cAER

A framework for event-based processing on embedded systems

Bachelor of Science in Informatics, Software Systems Thesis

submitted by:
Luca Longinotti
Zürich, Switzerland
Matriculation number 08-920-142

In collaboration with:
Department of Informatics, UZH
Prof. Dr. Davide Scaramuzza
Institute of Neuroinformatics, UZH/ETHZ
Prof. Dr. Tobi Delbruck

Supervisor: Christian Brändli

January 31, 2014

Abstract

A new class of sensors, inspired by biology, promises significant savings in terms of power consumption and computation. But for these savings to really be meaningful, the systems to which such sensors are connected for data processing, must exhibit low-power consumption too. A new software architecture has to be created to both run efficiently on such systems and take advantage of the event-based nature of these sensors' output. This thesis discusses the details of such an architecture and presents a working implementation, showing that the goals of portability to a wide range of systems and efficient resource usage have been successfully met.

Contents

Chapter 1

Introduction

A new class of neuromorphic sensors, so called because they take their inspiration from biology and especially the way the brain works, have been developed in the past few years, and are still an active area of research. Camera sensors, like the DVS128 silicon retina [10] in Figure 1.1, take their inspiration from the way the human retina works, and auditory sensors, such as the AER-EAR2 silicon cochlea [12], are inspired by how the human cochlea works. These new sensors promise significant advantages over current ones, mainly in the areas of power consumption, output redundancy and temporal resolution [13]. Systems such as flying robots can take advantage of these new sensors to react to events taking place in the world around them much quicker than with conventional methods and sensors, as shown in [3], and using less power, as discussed in [6].

The output of these sensors follows the address-event representation (AER) scheme, in which spikes are sent by the sensor chip, carrying information on what happened and when, to an external processing system. This method allows for low-power and low-latency communication [13]. These spikes, representing events, are only sent when something actually happens on the sensor, greatly reducing the amount of redundant information being produced, resulting in a significant reduction of computational costs on the processing system.

This particular nature of the output has led to the development of event-driven computation software, such as the jAER open-source project [8], whose visual output is show in Figure 1.2, that takes advantage of it to implement novel algorithms.

Ideally, the processing component of the system should consume little power itself, to be able to keep the advantages stated above, because requiring a full desktop system, burning hundreds of watts, to elaborate the data coming from these low-power sensors would work against one of their main benefits. Embedded systems are prime candidates to fill this role of power-efficient computation units, but they also pose significant challenges in terms of software design, owing to their often very limited computing resources.

The current jAER project is written entirely in the Java programming language, which already poses a significant problem: many embedded systems do not posses a working Java Virtual Machine (JVM) implementation, and even on those that do, the overhead imposed by it may already be prohibitive to the kind of applications people are interested in. Further, jAER is reliant on a graphical user interface (GUI) for interactive control and output visualization, a feature that is rarely present on target systems such as autonomous robots, and that again requires significant computational resources and

increases power consumption.

A new software needed to be written, that would take full advantage of the event-driven computation model, as well as being usable on a wide range of embedded systems, taking their limited available resources into account. This work explores just such a software component, its architecture and its implementation.

Figure 1.1: DVS128 camera sensor (image courtesy of iniLabs.com)

Figure 1.2: jAER visual output (image courtesy of jAER.sf.net)

Chapter 2

Related work

Event-based processing is becoming increasingly more important in the industry as a simple and efficient way to aggregate data coming from different sources, filtering it, finding patterns inside it, and reacting to them in some manner, be it visually presenting them, sending alerts to specialized personnel, or sending commands to a mechanical arm.

Several complex event processing (CEP, [5]) software products exist for big data applications, such as Esper [5] or spChains [1]. These are geared towards large enterprise IT systems, focusing on queries, correlation, aggregation and filtering over large networks of systems, and are not considered relevant for this work.

The only software that can be considered truly related to the very specific aims of this work is jAER, from which significant inspiration was taken.

2.1 jAER

The jAER project [8], started in 2007, is an event-processing framework for neuromorphic sensors written entirely in Java, initially developed by the Institute of Neuroinformatics at UZH/ETHZ, and gradually expanded over the years with contributions from all over the world, thanks to being open-source.

It focuses on supporting mainly silicon retinas and cochleas, enabling easy access to all possible options through a graphical user interface, and visualizing and processing their output in interesting ways, useful both for research and for the development of various applications, such as a pencil balancing robot [4] or an object tracking software [7].

Chapter 3

Architecture

A novel event-driven software architecture had to be thought of to meet the main goals of this thesis: portability to a wide range of embedded systems, efficiency of resource usage, and enabling users and developers to easily create complex applications, leveraging reusable code and powerful, industry-standard application programming interfaces (APIs).

A work pattern of getting events from an input, the *source*, then processing and filtering them, and finally moving them to an output, the *drain*, and repeating those steps again and again (usually in a loop), is common to most event-based processing software, and also forms the basis for the architecture described here.

3.1 Use cases

Determining the use cases for new software represents one of the first steps to take on the road to its realization. Knowing where the software is going to be deployed, and what kind of usage it will be subjected to, has far-reaching consequences on its architectural decisions.

The main target platforms were determined to be consumer-grade embedded systems, such as the Raspberry Pi, the PandaBoard, the Parallella or the Odroid embedded computers.

Between one and four external sensors would be connected to such systems via USB (Universal Serial Bus) interface, the most widely used interface for the current generation of neuromorphic sensors, as well as for a large range of other sensors that are available in the consumer and professional markets. The number of simultaneously connected sensors would in any case always be low, given the small number of I/O ports that embedded systems usually offer.

Such systems would predominantly run some small, optimized version of the Linux operating system.

3.2 Requirements

Based on the use cases above, the project goals and the mentioned issues (in chapter 1) with the jAER framework, a number of requirements quickly emerged:

- able to run on a wide range of low-power, embedded systems
- small memory footprint, conscious usage of CPU cores, no assumption that multiple of them will be present
- able to run on the Linux operating system
- able to communicate with external USB devices
- focus on processing events generated by connected sensors
- no focus on visualization, no dependency on any graphical user interface
- network-enabled to communicate easily with other systems and frameworks (like jAER) or forward data to other processing nodes
- able to run without supervision for long amounts of time
- configurable without the need for graphical user interfaces, possibly configurable remotely
- reusable code: framework structure and modularity

3.3 Programming language

The choice of an appropriate programming language is central to any software project. As discussed in chapter 1, the Java programming language was found to be ill suited for the task at hand. In fact, any kind of interpreted or virtual machine based language would add similar overhead. In the end, the C programming language was selected, for a number of reasons:

1. it's a minimal, low-overhead, low-level language.
2. wide support for it is available on all kinds of hardware; indeed many embedded systems or micro-controllers only have a C compiler and environment available.
3. it is well supported by all manners of different tools, from integrated development environments (IDEs) to static code analyzers.
4. it enjoys an enormous ecosystem of libraries, providing access to a large range of software and hardware functionality, many of clear interest for computational workloads (OpenCL, CUDA) or computer vision ones (OpenCV).

3.4 Events

What an event represents, how to access it, the format of the data it contains, its storage requirements and memory layout, are all fundamental questions for any event-based processing software.

Events need to have several characteristics to meet the requirements determined before in section 3.2:

- low memory consumption
- fast data access
- extensibility to future devices
- low iteration overhead
- low deletion overhead

When defining the new event format, a careful balance had to be maintained between memory consumption and amount of information that could be stored regarding future extensions. While it might be tempting to just use two 4-byte integers to encode the X and Y dimensions of an event coming from a camera pixel, to support devices with up to 4 GigaPixels per axis in theory, such sensors don't exist and won't exist for years to come, and this is even more true for the sensors developed at the Institute of Neuroinformatics, which have a reduced resolution due to increased pixel size and other factors. It then makes more sense to put both the X and Y values in one 4-byte integer, allocating each a fixed amount of bits, to conserve memory and especially storage bandwidth and capacity when transmitting events over the network or storing them to disk. With this method, it's possible to really minimize memory usage for each type of event individually.

Fast data access is still guaranteed, since, at most, accessing data stored with the above method would require an integer shift and AND operation, while storing data an integer shift and OR operation. Sometimes just an AND or OR operation, when the shift amount is zero. Such bit-wise operations are extremely fast on modern CPU architectures, usually taking just 1 CPU cycle. Intel for example reports latency values of 1 cycle for AND/OR instructions and 1 cycle for shifts (SHL/SHR instructions) for modern Haswell processors and for its Silvermont architecture (successor to Atom for low-power scenarios) [9].

To enhance iteration performance, and generally reduce the load that moving events from one processing stage to the other incurs; events are always grouped together into packets, and kept adjacent in memory. Going from event to event just requires a pointer increment, and moving a packet between processing stages just involves passing around a pointer to it.

A further benefit of grouping events into packets is that a packet header can be defined and used to store useful information about the events it contains, such as their type or source, in a non-redundant way, instead of putting that information with each event separately, further saving memory.

Finally, concerning the low deletion overhead, instead of implementing the removal of events by copying only the valid ones to a new packet, or shifting them around inside

the current packet, a 1-bit boolean flag has been made a mandatory part of each event, indicating if the event is to be considered valid (flag is 1) or invalid (flag is 0). A side-benefit of this approach is that it is still possible for later processing stages to examine invalidated events, which might be interesting for example for statistics.

All fields, both in the header and in the events themselves, are to be considered little-endian. The reason for this is simple: 95% of architectures on which this software is expected to run are in fact little-endian (x86, x86-64, ARM), and it would only impose needless overhead to change the data format to big-endian instead. Also, all fields and structures are tightly packed together, to avoid wasting memory on alignment and padding.

The following code shows the common data structure representing the packet header for all event packets:

```
struct caer_event_packet_header {
    uint16_t eventType;
    uint16_t eventSource;
    uint32_t eventSize;
    uint32_t eventTSOffset
    uint32_t eventCapacity;
    uint32_t eventNumber;
    uint32_t eventValid;
};

typedef struct caer_event_packet_header *caerEventPacketHeader;
```

The *eventType* field stores the type of event that is contained in the packet. Currently the following event types are defined:

Event type	Code	Description
SPECIAL_EVENT	0	Exceptional event (timestamp changes, external triggers, ...).
POLARITY_EVENT	1	Pixel polarity change (ON or OFF, representing the relative light intensity increase or decrease of a pixel).
SAMPLE_EVENT	2	Digital sample from Analog-Digital Converter (ADC).
EAR_EVENT	3	Cochlea event (ear, channel, ganglion and filter).
FRAME_EVENT	4	RAW digital image (packed ADC sample sequence).
IMU6_EVENT	5	Inertial Measurement Unit data, 6 axes (accel., gyro).
IMU9_EVENT	6	Inertial Measurement Unit data, 9 axes (+ compass).

Table 3.1: Event types

The *eventSource* field encodes a unique numeric ID that represents the input from which the event originates. The ID is unique inside a data processing loop and can be used to retrieve information on the input in later processing stages that work on a particular packet.

The *eventSize* field gives the size in bytes of an event and can be used to calculate where the next event is in memory, without having to check its type and interpreting its

memory as that type. This is important for efficient generic programming.

The *eventTSOffset* field gives an offset in bytes from the start of the event, at which the main timestamp integer is located. This is used to get the timestamp without having to check the type of event: all events must have a timestamp, which has to be a 4-byte unsigned integer. So, to get any event's timestamp, it is enough to add this offset to the event pointer and cast the resulting address to be a pointer to a 4-byte unsigned integer. Coupled with the previous *eventSize* field, it is then possible to get the timestamp of any event in any packet reliably, without having to concern oneself about the packet type and possible casts or other memory interpretation issues.

The last three fields are concerned with the number of events contained in a packet: *eventCapacity* tells how many events this packet could maximally hold and is determined by the amount of memory that was initially allocated to the packet (packets don't grow or shrink, to avoid re-allocation overhead); *eventNumber* gives the amount of meaningful events inside the packet, and is always equal or smaller than the capacity; and finally *eventValid* contains the number of events that are still valid, a value which is always equal or smaller than the number of events. Thanks to this information, it is possible to know how far one can iterate over a packet, or how much memory would need to be allocated if, for example, one wanted to copy only the valid events to another memory location.

The following code defines the data structures behind a polarity event and its packet (see table 3.1 for a definition of polarity event), probably the most frequently used type:

```
struct caer_polarity_event {
    uint32_t data;
    uint32_t timestamp;
};

typedef struct caer_polarity_event *caerPolarityEvent;

struct caer_polarity_event_packet {
    struct caer_event_packet_header packetHeader;
    struct caer_polarity_event events[];
};

typedef struct caer_polarity_event_packet *
   caerPolarityEventPacket;
```

This basic structure is common to all event types: the event packet is composed of the packet header, which was examined in detail above, and an array of events of that type. The packet header field is called packetHeader and is required to be the first field, which allows the interpretation of a pointer to the packet as a pointer to the header and vice-versa, since the first field is always guaranteed to live at offset 0. Each event has a 32-bit integer timestamp, as well as a number of bytes (in this case four) reserved for the actual event data. Please note that the first bit (bit 0) of the first byte of an event is reserved for the validity boolean flag, to allow uniform access to it, without requiring any ulterior kind of interpretation or type-casting.

3.5 Framework

The stated goal of this work is to develop a new event-driven data processing framework suitable for low-power applications and high-speed sensors. Such a framework should provide common functionality such as configuration management and logging, as well as handling all the data processing setup, such as starting up threads, processes, inputs and outputs. It should offer the user a simple, well-defined way to put together the parts of the framework he needs for his application, and be easy to deploy across various systems. It should further be easily possible to extend the functionality of the framework with self-written, application-specific code.

To this end it was decided to implement cAER as a collection of basic framework functionality and modules, which could easily be plugged and extended by the user. In the end, everything would compile down to one binary, which could then be deployed and executed on the target systems.

A broad overview of the resulting architecture is shown in Figure 3.1. The various components will be explained in detail in the following sections, first from a high-level architectural point of view, and then from a more implementation-minded point of view in chapter 4.

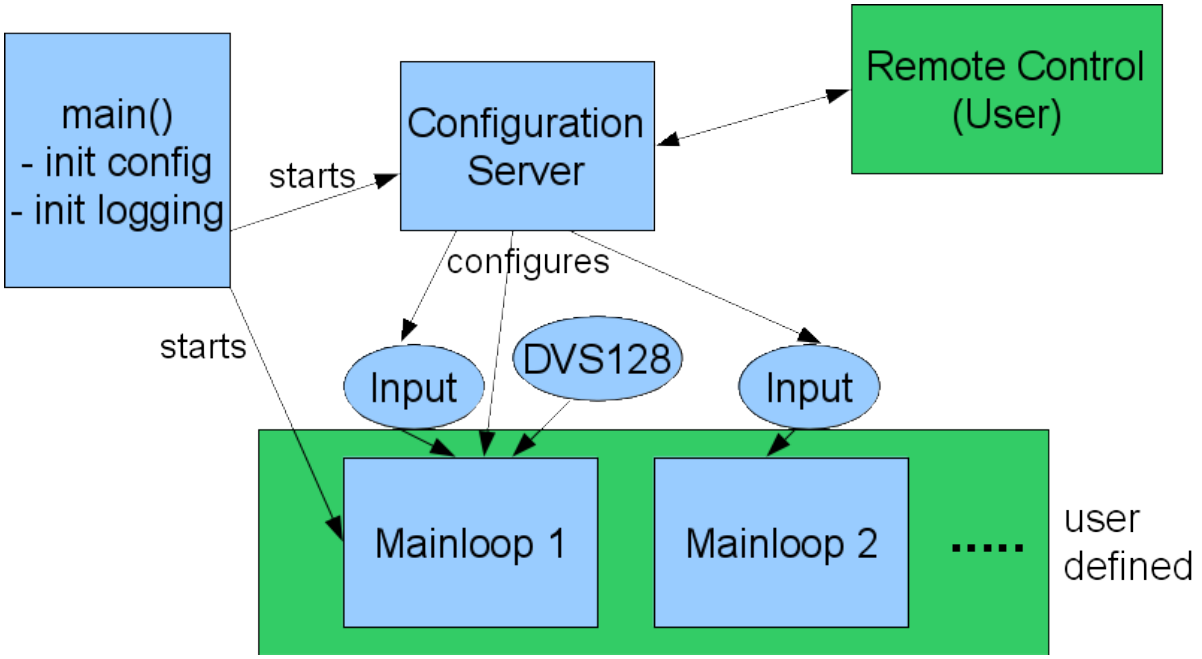

Figure 3.1: cAER architecture overview

3.6 Process and Thread structure

The main program thread is responsible for setting up configuration and logging, and then start the rest of the framework. Once that is done, it just waits for the orderly shutdown of those parts that it initialized, and is then the last thread to exit. This

provides a clear path for orderly shutdown; if the main thread were not kept around, who would be responsible for waiting on all the other threads to ensure a clean program termination? This way, the thread starting everything is also responsible for then waiting on everybody to finish their work. Keeping this thread around is no resource drain, as it just checks a variable and then goes back to sleep for one second in a loop.

Given the limited resources of the target systems, especially in terms of number of CPU cores, it was decided to only start one process, and keep the number of threads inside it to the bare minimum which would still provide a good logical separation of tasks and functionality. As such, each data processing loop runs inside its own thread, as well as each input. A further thread is started for the run-time configuration system, to allow users to modify system settings at any time, while the other parts of the program are running.

The total number of running threads is thus given by:

- main thread (1)
- run-time configuration system (1)
- data processing loops (N)
- inputs (at least N)

So, on a minimal setup with one data processing loop getting data from one input, a total of four threads would be started.

3.6.1 Daemon mode

To support the needs of server-like, always-on applications, a daemon mode was introduced. This allows the program to easily detach from the current terminal and run in the background, enabling it to be started on a system and then just let run for weeks or months without any user interaction.

3.7 Configuration

One of the foremost concerns when writing a framework that compiles down to a binary is how to configure it at start-up and at run-time. It would be tedious to stop the application, then manually change parts of the source code and recompile, every time one wanted to change a setting somewhere, so a mechanism has to be provided to allow changes to be done after the binary is compiled and once the program is running.

It is also important to determine how each part of the program would be able to access such a configuration system. A quick way to access the configuration for single parts of the code is as important as having a global view of all configuration settings for users to manipulate.

To this end, the Super Simple Hierarchical Storage (SSHS, 3.7.1) library was written and then employed. A configuration file from which to load the initial settings is also part of this system.

3.7.1 SSHS

SSHS is a stand-alone library that provides in-memory, hierarchical, structured information storage capabilities. It is inspired by the Java Preferences system, but with a better control of where its content is stored and when it's written out to disk, which happens exclusively on the explicit request of the user. The content of the hierarchical storage can be exported to XML for easy sharing, improved human readability and editing, and such XML files can also be imported back to initialize or update the storage's content.

It implements a tree-like structure, with nodes that contain links to other nodes (their children), as well as attributes. To retrieve a specific node, absolute paths from the root node (*/this/is/an/absolute/path/*) or relative paths from an ancestor node (*a/relative/path/*) can be utilized. Functions to verify the existence of nodes are also present.

The attributes contain the actual data, which can be of eight types: boolean, byte (8-bit integer), short (16-bit integer), int (32-bit integer), long (64-bit integer), float, double and string. Each attribute is uniquely identified by its type in combination with a string identifier. Functions to get and set attribute values are provided, such as GetBool() or PutInt(), as well as functions to test the presence of attributes.

Getting an attribute is only possible if the attribute was declared beforehand, by either loading it from an XML file or by setting it to a defined value with the various Put() or PutIfAbsent() functions. This ensures that a valid value is present, and avoids having to specify a default value on each Get() call, as has to be done, for example, in the Java Preferences system.

Listeners can be added to nodes to react to changes to both the node's children and to its attributes.

3.7.2 Remote configuration

SSHS (see section 3.7.1) provides the actual mechanism to store, notify, and, together with the modules, apply configuration changes. Given the nature of cAER, which might run as a background process, an external interface was required for the user to actually tell the program what configuration changes should be done.

This interface was implemented as a TCP network server, listening on a user-configurable IP address and port combination, and allowing concurrently connected users to operate on the SSHS back-end and change the values of settings (attributes). TCP was chosen for its reliability and ordering guarantees, as it would be unacceptable to loose configuration updates in transit over the network, or have them applied in an arbitrary order.

To communicate the details of a request, a custom protocol was implemented. Control messages have the following format:

- 1 byte for the action
- 1 byte for the type (representing SSHS types)
- 8 bytes for length information:
  - 2 bytes extra field length
  - 2 bytes node length
  - 2 bytes key length
  - 2 bytes value length
- then up to 4'086 bytes split between the extra field, the node, the key and the value strings. Each part, if present, must be NUL terminated, and the length shall also include the terminating NUL byte.

This results in a maximum message size of 4'096 bytes (4 KB).

The extra field is currently unused, and as such its length should always be zero. Its purpose is to allow future enhancements, such as authentication of requests.

The following requests can be made to the configuration server:

Action	Code	Type	Node	Key	Value
NODE_EXISTS	0	0 (unused)	absolute path	no	no
ATTR_EXISTS	1	any	absolute path	key string	no
GET	2	any	absolute path	key string	no
PUT	3	any	absolute path	key string	value string
GET_CHILDREN	5	0 (unused)	absolute path	no	no
GET_ATTRIBUTES	6	0 (unused)	absolute path	no	no
GET_TYPES	7	0 (unused)	absolute path	key string	no

Table 3.2: Configuration server requests

The response from the server follows a simplified version of the request protocol:

- 1 byte for the action
- 1 byte for the type (representing SSHS types)
- 2 bytes message length
- then up to 4'092 bytes of response message. The message must again be NUL terminated, if present, and the NUL byte shall be included in the length calculation.

This results again in a maximum message size of 4'096 bytes (4 KB).

The server always responds either with an error (action code 4) or with a valid answer, with the same action code as the initial request. The following table explains the response message format for the various actions that could have been submitted during the request phase:

Action	Code	Type	Message
ERROR	4	string	error string
NODE_EXISTS	0	bool	true/false
ATTR_EXISTS	1	bool	true/false
GET	2	same as request	result string
PUT	3	bool	true
GET_CHILDREN	5	string	concatenation of string child names
GET_ATTRIBUTES	6	string	concatenation of string attribute keys
GET_TYPES	7	string	concatenation of string type names for a key

Table 3.3: Configuration server responses

Thanks to this scheme, it is easily possible to implement an automatic discovery of the topology of the configuration back-end structure with the GET_CHILDREN, GET_ATTRIBUTES and GET_TYPES actions, in addition to standard check, get and set functionality, thanks to the NODE_EXISTS, ATTR_EXISTS, GET and PUT actions.

3.8 Logging

A logging system is a key component of any framework, since being able to output messages in a consistent way, with additional information such as the current system time, is quite important to inform the user of what's happening and assist in debugging failures.

Program code can call the *caerLog()* logging function, giving it an importance level and a message. The message will only be logged if it happens to be more important than

a user-set threshold, allowing efficient filtering of unimportant and undesired messages. The following log levels are recognized:

Log level	Code
LOG_EMERGENCY	0
LOG_ALERT	1
LOG_CRITICAL	2
LOG_ERROR	3
LOG_WARNING	4
LOG_NOTICE	5
LOG_INFO	6
LOG_DEBUG	7

Table 3.4: Log levels

All messages are written out to a log file, to ensure their persistence, for later analysis. For more immediate visual feedback, messages are also written out to the console, if cAER is not in daemon mode (see section 3.6.1), which would render this impossible.

3.9 Mainloops

As mentioned at the start of chapter 3, the core of any kind of data processing architecture is represented by the following three steps:

- get data from (one or more) inputs (*sources*)
- process / filter data
- send data to (one or more) outputs (*drains*)

Such processing usually happens in a loop, getting new data as soon as being finished with the old one or at fixed intervals in time.

In event-driven data processing, and especially in low-power scenarios, the expectation is that those three steps only happen when the input actually has generated meaningful data that can be processed, not before, and possibly also not after, to still guarantee acceptable reaction times to new events. The continuous repetition of the tree above steps still happens, but the start condition slightly changes to hinge on the availability of new events to process.

A simple and efficient implementation is still represented by a loop, which shall run the processing functions only when there is more data, or else wait for a small time constant, before checking the availability of data again.

In cAER this process is simply called the Mainloop, short for Main-Data-Processing-Loop. Mainloops are defined by the user and connect inputs to processing modules and finally to outputs. They represent the actual operations the user wants to repeatedly see happen on the data.

The thread-function that runs a Mainloop looks like this (in pseudo-code):

```
while (run_system) {
    if (data_is_available) {
        execute_mainloop_definition();
    }
    else {
        sleep(1 ms);
    }
}
```

3.9.1 Asynchronous inputs

To be able to run the Mainloop only when data is available, something else has to be running and making that data available. The input modules have a secondary part, which is executing inside a separate thread, and is responsible for getting the bare data from connected devices, network streams or files, and signaling its availability to the Mainloop (the data_is_available variable in the pseudo-code above).

The input threads themselves employ techniques, such as blocking I/O, or asynchronous USB transfers, that also limit activity to only when data is available at the device or network level, to avoid unnecessary overhead. At this point, minimal processing is performed to interpret the data and format it according to the event packet definitions from section 3.4, and then the data is made available to the Mainloop. The actual exchange of data between the two threads happens by means of a special lock-free variant of a ring-buffer, a high-performance, array-based data structure.

Memory allocated by the input threads is later automatically reclaimed after the current cycle of the Mainloop has completed, completely removing this particular concern from the user.

3.10 Modules

Modules form the core of all data processing within cAER. They implement ways for data to be made available (input modules), for it to be elaborated (processing modules) and for it to be sent somewhere else (output modules).

Each module is composed of a main function, which takes as parameters an unique module ID, as well as any needed input and output parameters, and this is the function called by users when defining the Mainloops. This function is pretty simple in its content: it's responsible for retrieving the persistent module data and then calling the module state machine.

The module state machine takes care of initializing, running, configuring and shutting down a module, in such a way that each module can be disabled and reactivated while the Mainloop is still running, to easily test the effects that a module's presence or absence has on the data stream. Each module can also allocate a region of memory to store permanent state that is to be accessible on each Mainloop cycle, such as maps, numbers, lists and so on. Every module defines up to four functions that the state machine calls to do its work; the only required function is the *moduleRun* function. The other three functions

are optional, but most modules will want to implement them: the *moduleInit* function takes care of initializing a module, ensuring all configuration values are present and have appropriate default values, initializing any memory and variables, etc.; its counterpart *moduleExit* is responsible for cleaning up any resources at shutdown. The remaining function, *moduleConfig*, contains the code that shall be executed when one of the module's configuration settings is changed by the user. All module functions must be able to complete their work in one Mainloop cycle, they cannot for example shut down in multiple stages. This is also enforced by these functions being required to not return any value that might indicate their failure or success: they must always succeed. The sole exception is the *moduleInit* function, which may return a boolean value of *false*, if initialization was not possible (due to a missing device, not enough memory, wrong configuration settings, ...), in which case initialization will be reattempted on the next Mainloop cycle. This does not allow multi-stage initialization, as there is no persistent memory between successive *moduleInit* calls that could be used to implement it.

The following pseudo-code illustrates the working of the module state machine, as is called on every Mainloop cycle:

```
// 'moduleStatus' contains the current module status.
// 'running' specifies the wanted status: running or not.

if (moduleStatus == RUNNING && running == YES) {
    if (configuration was updated) {
        if (moduleFunctions->moduleConfig is defined) {
            moduleFunctions->moduleConfig(moduleData);
        }
    }

    moduleFunctions->moduleRun(moduleData, argsNumber, args);
}
else if (moduleStatus == STOPPED && running == YES) {
    if (moduleFunctions->moduleInit is defined) {
        if (!moduleFunctions->moduleInit(moduleData)) {
            return;
        }
    }

    moduleStatus = RUNNING;
}
else if (moduleStatus == RUNNING && running == NO) {
    moduleStatus = STOPPED;

    if (moduleFunctions->moduleExit is defined) {
        moduleFunctions->moduleExit(moduleData);
    }
}
```

Chapter 4

Implementation

The following chapter details cAER from a user's point of view, explaining several more technical and implementation oriented matters, that directly relate with the use and deployment of the software on a target system. It refers to version 0.9.4 of the software, released on January 30, 2014 under the BSD license, and available for download at `https://github.com/llongi/caer/`.

4.1 Structure

The implementation is divided into multiple files and directories, to allow for a clean separation of concerns.

The main.c source-code file connects all the parts into a working program, as detailed in section 4.5.

The base/ directory contains all the framework functionality, explained in section 3.5 and onwards.

The events/ directory houses all the definitions of the events and their packets, according to the format discussed in section 3.4. All event definitions, as well as the common packet header definition, are implemented as stand-alone, reusable header files, to promote and simplify integration with other projects, which will easily be able to interpret data coming from cAER by including these files.

ext/ contains external libraries that are used throughout the program, as well as internal ones such as SSHS that were developed specifically for this project (see section 3.7.1).

The modules/ folder contains all the input (section 4.6), processing (section 4.7) and output (section 4.8) modules, that a user may employ for his application.

Finally, the utilities/ folder contains small helper programs, detailed in section 4.4.

4.2 Installation

Given the stated goal of portability, the number of dependencies on external software had to be kept to a minimum, to avoid ones that might not build on a certain platform or require special hardware to run.

The following software is required to successfully compile cAER:

- A C compiler supporting the C99 language standard, as well as some GCC extensions (function/type attributes, inline assembler, synchronization intrinsics, thread-local storage), is needed to compile the code. GCC is currently the only compiler that meets all these requirements, starting with version 4.7.
- A POSIX environment (Portable Operating System Interface), to supply threading, networking and file-system access functionality.
- cmake 2.8+ for easy build management and cross-platform compilation support.
- Mini-XML 2.7+ for XML file parsing, used for the XML configuration file import and export features.
- DVS128 module: libusb 1.0.16+ (recommended 1.0.18+) to access USB devices from the application.

The installation procedure, once all dependencies are satisfied, is composed of four simple steps:

1. Get program sources
2. cmake .
3. Edit the main.c source-code file, to adjust the *main()* function and define a custom Mainloop (further information can be found in sections 4.5.1 and 4.5.2)
4. make

At this point, the *caer* executable, as well as its helper utilities, are ready for deployment and execution.

4.3 Configuration

On the first execution of the *caer* program, all configuration will be initialized to its default values.

Manipulating configuration values is possible either by using the *caerctl* utility (see section 4.4.1) at run-time, or by editing the XML configuration file that is written each time the program shuts down.

The following listing shows an excerpt from the configuration file in question:

```
 1:<sshs version="1.0">
 2:   <node name="" path="/">
 3:     <node name="1" path="/1/">
 4:       <node name="2-FileOutput" path="/1/2-FileOutput/">
 5:         <attr key="directory" type="string">/home/caer</attr>
 6:         <attr key="prefix" type="string">caer_out</attr>
 7:         <attr key="validEventsOnly" type="bool">true</attr>
 8:       </node>
 9:       ...
10:     </node>
11:     ...
12:   </node>
13:</sshs>
```

The first line specifies that this file conforms to the SSHS format specification (from section 3.7.1), version 1.0. After that, the tree-like hierarchical structure is clearly visible, with the root node (line 2) holding all Mainloops (line 3), which in turn hold all the modules (line 4). The *<attr>* tags contain the configuration values themselves, the *key* attribute identifies and names the value, the *type* attribute specifies its SSHS type, and the tag's value itself represents the effective configuration value.

In the above example, the *validEventsOnly* configuration value is of boolean type, and can assume values of *true* or *false*.

To allow for consistent configuration updates inside the modules, changes to the configuration are only applied once per Mainloop cycle, by the module state machine, right before the actual data processing code is executed (see *moduleConfig* in section 3.10). Since changes to the configuration can happen at any time, due to the configuration server thread (see section 3.7.2), a mechanism had to be developed to notify the running modules that a configuration refresh was desired on their next execution. A single, atomic variable in each module is used for the purpose of asynchronously signaling configuration changes, allowing the module to quickly determine on each run if it needs to update its configuration and react to eventual changes. This provides an efficient inter-thread communication solution, instead of relying on expensive locks to synchronize some additional state variable. SSHS listeners are added to each module's configuration nodes, to set the above atomic variable appropriately when changes occur.

4.4 Utilities

Several command-line utilities are distributed together with the *caer* program:

caerctl is used to query and set configuration values while the *caer* program is running.

unixststat provides a quick way to test the local Unix socket client output and print information on incoming traffic from that output.

udpststat provides the same for the UDP network client output.

tcpststat again provides similar functionality for the TCP network server output.

4.4.1 caerctl

The *caerctl* utility takes exactly two arguments: an IP address and a port, to which a connection attempt will be made. At this address, a *caer* instance with a reachable configuration server should be running. If no arguments are specified, the default IP:port values of *127.0.0.1:4040* are used. Once *caerctl* has connected successfully with the remote configuration server, the following commands are available for the user to type in and submit:

node_exists: checks whether a node exists or not.

Usage: node_exists <node string>

attr_exists: checks whether a node's attribute exists or not.

Usage: attr_exists <node string> <key string> <key type string>

get: queries the current value of the specified attribute.

Usage: get <node string> <key string> <key type string>

put: sets the value of the specified attribute to the supplied one.

Usage: put <node string> <key string> <key type string> <value string>

quit or exit: disconnects from the configuration server and closes the program.

Usage: quit / exit

To simplify interaction with the user and not require him to remember all the configuration paths, automatic command-completion has been implemented, by using the GET_CHILDREN, GET_ATTRIBUTES and GET_TYPES actions from section 3.7.2. By pressing the TAB key, the user will be presented with a list of possible completions to select from. Further completions are cycled by pressing TAB again, until returning to the original starting point. Unambiguous completions are selected automatically; in case of multiple choices, the SPACE key confirms the current choice. Command history is also provided, one can navigate to previous commands and back with the UP and DOWN arrow keys, and the commands entered during previous *caerctl* sessions are saved to a file named *.caerctl_history* in the user's home directory, and made available again on the next session.

The implementation of such functionality requires support from an external library, three of them were evaluated: GNU Readline was the first choice and discarded because of the forced GPLv2 license, which might have prevented certain commercial and industrial users from seriously considering cAER. The mediocre documentation did not help. Editline was discarded because of problems in getting the contents of the currently being typed line of text, at least while in Readline-compatibility mode. Again, the scarce documentation did not help clear matters up. In the end the choice fell on Linenoise, a very small library that implements the most basic functionality needed, while also allowing easier integration into a project by just requiring direct inclusion of two small source-code files and avoiding any licensing restrictions by employing the BSD license.

4.4.2 unixststat

The *unixststat* utility takes exactly one argument: an absolute file-system path, where a local Unix socket will be created and configured in listening mode. If no arguments are specified, the default socket path value of */tmp/caer.sock* is used. A continuous stream of information on the incoming data packets will then be printed on the console.

4.4.3 udpststat

The *udpststat* utility takes exactly two arguments: a local IP address and a port, on which to start listening for incoming UDP packets. If no arguments are specified, the default IP:port values of *127.0.0.1:8888* are used. A continuous stream of information on the incoming data packets will then be printed to the console.

4.4.4 tcpststat

The *tcpststat* utility takes exactly two arguments: an IP address and a port, to which a connection attempt will be made. At this address, a *caer* instance with a reachable TCP network server output module (described in section 4.8.5) should be running. If no arguments are specified, the default IP:port values of *127.0.0.1:7777* are used. A continuous stream of information on the incoming data packets will then be printed to the console.

4.5 Application definition

The main actions required by a user to tailor the resulting *caer* binary application to his needs are:

- editing the *main()* function in the main.c source-code file
- writing an appropriate Mainloop function to process the data as needed and reference it in the main.c source-code file

4.5.1 main() definition

The main.c source-code file contains the top-level application entry point, where execution begins from. In C programs, this is represented by a function with the signature of *int main(int argc, char *argv[])*, which takes the number of command-line arguments given when starting the program as well as their content as parameters.

A suitable default implementation is already present and shown below, but the contents of the function can easily be changed to suit the more specific needs of the user.

```
int main(int argc, char *argv[]) {
1:      caerConfigInit("caer-config.xml", argc, argv);
2:      caerLogInit();

3:      caerDaemonize(); // Optional

4:      caerConfigServerStart(); // Optional (together with stop)

5:      struct caer_mainloop_definition mainLoops[2] =
            { { 1, &mainloop_1 }, { 2, &mainloop_2 } };
6:      caerMainloopRun(&mainLoops, 2);

7:      caerConfigServerStop(); // Optional (together with start)

8:      return (EXIT_SUCCESS);
}
```

The first line, calling the *caerConfigInit(const char *configFile, int argc, char *argv[])* function, is required and must always be the first line to appear. As the name implies, it initializes the configuration sub-system (based on SSHS, see section 3.7.1), which is later needed by all other components, and imports the initial configuration from the XML configuration file (see section 4.3), as well as setting it up so that any configuration changes will be written out to that file at shutdown.

It also tries to interpret any given command-line parameters and uses their values to set and override previous configuration settings. The command-line override takes the following format and can be repeated multiple times to override multiple options:
-o <node string> <key string> <key type string> <value string>

The configuration file-name parameter can be either an absolute path to a file or a single file-name, which is treated as residing inside the current working directory of the program. It is possible to disable the loading and saving of the settings to the configuration file by passing NULL as a value instead of a file-name.

The argc and argv parameters should be the same as the original arguments given to *main(int argc, char *argv[])*. It is also possible to disable command-line parsing and configuration overriding by specifying argc to be 0 and argv to be NULL.

The second line is again required and must appear in that order. It calls the *caerLogInit()* function, which initializes the logging sub-system and opens the log-file for writing. It gets the relevant settings from the configuration sub-system's */logger/* node; those are:

logFile the file where log messages are stored.

Type: string, Default value: <current working directory>/caer.log

logLevel the cut-off level for log messages. Messages of lesser importance are ignored.

Type: byte, Default value: 5 (corresponds to LOG_NOTICE as shown in section 3.8)

At this point it is possible to detach the application from the current terminal and have it continue running as a background process. This is done by calling the *caerDaemonize()*

function. Not calling it simply results in the program remaining attached and dependent on the current terminal.

Now it's time to start the configuration server, if such functionality is actually desired. If the configuration should not be modifiable at run-time, for example if a correct configuration file is already present, the *caerConfigServerStart()* function, and its counterpart *caerConfigServerStop()*, can simply not be called. If, on the other hand, *caerConfigServerStart()* is called, one must not forget to also call *caerConfigServerStop()* right before the application's exit point (usually the successful return from the *main()* function).

The configuration server stores its own settings in the */server/* node:

backlogSize maximum number of pending connections kept in the connection queue.

Type: short, Default value: 5

concurrentConnections maximum number of allowed simultaneously connected clients.

Type: short, Default value: 5

ipAddress IP address on which to listen for connections.

Type: string, Default value: 127.0.0.1

portNumber port number on which to listen for connections.

Type: short, Default value: 4040

Finally, once all the support infrastructure has been brought up, the Mainloops can be defined and started. To define Mainloops the user has to create an array of *caer_mainloop_definition* structures and initialize it accordingly (as in line 5 above). The size of the array should correspond to the number of Mainloops one wants to define and start. Each array element should have its two fields initialized: the first is of type uint16_t (16-bit unsigned integer) and contains a user-chosen, unique numeric ID that helps in identifying the Mainloop later on, for example when accessing its configuration. The second field is a pointer to the actual Mainloop function. Once the Mainloops have been defined, starting them requires calling the *caerMainloopRun(struct caer_mainloop_definition (*mainLoops)[], size_t numLoops)* function, which takes as arguments a pointer to the above array containing the definitions, and the number of Mainloops that should be started from those definitions. A zero value will cause the function to return immediately without doing anything. The various Mainloops are then started sequentially and will continue to run until terminated.

It is possible to shut down the whole program and all Mainloops by changing the special configuration setting called *shutdown* of the root node / to the value *true*; this is a non-reversible operation.

4.5.2 Mainloop definition

Mainloops specify the flow of data and how it should be processed. At their core, they are simply functions that will be called repeatedly in a loop (see the pseudo-code in section 3.9 for more details).

Defining them is quite simple; one just has to write a function, taking no arguments and returning a boolean value, which indicates if this Mainloop should be called again on the next iteration, or if it should instead be terminated. In most cases, users will want to return the value *true* all the time.

```
static bool mainloop_1(void) {
    // Typed EventPackets contain events of a certain type.
    caerPolarityEventPacket dvs128_polarity;
    caerInputDVS128(1, &dvs128_polarity, NULL);

    // Output to network via UDP.
    caerOutputNetUDP(2, 1, dvs128_polarity);

    return (true); // If false, processing of this loop stops.
}
```

Every Mainloop has a special configuration setting called *shutdown*, which is another way for the user to terminate its execution, by changing that setting to the value *true*. Exiting a Mainloop is a non-reversible operation, since once shut down, the Mainloop will disable all its modules, free its memory and terminate the thread on which it was running.

The content of a Mainloop should usually be quite simple and straightforward: packets of events are defined, input modules are called, with pointers to packets to fill them, or a NULL value if that type of input is not desired, packets are then passed to processing modules, until they are given to output modules to send away over the network or write to a file somewhere on disk.

4.6 Input modules

Input modules provide events to a Mainloop.

Each input module defines a configuration sub-node called */sourceInfo/*, which contains constants and other read-only information about the module and the events it produces. This can then be accessed by other modules to find out values such as the size in pixels of a camera or the number of audio channels available.

One limitation exists: the same input module accessing the same underlying event producer (hardware device, network stream, file, ...) cannot exist more than once across all Mainloops. Sharing the same input over multiple Mainloops would require expensive copying and distribution of data, which is best avoided in resource constrained environments. Hardware device access modules usually try to get exclusive access to their device, rendering this kind of error impossible: a second instance of the same module accessing the same device would simply not be able to open it.

4.6.1 DVS128

```
void caerInputDVS128(uint16_t moduleID, caerPolarityEventPacket
    *polarity, caerSpecialEventPacket *special);
```

This input module allows a DVS128 camera to be connected and events to be captured from it. As can be seen from its function signature above, it generates polarity event packets and special event packets, by putting their address into the given pointer. If no event packets of a certain type are available at the moment, NULL is assigned instead. This is done to not delay the current Mainloop cycle if certain types of produced events are much more scarce than others. To indicate that one is not interested at all in an event packet of a certain type, NULL can be passed to the function instead of a pointer.

As explained in section 3.9.1, this module starts a separate thread, the Data Acquisition Thread, that reads the events from the hardware device and transforms them into suitable event packets, as defined in section 3.4, and then passes them on for further processing to the Mainloop. This module uses the libusb library for accessing the DVS128 camera, which is connected to its host via a USB 2.0 interface. For higher performance, asynchronous USB transfers are employed.

Multiple cameras can be connected and accessed by calling this module multiple times: each instance will then be connected to one camera. The order in which cameras are accessed is based on the host USB bus and device numbers, lower numbers being accessed first by module instances appearing before in a Mainloop. Across different Mainloops, the order is undefined.

If no camera is currently connected to the host system, the module will attempt to connect to one on each Mainloop cycle. Conversely, if a connected camera is physically disconnected, the module will detect this and shut itself down.

The DVS128's configuration node offers the following settings for customization:

bufferNumber the number of the USB buffers currently being used for asynchronous data transfers with the device.

Type: int, Default value: 8

bufferSize the size in bytes of the USB buffers currently being used for asynchronous data transfers with the device.

Type: int, Default value: 4'096 bytes

dataExchangeBufferSize the number of elements the ring-buffer used to transfer event packets from the Data Acquisition Thread to the Mainloop can hold before it gets full and starts rejecting newly produced packets.

Type: int, Default value: 64

polarityPacketMaxInterval the maximum time interval in μs between the first and the last event in a polarity event packet, after which it is transferred to the Mainloop.

Type: int, Default value: 5'000 μs

polarityPacketMaxSize the maximum number of events in a polarity event packet, after which it is transferred to the Mainloop.

Type: int, Default value: 4'096

shutdown enables or disables this module.

Type: bool, Default value: false

specialPacketMaxInterval the maximum time interval in μs between the first and the last event in a special event packet, after which it is transferred to the Mainloop.

Type: int, Default value: 1'000 μs

specialPacketMaxSize the maximum number of events in a special event packet, after which it is transferred to the Mainloop.

Type: int, Default value: 128

A sub-node called */bias/* is also provided to set the DVS128's bias currents. The following biases are specified as integer settings: *cas, diff, diffOff, diffOn, foll, injGnd, pr, puX, puY, refr, req, reqPd.* Their default values are based on the official fast bias settings for the DVS128 from the jAER project [8] repository (DVS128Fast.xml in trunk-/biasgenSettings/DVS128/).

The */sourceInfo/* sub-node defines the following values that can be queried by other modules:

sizeX the X axis resolution for this camera in pixels.

Type: short, Default value: 128 pixels

sizeY the Y axis resolution for this camera in pixels.

Type: short, Default value: 128 pixels

4.7 Processing modules

Processing modules modify the data contained in event packets, or can even create new ones with new types of data, based on their input.

Any processing module should be prepared to have a NULL value instead of an event packet being handed to it, this can happen for example when no event packets of that type were available from the preceding input modules at that moment in time.

4.7.1 Background Activity Filter

```
void caerBackgroundActivityFilter(uint16_t moduleID,
    caerPolarityEventPacket polarity);
```

The Background Activity Filter takes polarity event packets and invalidates any polarity event that is not supported by other polarity events in its neighborhood within a certain time window. It implements this by looking up the latest timestamp before the current one in a map and comparing the difference against a fixed threshold value set by the user. This considerably reduces the background noise generated by the camera and helps in clearing up its output.

The following settings are recognized:

deltaT maximum time-difference in μs between the current event and the last supported activity by one of its neighbors, after which events are declared invalid.

Type: int, Default value: 30'000 μs

shutdown enables or disables this module.

Type: bool, Default value: false

subSampleBy sub-sample by shifting the x and y values by this many positions. This results in a logical halving of the map size on both axes for each shift.

Type: byte, Default value: 0

4.8 Output modules

Once elaborated, the events need to be either saved or redirected somewhere, be it for further processing or to control external hardware, such as a robotic arm: output modules are the ones responsible for this operation.

Any output module should be prepared to have a NULL value instead of an event packet being handed to it, this can happen for example when no event packets of that type were available from the preceding input modules at that moment in time. Checking that there are valid events on which to work is also usually a good idea, since an event packet might have been completely invalidated by preceding processing modules, and factually be "empty" of any useful data, at the time it reaches an output module.

Event packets are sent directly 1:1 over the network or stored to file. This reduces the additional processing to be done at output time, in the common case, to nothing.

4.8.1 File

```
void caerOutputFile(uint16_t moduleID, size_t outputTypesNumber,
    ...);
```

The file output module writes event packets directly to a file. The following scheme is utilized to generate the file-name:
<directory>/<prefix>-YEAR-MONTH-DAY_HOUR:MINUTE:SECOND.aer2
The user controls the directory and prefix parts, and a suffix containing the current time is appended, so as to always supply start-of-recording temporal information automatically. The AER2 file extension signals that a new file format is being used, based on the event packets as defined in section 3.4.

It can take an arbitrary number of event packets and types, and will output them in the order they are given. The total number of output packets has to be specified for this to work correctly.

The user may specify if the full event packet (valid and invalid events) shall be written, or just the valid events. Writing only the valid events incurs a small performance penalty, since they have to be separated from the invalid ones.

The following settings are recognized:

directory the directory where data files will be saved to.

Type: string, Default value: <user home directory>

prefix the file-name prefix part.

Type: string, Default value: caer_out

shutdown enables or disables this module.

Type: bool, Default value: false

validEventsOnly only output valid events, discarding the invalid ones.

Type: bool, Default value: false

4.8.2 Unix socket client

```
void caerOutputUnixS(uint16_t moduleID, size_t outputTypesNumber
    , ...);
```

The Unix socket client output module writes event packets directly to a local Unix socket and is the preferred way to send data to another process on the same machine. Order and reliability of the communication are guaranteed. The specified local Unix socket must already have been created by another process that wants to receive data from cAER and be in listening mode (see the *unixststat* utility source-code for an example).

It can take an arbitrary number of event packets and types, and will output them in the order they are given. The total number of output packets has to be specified for this to work correctly.

The user may specify if the full event packet (valid and invalid events) shall be sent, or just the valid events. Sending only the valid events incurs a small performance penalty, since they have to be separated from the invalid ones.

The following settings are recognized:

shutdown enables or disables this module.

Type: bool, Default value: false

socketPath the path to the local Unix socket to connect with.

Type: string, Default value: /tmp/caer.sock

validEventsOnly only output valid events, discarding the invalid ones.

Type: bool, Default value: false

4.8.3 UDP network client

```
void caerOutputNetUDP(uint16_t moduleID, size_t
    outputTypesNumber, ...);
```

The UDP network client output module sends event packets directly over an UDP connection to a remote host. The UDP protocol guarantees neither ordered transmission nor transmission reliability, and should thus only be used on the local network, where the user has more control over what's happening, or in situations where data loss is potentially acceptable, and where the higher efficiency of UDP, due to decreased protocol overhead, outweighs the risks. In all other cases, the use of TCP is recommended. Given the connection-less nature of the UDP protocol, no checking is performed to verify if, on the specified remote host, there really is some process ready to listen to the data being sent. Data packets are simply sent out and then "forgotten" from the point of view of the cAER framework.

It can take an arbitrary number of event packets and types, and will output them in the order they are given. The total number of output packets has to be specified for this to work correctly.

The user may specify if the full event packet (valid and invalid events) shall be sent, or just the valid events. Sending only the valid events incurs a small performance penalty, since they have to be separated from the invalid ones.

The following settings are recognized:

ipAddress the IP address of the remote host to send packets to.

Type: string, Default value: 127.0.0.1

portNumber the destination port on the remote host to send packets to.

Type: short, Default value: 8888

shutdown enables or disables this module.

Type: bool, Default value: false

validEventsOnly only output valid events, discarding the invalid ones.

Type: bool, Default value: false

4.8.4 TCP network client

```
void caerOutputNetTCP(uint16_t moduleID, size_t
    outputTypesNumber, ...);
```

The TCP network client output module sends event packets directly over a TCP connection to a remote host. TCP is a reliable, in-order, connection-oriented protocol. As such, for the connection to succeed, a TCP server must be ready and waiting for connections on the specified remote host.

It can take an arbitrary number of event packets and types, and will output them in the order they are given. The total number of output packets has to be specified for this to work correctly.

The user may specify if the full event packet (valid and invalid events) shall be sent, or just the valid events. Sending only the valid events incurs a small performance penalty, since they have to be separated from the invalid ones.

The following settings are recognized:

ipAddress the IP address of the remote host to connect to.

Type: string, Default value: 127.0.0.1

portNumber the destination port on the remote host to connect to.

Type: short, Default value: 8888

shutdown enables or disables this module.

Type: bool, Default value: false

validEventsOnly only output valid events, discarding the invalid ones.

Type: bool, Default value: false

4.8.5 TCP network server

```
void caerOutputNetTCPServer(uint16_t moduleID, size_t
    outputTypesNumber, ...);
```

The TCP network server output module starts a TCP server on the local host and waits for connections from other systems. Once a remote system has completed the connection procedure, the output module sends event packets directly over the established TCP connection to the remote system. The simple action of connecting to this server by a client and successfully completing the connection procedure, taking into account the maximum number of allowed clients, puts it into the list of clients that data is sent to, starting the data transfer right away. Closing the connection on the client's part takes them off this list, putting a stop to the data transfer. A change to the maximum number of connected clients also has repercussions on already connected ones: if the new value is smaller than the number of currently connected clients, some of them will be disconnected to adhere to the new quota.

It can take an arbitrary number of event packets and types, and will output them in the order they are given. The total number of output packets has to be specified for this to work correctly.

The user may specify if the full event packet (valid and invalid events) shall be sent, or just the valid events. Sending only the valid events incurs a small performance penalty, since they have to be separated from the invalid ones.

The following settings are recognized:

backlogSize maximum number of pending connections kept in the connection queue.

Type: short, Default value: 5

concurrentConnections maximum number of allowed simultaneously connected clients.

Type: short, Default value: 5

ipAddress the local IP address on which to have the server listen for incoming connections.

Type: string, Default value: 127.0.0.1

portNumber the local port on which to have the server listen for incoming connections.

Type: short, Default value: 7777

shutdown enables or disables this module.

Type: bool, Default value: false

validEventsOnly only output valid events, discarding the invalid ones.

Type: bool, Default value: false

Chapter 5

Results

The main result of this thesis is a working implementation of the software architecture described in chapter 3 and further explained in chapter 4. The whole project encompasses about 9'600 lines of code, of which circa 3'600 are part of external libraries, resulting in ∼6'000 lines of code written specifically for this project. As detailed over the previous chapters, all requirements have been carefully considered and met.

The software was tested on a number of embedded systems, such as the Raspberry Pi and the PandaBoard, as well as on a standard desktop system from Intel, and exhibited no problems during installation and operation, proving compatible with a wide range of architectures: ARMv6, ARMv7 and x86/x86-64.

It has also been shown that the implementation of event-based algorithms is easily possible with the software presented here, in fact, the Background Activity Filter presented in section 4.7.1 was ported from the jAER framework in less than thirty minutes, thanks to a very similar concept of how to do event-based computation with event packets.

Basic performance tests have been done on two systems, the first is an example of an embedded system: a Raspberry Pi Model B, featuring a 700 MHz ARMv6 1-core processor with 512 MB of RAM, and the second is a normal desktop computer, with an Intel Core i7-3770K x86-64 quad-core CPU running at up to 3.5 GHz, with 16 GB of RAM available. These two systems sit at the two extremes of consumer-grade hardware, one having very limited resources but consuming just a couple of watts, the other having plenty of computational resources, but consuming at least an order of magnitude more electrical power.

All binaries have been compiled with GCC 4.8, resulting in binary sizes of 102 KB and 112 KB respectively for the Raspberry Pi and the desktop system. The Raspberry Pi was running the Raspbian "jessie" Linux distribution, while the desktop system was running the Fedora 20 Linux distribution (64-bit edition).

Processor usage was estimated by looking at the %CPU column of the *ps* [11] utility and memory usage was calculated by the *ps_mem* [2] Python utility. Profiler suites, such as Valgrind or Google Perftools, are not supported on the Raspberry Pi and were thus not considered as an alternative to get this information. Both evaluations have been run with one DVS128 camera connected to the system, with one active Mainloop getting polarity events from it and sending them out via UDP to the network.

Raspberry Pi	static wall	heavy activity
Processor usage:	∼15%	∼25%
Memory usage:	∼690 KB	∼2'600 KB

Table 5.1: Raspberry Pi performance evaluation

Desktop system	static wall	heavy activity
Processor usage:	∼3%	∼3.5%
Memory usage:	∼2'600 KB	∼2'700 KB

Table 5.2: Desktop system performance evaluation

For one test ("static wall") the DVS128 was pointed at a white, non-changing wall, for the other test ("heavy activity"), a human hand was repeatedly waved in front of it at a constant rate, as would be the case with gesture recognition scenarios.

Both memory and CPU consumption are satisfactorily small on the embedded system, as shown in table 5.1.

The more powerful desktop computer doesn't manage to record a big difference in its processor usage values, according to table 5.2, probably due to it being so much faster at elaborating data, as well as being able to run several threads concurrently on its multiple cores. The surprisingly big initial memory usage could be explained by it loading a number of big libraries at program startup, that might have been linked into the executable. Those last assumptions regarding the desktop system's values will have to be validated with further tests in the future.

Chapter 6

Future work

There are several areas of interest that could be developed further now that a basic software framework is present.

There are several hardware devices, both neuromorphic and consumer-oriented, that it would be interesting to integrate with the system.

Still more attention could be paid to the networking side of things, implementing for example authentication and encryption of the communication and configuration channels, which could be greatly useful in industrial usage scenarios or when working over untrusted connections, such as the Internet at large, or wireless connections.

There are also a multitude of algorithms, written for jAER or other frameworks, that might benefit from being ported to cAER, to reduce their resource usage or integrate them with previously unavailable APIs and devices, enabling them to run in new environments or with new features.

Chapter 7

Conclusions

New sensors always offer exciting and novel opportunities to interact with the world, but they must also be supported by infrastructure that makes their output valuable and useful, making it possible to elaborate this new data and react to it. Existing software solutions in this field have shown themselves to not be particularly suitable to run on low-power embedded systems, so a new software architecture and implementation had to be written with these constraints in mind.

The cAER software presented here enables the implementation of event-based processing algorithms for these novel sensors, even on very resource limited systems. It introduces a new encoding and transmission format for events, and a supporting framework to process them. Special attention has been paid to configuration and control of the software, as well as interoperability with other frameworks via a multitude of output options. Its modular approach enables users to select only the options they truly need for their applications, while enabling developers to easily add new algorithms and solutions.

Low-power sensors with high temporal resolution and sparse output are highly promising, and can now finally be paired with software that takes full advantage of their characteristics, completing another critical step on the way to revolutionary real-world applications in fields such as autonomous robotics, where battery capacity is limited and fast reactions are key, or surveillance, where removing redundant data could save significant amounts of storage.

Bibliography

[1] D. Bonino, F. Corno, *spChains: A Declarative Framework for Data Stream Processing in Pervasive Applications*, 2012, The 3rd International Conference on Ambient Systems, Networks and Technologies

[2] P. Brady, *ps_mem utility*, 16 October 2013. Available: https://github.com/pixelb/ps_mem/

[3] A. Censi, J. Strubel, C. Brändli, T. Delbruck, D. Scaramuzza, *Low-latency localization by Active LED Markers tracking using a Dynamic Vision Sensor*, 2013

[4] J. Conradt, M. Cook, R. Berner, P. Lichtsteiner, RJ. Douglas, T. Delbruck, *A Pencil Balancing Robot using a Pair of AER Dynamic Vision Sensors*, 2009, Proc. of the International Conference on Circuits and Systems (ISCAS)

[5] P. Dekkers, *Complex Event Processing*, October 2007

[6] T. Delbruck, P. Lichtsteiner, *Fast sensory motor control based on event-based hybrid neuromorphic procedural system*, 2007, IEEE Intl. Symp. Circuits Syst. 2007:845-848

[7] T. Delbruck, *Frame-free dynamic digital vision*, 2008, Proceedings of Intl. Symp. on Secure-Life Electronics, Advanced Electronics for Quality Life and Society

[8] T. Delbruck, *jAER open-source project*, 2007. Available: http://jaer.sf.net/

[9] Intel Corporation, *Intel 64 and IA-32 Architectures Optimization Reference Manual*, July 2013

[10] P. Lichtsteiner, C. Posch, T. Delbruck, *An 128x128 120dB 15us-latency temporal contrast vision sensor*, February 2008, IEEE J. Solid State Circuits, 43(2), 566-576

[11] Linux manual pages section 1: User Commands, *man ps*, December 2011

[12] S-C. Liu, A. van Schaik, B. A. Minch, T. Delbruck, *Event-based 64-channel binaural silicon cochlea with Q enhancement mechanisms*, 2010, Proceedings of the IEEE International Conference on Circuits and Systems, pp. 2027-2030

[13] S-C. Liu, T. Delbruck, *Neuromorphic sensory systems*, 2010