not needed)
REQ007	Take photographs	Take a picture with the camera and send it over wifi	Yes

REQ008	Loop recording at 1 fps	Record 60 frames at 1 fps in a circular buffer	Yes
REQ009	Circular buffer freezing	Freeze the circular buffer	Yes
REQ010	Recording @ 25 fps or 12 fps	Record during the alarm duration	Yes
REQ011	Send an alarm to the cloud app	Send the alarm flag and alarm type to the cloud server	Partially – no cloud server used, but direct connection with the companion device
REQ012	Send the picture of the alarm	Send the picture corresponding to the triggering of the alarm to the cloud server	Partially – no cloud server used, but direct connection with the companion device
REQ013	Send the pre-alarm buffer	Send the circular buffer to the cloud server	Partially – no cloud server used, but direct connection with the companion device
REQ014	Send the post alarm buffer	Send the post alarm buffer to the cloud server	Partially – no cloud server used, but direct connection with the companion device
REQ015	Send the live video	Send the live video to the cloud app	Partially – no cloud server used, but direct connection with the companion device
REQ016	Return to standby mode	• Transfer all buffers to the cloud server • Clear buffers • Resume circular buffer recording	Yes

REQ017	Presence detection	Detect one or several objects moving in front of the camera. Size of the objects to be determined (1 person at 5 meters)	Yes, face detection when a person is in front of the camera
REQ018	Tampering detection	• Detection of a dark picture for more than 5 sec • Detection of a partially occulted picture for more than 5 sec • Detection of a highly blurred picture for more than 5 sec	Yes
REQ019	Face detection	• Detect the presence of a face with a width ranging between x and y pixels (tbd) • Detect a face oriented between + and – X degrees horizontally (tbd) • Detect a face oriented between + and – X degrees vertically (tbd)	Yes
REQ020	Face contrast	Detect a face with a minimum contrast of (10%) of the full scale level (tbd)	Yes
REQ021	Scene Illumination	Operation with a minimum illumination of 1 lux on the scene	Not tested
REQ022	Face thumbnail extract and send	Extract a thumbnail image of the face detected in the picture to the cloud server	Yes (direct connection)
REQ023	Upload face patterns to the module	Upload a list of known face patterns	No (optional scenario, not implemented)

REQ024	Send positive face recognition event	In case a facial match has been detected, send the event, face thumbnail and ID to the cloud server	No (optional scenario, not implemented)
REQ025	Start bi-directional audio	Start audio communication with the module	No (optional scenario, not implemented)
REQ026	Stop bi-directional audio	Stop audio communication with the module	No (optional scenario, not implemented)

7.2. Software architecture

The architecture is mainly composed of an application software and a communication module in the form of an MQTT broker. This broker is using the PULGA library derived from the MOSQUITTO implementation.

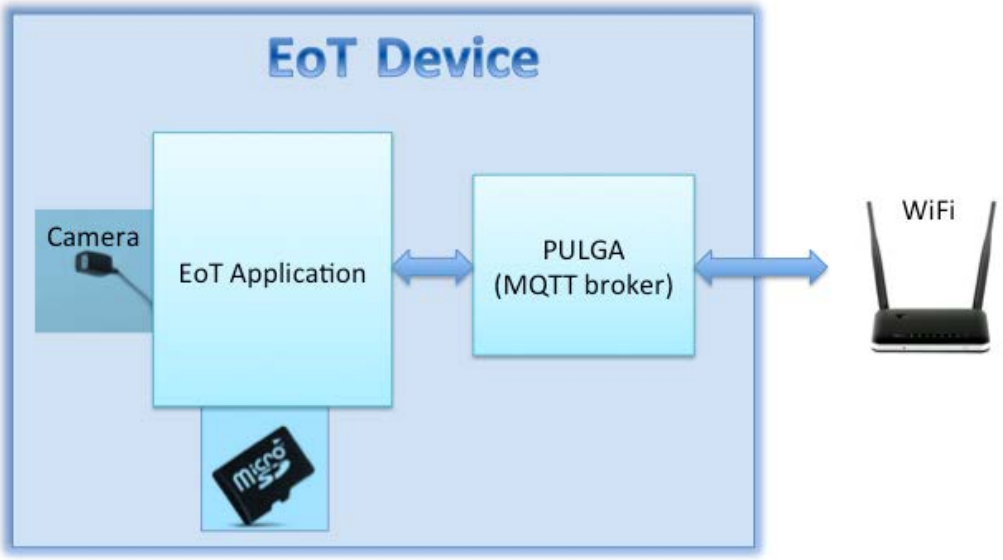

Figure 1: Architecture of the EoT Device Software

7.3. Camera and ISP

This module provides control functionality for the camera. It defines an abstract interface for camera control and provides a specific implementation for the Sony IMX 208 camera. The module can be extended to work with other cameras by providing an implementation of the camera interface for a new camera model. The implementation for the Sony camera provides frame rate control, auto-exposure (AE) and automatic gain control (AGC). The raw frames from the camera are pre-processed by an ISP, which takes care of receiving frames through the MIPI interface, de-noising them, eliminating dead pixels, applying a

lens shading correction, gathering statistics for AE and AGC as well as applying a gamma correction.

7.4. Event detection

This module provides the event detection functionality. Specifically, it can detect motion and tampering events as well as faces. Which types of events shall be detected can be configured through a mobile app, which is described in detail in the configuration software description document. A unique ID is assigned to each detected event.

7.5. Event Recording

This module provides video recording functionality for detected events. If the event detection module detects a new event, recording is started or continued (if recording was started before already). If no events are detected for a predefined duration, the recording is stopped. For each video frame a timestamp is stored as well.

7.6. Network communication

This module provides network communication functionality to the app. Specifically, it communicates with a mobile app through MQTT. This way the mobile app can update the internal clock of the EoT device and inform the EoT app about configuration changes. Furthermore, the EoT app can send alerts about events and – upon request from the mobile app – recorded events and video clips or a live video stream.

7.7. Application logic

This is the implementation of the logic of the application and integrates the other modules in the app. It determines how to react to messages it receives from the mobile app as well as how to react to the detection of a specific event. In particular, it ensures that the event is recorded along with a video clip and that an alert is sent to the mobile app.

8. EOT APPLICATION SOFTWARE DOCUMENTATION

In the following, we describe the most important interfaces of the modules.

8.1. EoT libraries used

This application uses the following libraries created in work package 3:

- MQTT broker
- WiFi functions
- JSON parser
- SDCardIO
- HistrogramMatching
- LEDs
- libccv

8.2. Camera and ISP

This module consists of two parts: the camera and the ISP management functionalities. They are described in the following sections.

8.2.1. Camera interface

This component provides a general interface for video cameras as well as an implementation of that interface for the Sony IMX 208 camera. The camera interface for the IMX 208 is defined as a singleton C++ class as follows:

```
class IMX208 : public ICamera
{
public:
  static IMX208* getInstance(bool colorCamera = false);

  IMX208(const IMX208&) = delete;
  IMX208(IMX208&&) = delete;
  IMX208& operator=(const IMX208&) = delete;
  IMX208& operator=(IMX208&&) = delete;
  ~IMX208() override;

  uint32_t getWidth() const override;
  uint32_t getHeight() const override;
  uint32_t getBytesPerPixel() const override;

  bool start() override;
  bool stop() override;

  uint8_t* getNextFrameBuffer() const override;
  uint32_t getFrameSequenceNumber() const override;

private:
  IMX208(bool colorCamera);
};
```

It extends the generic camera interface ICamera. The individual methods are described in the following:

The getWidth, getHeight and getBytesPerPixel methods return the width, height and depth (in bytes) of the video frames respectively. To start or stop capturing

video frames the respective start and stop methods need to be called. The getNextFrameBuffer method will block until a new frame becomes available and then return a pointer to the frame data. Internally, it performs a triple buffering to ensure smooth operation. If frames are polled at a rate lower than the camera's frame rate, it will automatically drop frames. Automatic gain and exposure control are also handled internally without any need for further configuration. Finally, the getFrameSequenceNumber method will return the sequence number of the current frame. This can be used, for example, to detect skipped frames.

8.2.2. ISP interface

The frame pre-processing uses dedicated hardware blocks of the Myriad processor to ensure optimal efficiency and speed. In Myriad terminology the interface to this hardware implementation of the image processing pipeline is called OPipe. Based on this OPipe interface, we have created a specific implementation for the Sony IMX 208 camera called IMX208Opipe. Its public interface is defined as follows:

```
struct OpipePipeline
{
  Opipe   p;
  DBuffer* pOut;  // to direct DDR output
};

class IMX208Opipe
{
public:
  static void createOpipe(OpipePipeline* pipeline,
                          uint32_t width,
                          uint32_t height);

  static void configureOpipe(Opipe* opipe,
                             AeAwbPatchStats* aeStats,
                             const uint16_t* lutTable,
                             bool colorCamera);
};
```

The first method createOpipe is used to initialize a new ISP configuration. It needs the width and height of the video frames and a structure containing all other necessary objects. In particular that includes the OPipe itself and a pointer to a location in memory to which the processed frame is written to. The OPipe is initialized by this method and further configured by the configureOpipe method. It requires the initialized OPipe as well as a location in memory where the auto exposure statistics shall be written to and a memory location containing the look-up table used for gamma correction. The last parameter determines whether the ISP is configured to work with a colour or grayscale version of the camera.

8.3. Event detection

The event detection module consists of several interfaces that address different event types. They are described in the following sections.

8.3.1. MotionDetection.hpp

This interface provides only one function: bool detectMotion(const frameBuffer* grayImage). It requires a pointer to a frameBuffer object containing a grayscale image as input. It returns true if motion was detected in this frame and false otherwise. The motion detection code assumes that it is called every frame. This way it can internally build the statistics necessary to decide whether motion has been observed or not (e.g. only noise). If this assumption is violated, false positives may occur.

8.3.2. TamperingDetection.hpp

This interface also contains only one function: bool detectTampering(const frameBuffer* rgbImage). It takes a frameBuffer object containing a RGB colour image and returns whether tampering was detected (true) or not (false). The tampering detection code also assumes that it is called every frame. This way it can internally build the statistics necessary to decide whether tampering has taken place or not. If this assumption is violated, false positives may occur.

8.3.3. Detection.hpp

This interface contains only the following function: void detection(const frameBuffer* rgbImage). It is a convenience function that takes care of everything necessary to execute motion and tampering detection. For example, it performs the grayscale conversion for the motion detection as well as a downscaling of the image in order to increase efficiency and execution speed. Furthermore, it checks whether motion and tampering detection have been enabled in the configuration and only executes them if that is the case. Finally, it also sets internal event status flags. They are, for example, used by the networking module to alert the user about detected events.

8.3.4. FaceDetection.hpp

This interface provides access to a C++ class for face detection. Besides the constructor and destructor, it only has one public method. The constructor loads the trained face detection model from the SD card into main memory and initializes the FaceDetector class. The memory is released by the destructor when the face detector object is destroyed. The remaining method takes care of detecting faces in a video frame and is defined as follows:

```
uint32_t FaceDetection::detect(const frameBuffer& frame,
                               frameBuffer& yCbCrFrame,
                               uint8_t* jpgBuffer,
                               unsigned int jpgBufferSize);
```

It requires a framebuffer object containing a video frame as input. Furthermore, it receives a version of the frame in YCbCr colour space. If a face is detected, the respective area in the YCbCr image is cut out and encoded as a JPEG. The JPEG is written to the jpgBuffer pointer provided in the function's interface. The last parameter jpgBufferSize informs the function about the size of the JPEG buffer. This way it can ensure that the JPEG encoded image is not written beyond the bounds of the buffer.

8.4. Event recording: Recording.hpp

Event recording module is used to store frames (data) and information about the events they belong to (meta-data). It provides functions for frame rate control (RecordSetFPS), frame recording (RecordFrame), meta-data look-ups

(RecordFindFirstMetaData, RecordCountMetaData, RecordFindMetaData, RecordGetMetaData) and frame look-up (RecordGetItem). They are described in the following sections.

8.4.1. Frame rate setting

The function void RecordSetFPS(uint8_t fps) is used to inform the recording process of the frame rate (the fps parameter) at which frames are supplied. This information is stored with each recording in order to be able to handle cases where the frame rate may change from recording to recording.

8.4.2. Frame recording

The function void RecordFrame(const uint8_t* jpeg, uint32_t jpegSize) records a single JPEG encoded video frame (parameter jpeg) of a specific size (parameter jpegSize) by adding it to the end of the list of recorded frames for the currently active event.

8.4.3. Finding meta-data for recorded events

Meta-data look-ups can be done either by event (ID and type) or by time. The former can be done with the following function:

int32_t RecordFindMetaData(uint32_t eventId, EventType eventType)

It takes an event ID and type and returns an index that can be used to access the meta-data store. If now matching entry could be found, it returns -1.

Look-ups using a time span are done with the following two functions:

uint32_t RecordFindFirstMetaData(uint64_t from, uint64_t to);

uint32_t RecordCountMetaData(uint64_t from, uint64_t to);

Both require the beginning and end of the time span to search for. RecordFindFirstMetaData returns the index of the first event in that time frame or 0 if none were found. The second function RecordCountMetaData returns the number of entries in that period of time.

8.4.4. Retrieving meta-data and frames

Once an index or a range of indices has been acquired through a meta-data lookup, it can be used to access the meta-data with the help of the following function: RecordingMetaData RecordGetMetaData(uint32_t index).

It takes an index and returns the respective meta-data element. It has the following structure:

```
struct RecordingMetaData {
  EventType eventType;     // The type of the event, e.g. motion.
  uint8_t fps;             // The framerate of the recording.
  uint32_t eventId;        // The ID of the event.
  uint32_t startIndexFrame; // Index of the first recorded frame.
  uint32_t endIndexFrame;   // Index of the last recorded frame.
  uint64_t startTimestamp;  // Timestamp of the first rec. frame.
  uint64_t endTimestamp;    // Timestamp of the last rec. frame.
};
```

The range of frame indices in the meta-data structure can be used to access the recorded frame through the following function:

RecordingItem& RecordGetItem(uint32_t index);

It takes a frame index and returns a reference to the respective recorded data, which has the following structure:

```
struct RecordingItem {
  std::array<uint8_t, RECORDING_JPEG_IMAGE_MAX_SIZE> binary;
  uint32_t size;
};
```

In this structure binary is an array containing the binary JPEG image data. The second item named size indicated the actual size of the binary data in bytes.

8.5. Network communication

The software on the EoT device and the configuration app are communicating through the MQTT protocol. In this demonstrator the MQTT broker is running on the EoT device. It was implemented as a reusable library in WP3. This demonstrator uses that implementation and wraps it in a class called Broker, which acts as a MQTT broker and client at the same time. It is defined as follows:

```
class Broker
{
public:
  static Broker& getInstance();
  void createWifi(const char* ssid, const char* password);
  void connectToWifi(const char* ssid, const char* password);
  void disconnectWifi();
  void start();
  void sendMessage(const char* topic,
                   const char* messageData,
                   uint32_t messageDataLength);
  bool listen();
  void stop();

private:
  Broker() = default;
};
```

The class manages everything from the Wi-Fi hardware to handling incoming messages. To this end the class implements the singleton pattern to ensure that at any given time at most one instance exists. This ensures the used hardware resources (i.e. the Wi-Fi chip) are not concurrently used. The individual methods are explained in the following.

8.5.1. Class instantiation

Access to the only instance is provided through the static getInstance method, because the constructor is private in accordance with the single pattern. This way the class can only be instantiated by itself.

8.5.2. Wi-Fi management

The wireless connection is managed with three methods. The first one createWifi creates a wireless access point with the provided SSID. The wireless communication is encrypted by the WPA standard and the specified password. The second method connectToWifi connects the EoT device to an existing WPA encrypted wireless access point with the specified SSID and password.

8.5.3. MQTT communication

The MQTT broker is started and stopped by the respective start and stop methods. It requires a working Wi-Fi setup either by establishing an access point or by connecting to one. The start method will also make sure that the client is subscribed to the relevant MQTT topics defined for this demonstrator. MQTT messages can be sent through the use of the sendMessage method. It requires a MQTT topic as zero terminated string, the MQTT message itself a string (messageData) with known length (messageDataLength). In order to receive messages, the listen method needs to be called periodically. The call is relayed to the MQTT broker implementation, which will check the receive buffers in the Wi-Fi chip for new data and process any incoming messages. Specifically, it will check the MQTT topic and relay the messages to all subscribers of that topic. In case of topics relevant for the peephole app it will relay those to the broker class. The messages in this demonstrator are encoded as JSON documents. The broker class will parse these messages and trigger the relevant reactions.

9. CONCLUSIONS

In this document, the deliverable software for the EoT board part of the Peephole demonstrator has been presented and documented. The software itself has been made available to the reviewers on a GitLab server. The software running on the EoT Board is only one part of the demonstrator and the second part is presented in document “D4.2 – Peephole demonstrator Configuration App”. After describing the demonstrator, the software has first been presented, then a complete documentation of the API of the software has been provided.

- End of document -