FACULTY OF APPLIED SCIENCES
UNIVERSITY
OF WEST BOHEMIA

DEPARTMENT OF
COMPUTER SCIENCE
AND ENGINEERING

Master's Thesis

Real-Time Concept for SmartCGMS

Petr Kocián

PILSEN, CZECH REPUBLIC

2024

FACULTY OF APPLIED SCIENCES
UNIVERSITY
OF WEST BOHEMIA

DEPARTMENT OF
COMPUTER SCIENCE
AND ENGINEERING

Master’s Thesis

Real-Time Concept for SmartCGMS

Petr Kocián

Thesis advisor
Doc. Ing. Tomáš Koutný, Ph.D.

PILSEN, CZECH REPUBLIC **2024**

Citation in the bibliography/reference list:
KOCIÁN, Petr. *Real-Time Concept for SmartCGMS*. Pilsen, Czech Republic, 2024. Master's Thesis. University of West Bohemia, Faculty of Applied Sciences, Department of Computer Science and Engineering. Thesis advisor Doc. Ing. Tomáš Koutný, Ph.D.

ZÁPADOČESKÁ UNIVERZITA V PLZNI
Fakulta aplikovaných věd
Akademický rok: 2023/2024

ZADÁNÍ DIPLOMOVÉ PRÁCE
(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Petr KOCIÁN**
Osobní číslo:	**A21N0034P**
Studijní program:	**N3902 Inženýrská informatika**
Studijní obor:	**Počítačové systémy a sítě**
Téma práce:	**Real-Time koncept SmartCGMS**
Zadávající katedra:	**Katedra informatiky a výpočetní techniky**

Zásady pro vypracování

1. Seznamte se s frameworkem SmartCGMS.
2. Seznamte se s operačním systémem FreeRTOS.
3. Seznamte se se zařízeními RPI Zero a vývojovým kitem ESP32.
4. Dle stávajícího rozhraní SmartCGMS navrhněte jeho koncept pro FreeRTOS, který poběží na daných zařízeních.
5. Implementujte navržený koncept, který bude přenositelný na úrovni zdrojového kódu mezi instrukčními sadami AMD64/x86-64, Armv6 a ESP32.
6. Dále implementujte 4 jednoduché SmartCGMS filtry – (1) čtení, (2) transformace a (3) vizualizace dat, a (4) watchdog.
7. Otestujte funkčnost celého řešení a změřte jeho provozní charakteristiky.
8. Kriticky zhodnoťte dosažené výsledky.

Rozsah diplomové práce:	**doporuč. 50 s. původního textu**
Rozsah grafických prací:	**dle potřeby**
Forma zpracování diplomové práce:	**tištěná/elektronická**
Jazyk zpracování:	**Angličtina**

Seznam doporučené literatury:

dodá vedoucí diplomové práce

Vedoucí diplomové práce:	**Doc. Ing. Tomáš Koutný, Ph.D.** Katedra informatiky a výpočetní techniky
Datum zadání diplomové práce:	**8. září 2023**
Termín odevzdání diplomové práce:	**16. května 2024**

L.S.

Doc. Ing. Miloš Železný, Ph.D.	**Doc. Ing. Přemysl Brada, MSc., Ph.D.**
děkan	vedoucí katedry

V Plzni dne 11. října 2023

Declaration

In Pilsen, on 16 May 2024

. . . .
Petr Kocián

Abstract

SmartCGMS is a framework for continuous glucose monitoring systems. It has been used extensively for testing and optimizing in silico models, but it needs to improve its support for execution on low-power devices.

This paper proposes a real-time SmartCGMS concept based on FreeRTOS to examine options for compiling and executing SmartCGMS on low-power devices. The concept has been deployed as a native application on an ESP32 and a Raspberry Pi Zero W. Additionally, it was compiled into WebAssembly as it is an emerging approach to deploying applications to low-power devices in the Internet of Things. Compiling to WebAssembly also allows configuring and experimenting with SmartCGMS directly in the web browser. The native application and WebAssembly module were compared on GNU/Linux to decide which approach to adopt in the SmartCGMS framework.

The results confirmed the expectations of the computational and memory overhead of the WebAssembly approach. Furthermore, the size of the compiled WebAssembly SmartCGMS concept revealed that the ESP32 does not offer sufficient DRAM for the WebAssembly approach with the used setup. Regardless of these limitations, the WebAssembly approach should not be abandoned as it provides a way to execute SmartCGMS in the web browser that could help it to widespread adoption.

Abstrakt

SmartCGMS je framework zaměřený na systémy kontinuálního monitorování glukózy. SmartCGMS je již široce používáno na testování a optimalizaci in silico modelů, ovšem je potřeba rozšířit jeho podporu na nízko-příkonových zařízeních.

V této práci je navrhnut real-time koncept SmartCGMS běžící na FreeRTOS pro vyhodnocení a prozkoumání možností kompilace a spouštění SmartCGMS na nízko-příkonových zařízeních. Navrhnutý koncept byl spušten jako nativní aplikace na ESP32 a Raspberry Pi Zero W. Kromě toho byl koncept zkompilován do WebAssembly, jakož nově se rozvíjející způsob spouštění aplikací na zařízení v Internet of Things. Kompilace do WebAssembly dále umožňuje konfiguraci a experimentování se SmartCGMS ve webovém prohlížeči. Nativní aplikace a WebAssembly modul byly porovnány na GNU/Linux za účelem výběru, který přístup dále použít pro vývoj SmartCGMS.

Výsledky potvrdily očekávání ohledně výpočetní a paměťové náročnosti při použití WebAssembly. Dále bylo odhaleno, že ESP32 nenabízí dostatečnou DRAM paměť pro spuštění SmartCGMS WebAssembly modulu. Nehledě na tato omezení umožňuje WebAssembly spouštění ve webovém prohlížeči, což může pomoci rozšířit povědomí o SmartCGMS, a proto by WebAssembly mělo zůstat podporovanou platformou.

Keywords

Continuous glucose monitoring • SmartCGMS • Low-power devices • FreeRTOS • WebAssembly

Contents

1 Introduction **3**

2 State of the Art **4**

2.1 Diabetes Mellitus 4

2.1.1 Insulin Therapy 5

2.2 Continuous Glucose Monitoring 5

2.2.1 OpenAPS 6

2.2.2 AndroidAPS 7

2.2.3 Loop 7

2.2.4 Medtronic MiniMed 670G 7

2.2.5 SmartCGMS 7

2.3 Current Efforts to Execute SmartCGMS on Low-Power Devices . 8

2.3.1 WebAssembly 9

2.4 Building Blocks of Proposed SmartCGMS Concept 9

2.4.1 SmartCGMS 10

2.4.2 FreeRTOS 13

2.4.3 WebAssembly 14

3 Proposed System **18**

3.1 System Design 19

3.1.1 Static Linking 19

3.1.2 Filter Preprocessor 19

3.1.3 Preserving Modularity 20

3.1.4 WebAssembly 21

3.2 Target Architectures 21

3.2.1 Low-Power Devices 21

3.2.2 High Performance Computing 22

4 Implementation **23**

4.1 SmartCGMS Codebase 23

4.1.1 Functions Incompatible with Embedded Environments . . 23

4.1.2 Resolving Symbols 24
4.1.3 Managing Entities 24
4.2 Filter Preprocessor 27
4.3 SmartCGMS Concept API 28
4.4 Filters 28
4.4.1 Data Reading Filter 29
4.4.2 Data Transformation Filter 30
4.4.3 Data Visualization Filter 30
4.4.4 Watchdog Filter 31
4.5 Supported Platforms 32
4.5.1 Raspberry Pi Zero W 33
4.5.2 ESP32 33
4.5.3 WebAssembly (WASM) Standalone Runtime 34
4.5.4 WASM Targeted at The Web 35

5 Evaluation **39**
5.1 Correctnes of Execution 40
5.1.1 Experimental Setup 40
5.1.2 Results and Discussion 41
5.2 Memory Usage 42
5.2.1 Experimental Setup 42
5.2.2 Results and Discussion 43
5.3 Execution Time 47
5.3.1 Experimental Setup 47
5.3.2 Results and Discussion 48

6 Conclusion and Future Work **50**

Bibliography

User Manual

List of Figures

List of Tables

List of Listings

Acronyms

Introduction 1

Diabetes mellitus is a globally widespread metabolic disorder characterized by elevated blood glucose levels, which, if left unmanaged, can lead to severe complications. Measuring blood glucose levels is essential for diabetic patients. A glucometer can be used to obtain approximate values from a drop of blood. A more convenient way is using a Continuous Glucose Monitoring (CGM) system. CGM is a method of tracking blood glucose levels using a blood glucose sensor that measures the levels, which are then transmitted to a receiver that can display the data or process them further.

SmartCGMS is a software framework for signal processing focused on CGM. SmartCGMS uses a fall-through architecture to represent the CGM system. The signal passes through in one direction. Each part of the system (e.g., a transmitter or an insulin pump) is represented by a filter. Individual components, i.e., SmartCGMS filters, can be exchanged without adversely affecting the rest of the system (the idea of High Level Architecture (HLA)). This enables a SmartCGMS application to be developed and optimized using simulated devices and deployed to a real device without modifications.

The project aims to propose, implement, and evaluate an approach to address this challenge. It involves modifying SmartCGMS to create a concept that could be run on devices using FreeRTOS. The main requirement is to retain compatibility with the already supported platforms. Additionally, four filters should be implemented to demonstrate the concept's functionality: a filter to read data, transform the data, visualize the data, and a watchdog filter. The project's findings can be used as a base for further SmartCGMS development.

State of the Art 2

As the prevalence of diabetes mellitus continues to rise, advancements in technology are being directed toward improving the treatment of the disease. Continuous Glucose Monitoring (CGM) systems allow diabetic patients to continuously monitor their blood glucose levels. In combination with an insulin pump, a closed-loop system can be created. Such a system can autonomously control patients' blood glucose levels. Today's closed-loop systems are not fully autonomous yet and require patients to manually inject correcting insulin doses. The systems are referred to as hybrid closed-loop. There are multiple hybrid closed-loop systems available for diabetic patients, including certified (e.g.,Food and Drug Administration (FDA) approved) [1] and Do-It-Yourself (DIY) devices [2][3][4]. Before a device can be made commercially available, it must obtain approval from a governing body (e.g., FDA). During the approval, clinical studies are conducted to assess the security and safety of the device. Simulation can be used in the clinical studies [5]. SmartCGMS provides a framework to create CGM system or hybrid closed-loop software that can be tested in a simulation and without any modifications deployed to a real device [6]. However, its support for low-power devices is limited. This chapter summarizes the current state of popular available hybrid closed-loop systems, describes a recently-emerging way of bringing WebAssembly (WASM) to Internet of Things (IoT) as a way to create portable software, and details technologies used for the realization of this project.

2.1 Diabetes Mellitus

Diabetes mellitus is a chronic metabolic disorder characterized by high blood glucose levels (hyperglycemia) resulting from insufficient insulin production or the body's inability to use insulin effectively. Failing to keep blood glucose levels in the optimal range can result in various health complications for the patients (e.g., kidney damage, nerve damage, and cardiovascular diseases). The blood glucose levels can be regulated by injecting insulin into the patient's bloodstream. However, an excessive insulin dose can result in a too-low blood glucose level (hypoglycemia). To avoid hyperglycemia and hypoglycemia, diabetic patients must closely monitor and control their blood glucose levels.

2.1.1 Insulin Therapy

Some diabetic patients require insulin therapy. The patients inject synthetic insulin into the fat layer under their skin, where it is absorbed into the bloodstream. Two types of insulin are used: long-acting and short-acting. The long-acting insulin provides a steady supply of insulin. It helps regulate blood glucose levels throughout the night and between meals. The short-acting insulin is injected before eating a meal to help process the consumed carbohydrates or as a correction when the blood glucose level is too high. It is referred to as a bolus dose.

Insulin pumps can be used for insulin therapy as an alternative to insulin injections. An insulin pump continuously injects insulin under the patient's skin. It only uses short-acting insulin to regulate blood glucose levels. The short-acting insulin is injected continuously in small doses to mimic the body's background level of insulin production. This constant stream of short-acting insulin is referred to as basal rate. Before a meal, the user can inject a higher bolus insulin dose to keep blood glucose levels in the desired range.

2.2 Continuous Glucose Monitoring

To determine the required insulin dose, diabetic patients may measure their blood glucose levels using a blood glucose meter. A blood glucose meter (glucometer) is a device that measures blood glucose levels from a drop of blood. The blood is usually extracted from the patient's fingertip. Even diabetic patients without insulin therapy are required to measure their blood glucose levels to help manage their medications and evaluate the results of ongoing therapy (e.g., diet and exercise).

CGM system can be used instead of a glucometer. A CGM system is a device that estimates a patient's blood glucose level by using a sensor located under their skin. It provides a real-time estimated value of the blood glucose levels. It consists of three parts: a blood glucose sensor, a transmitter, and a receiver. The sensor measures an electric current produced by a glucose-triggered electrochemical reaction. The measured current is denoised and transformed into a blood glucose level estimate. Then, these estimates are transmitted to the receiver. The receiver can be a smartphone, an insulin pump, or a separate device that displays or further processes the measured data.

An insulin pump can be connected to a CGM system to create a closed-loop or a hybrid closed-loop system. Such a system automatically adjusts the patient's insulin dosage based on the measurements from the CGM. A closed-loop system is fully autonomous. A hybrid closed-loop system requires users to input their meals into the system.

2.2.1 OpenAPS

OpenAPS is an example of a hybrid closed-loop system. It is a DIY solution for diabetic patients. It is distributed as an open-source software with thorough documentation on how to set it up. The OpenAPS solution uses a certified CGM system to collect the data but in a non-certified manner. It processes the data and then adjusts the injected insulin dosage of a remotely connected certified insulin pump. It works only with specific CGM systems and insulin pumps. For compatibility reasons, it sometimes uses legacy technology. As a result, the DIY solutions like OpenAPS are not certified by any authority.

OpenAPS uses an algorithm called `oref0`. It is a heuristic-based algorithm. OpenAPS argues that an algorithm with decision-making that can be understood by the user is beneficial. For safety reasons, the *oref0* algorithm only adjusts the basal rate of the insulin pump. The basal insulin rate's maximum is low enough not to overdose the patient. OpenAPS offers an improved version of the original `oref0` algorithm referred to as `oref1`. A study conducted by Petruzelkova et al. [7] concluded that the OpenAPS `oref0` algorithm is a safe alternative to the FDA approved algorithm developed by Medtronic. Further studies showed that the OpenAPS solution can improve the patient's blood glucose levels [8]. However, Künzler et al. [9] showed the limitations of DIY hybrid closed-loop systems, in their study users of OpenAPS encountered malfunctions that rendered the system unusable for 23% of the total measured time.

It should be strongly noted that Petruzelkova et al. did not investigate the hardware and software design of the OpenAPS solution, and they did not evaluate any fault-tolerant or fail-safe aspect of the system. By examining its source code, particularly the recent git commit `da7015c` [10], it can be seen that the software does not meet fail-safe nor fault-tolerant requirements. This is in accordance with study [9]. Therefore, we conclude that improvement of the patient medical condition could have been achieved rather by increased self-monitoring of the blood glucose level and more frequent patient interventions (bolus, meal, physical activity, stress relief, etc.), thus being the `oref`-benefit a self-bias affected. This is in accordance, e.g., with study [11] that states "OpenAPS shows similar results to more rigorously developed and tested AP technology in a highly selective, motivated and technology-adept population of individuals with type 1 diabetes" [sic!]. Another study [12], is concerned about the cybersecurity of DIY solutions, explicitly naming OpenAPS, whereas the cybersecurity comes from the hardware and software design of the system.

When building an OpenAPS system, a CGM system, an insulin pump, and additional hardware that runs OpenAPS are required. OpenAPS recommends, for example, the Raspberry Pi Zero WH. If using the Raspberry Pi, the OpenAPS application runs on the Raspberry Pi OS Lite operating system. This allows the application to be updated without rebuilding the whole system. [2]

2.2.2 AndroidAPS

AndroidAPS (AAPS) is another DIY hybrid closed-loop system. It is an Android application based on the `oref0` and `oref1` algorithms from OpenAPS. The AAPS system uses an AAPS application to control the insulin rate and another application to communicate with the CGM system. Some pumps might require an additional device to be able to communicate with the phone - a RileyLink compatible device. RileyLink is a protocol defined for communication with insulin pumps. The data is collected using Bluetooth. AAPS is compatible with more insulin pumps and CGM systems than the OpenAPS solution. [3]

2.2.3 Loop

Loop is also a DIY hybrid closed-loop system. It is an application for iPhones. It uses its own algorithm to determine the optimal insulin bolus and basal rate. Unlike OpenAPS's `oref0` algorithm, which only computes the basal rate. Loop does not issue insulin bolus commands by default. It only suggests a recommended dose. The Loop system also requires a RileyLink compatible device to communicate with certain insulin pumps. [4]

2.2.4 Medtronic MiniMed 670G

Medtronic MiniMed 670G is the first FDA approved hybrid closed-loop system, but more hybrid closed-loop systems have been approved since its approval in 2016. The MiniMed 670G comprises a CGM system and an insulin pump. The insulin basal rates are computed using a SmartGuard algorithm developed by Medtronic. The MiniMed system offers two modes: auto mode and manual mode. In auto mode, the basal insulin dosage is automatically adjusted based on data from the CGM system. In manual mode, users have control over setting the basal rate themselves. Numerous studies, such as [13] and [14], have shown that the device in auto mode - acting as a hybrid closed-loop - has a positive effect on a patient's blood glucose levels. [15][1]

2.2.5 SmartCGMS

SmartCGMS is an open-source software architecture framework for continuous glucose monitoring. SmartCGMS is based on the idea of co-simulation and High Level

Architecture (HLA) standard. HLA is a standard that allows the combination of multiple real and simulated devices in a distributed simulation [16]. In the SmartCGMS framework, such a device can be, for example, a glucose sensor, insulin pump, or a predictor of glucose level. The SmartCGMS framework defines filters that represent individual parts of the system. The concept of co-simulation allows developers to perform optimization and development on simulated devices and then switch the simulated device to a real one when the program code is ready. An example of how SmartCGMS can be used has been shown in [6]. In this study, SmartCGMS has been used to evaluate the `oref` algorithm using an FDA approved virtual diabetic patients simulator DMMS.R [5]. By taking advantage of SmartCGMS co-simulation support, the original `oref` code could have been directly used in the experimental setup [6]. [17]

2.3 Current Efforts to Execute SmartCGMS on Low-Power Devices

SmartCGMS officially supports these platforms:

- Win64
- MacOS 64
- Debian 64
- Android arm64-v8a
- Android armeabi-v7a

In a study by Otta [18] an approach was proposed to execute SmartCGMS on the ESP32 system on a chip. In the proposal, Otta suggests using Espressif IoT Development Framework (ESP-IDF) as the underlying software system. The ESP-IDF is based on FreeRTOS and provides multiple software components (e.g., peripheral drivers and network stack). Using a Real-time operating system (RTOS) ensures the system's critical tasks are executed in time. In the proposed system, several tasks run on top of FreeRTOS to manage applications, remote communication, and updates.

The proposal defines features - applications that can be remotely installed (e.g., SmartCGMS). Each feature is a WASM module run by a lightweight virtual machine. There are already multiple supported WASM runtimes for ESP-IDF [19][20]. The features can then be updated during runtime.

2.3.1 WebAssembly

WASM is a binary format defined by The World Wide Web Consortium [21]. The definition includes a set of virtual instructions. The instructions are executed by a virtual stack machine. The code is distributed in WASM binary files called WASM modules. Initially, WASM was meant for web browsers, but the virtual instructions set can be implemented in a standalone runtime. Some of these standalone runtimes also support embedded devices (e.g., the ESP32) [19] [20].

WASM was designed to be safe, fast, and portable [22]. The binary files are called modules. The modules define functions, globals, tables, and memories. Definitions can be both imported and exported. All imports must be resolved when instantiating the module. A module is the static representation of the program. The dynamic representation is called an instance. Instances hold the runtime state of the WASM module, allowing execution and interaction with the host environment. Functions represent an executable block of code. Exported functions can be called from the host environment directly, or they can be called by other WASM functions, even recursively. Globals are global variables defined in the module. Tables are arrays of references to objects. The memory of a WASM module is a large array of bytes (referred to as linear memory). The unit of size for WASM module memory is 64 KB. Each module can only have one memory, but it is possible to grow the module's memory in increments of 64 KB. The memory is separate from the code space and execution stack. This makes executing WASM modules safe, as they cannot corrupt their execution environment. The module's memory can, however, be accessed by the execution environment. This makes it possible to make a shared memory region between the module and the execution environment. [22] [21]

Several papers have been published on the usability of WASM in low-power devices, mainly with a focus on the IoT, such as [23] and [24]. Its low overhead, safety, and portability make it a good candidate for IoT applications [25]. The main appeal of WASM for SmartCGMS is the ability to update the application dynamically as suggested by [18] and [24] and evaluated by [26]. Compiling SmartCGMS for low-power devices as a native application using an RTOS (e.g., FreeRTOS) would require flashing the whole firmware when introducing new functionality or fixing issues.

2.4 Building Blocks of Proposed SmartCGMS Concept

This paper proposes and evaluates another approach for executing SmartCGMS on low-power devices. The project's requirements are to define and implement a real-time SmartCGMS concept based on FreeRTOS. The concept should maintain compatibility with the x86-64 Instruction Set Architecture (ISA). The concept will also be compiled into the WASM format to test the feasibility of Otta's suggested solution [18]. The following sections describe the technologies used to implement the proposed system. First, the SmartCGMS framework and its relevant parts are described. The next section describes FreeRTOS, and the last section is dedicated to WASM tools.

2.4.1 SmartCGMS

The general description of SmartCGMS and the description of what it can be used for is in section 2.2.5. This section details the project-relevant SmartCGMS internals.

2.4.1.1 SmartCGMS Architecture

SmartCGMS uses a fall-through architecture for signal processing. The steps of the processing are represented by filters, which together form a filter chain. A signal is processed as a series of messages called device events that linearly pass through the filter chain. Typically, at the beginning of the filter chain, there is a filter representing a source of device events. This filter can either simulate a device (e.g., by generating events from a predefined dataset) or read data from a real device. The device event is then sent to the next filter. Each filter can decide to destroy, modify, or pass the event through. It can also generate a new event. At the end of the filter chain, an output filter can represent, for example, an insulin pump or a filter visualizing the data from the events. An example of a filter chain can be seen in figure 2.1. [27]

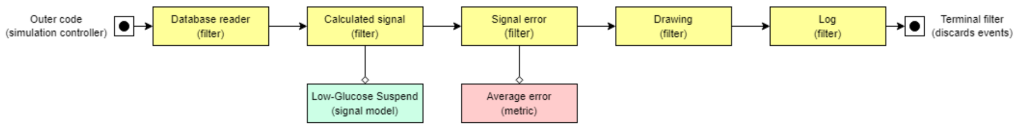


Figure 2.1: Example of a SmartCGMS filter chain [27].

SmartCGMS also defines other entities (e.g., models and signals). Each entity is compiled into a dynamic library. A SmartCGMS application is a set of dynamic libraries. The main SmartCGMS components are SmartCGMS core and SmartCGMS common. The common library defines interfaces for SmartCGMS entities based on the Component Object Model, and it provides base implementations for SmartCGMS classes, functions, and utilities. The SmartCGMS core library manages the resources of a SmartCGMS application (e.g., builds the filter chain and creates events). A typical SmartCGMS application includes the core and common SmartCGMS dynamic libraries and then a dynamic library for each entity that is used in the application. The SmartCGMS project currently offers precompiled binaries for Windows, MacOS, Debian, and Android operating systems. [27]

2.4.1.2 **Device Event**

The device event represents a message in the SmartCGMS filter chain. It is a structure that holds a device globally unique identifier (GUID) that identifies each event originator, signal GUID that identifies the signal carried by the event, device and logical time, and segment ID (specifying a time segment the event belongs to). The data carried by the event is stored in a union containing a level (signal level value), parameters (array of parameters), and info (a string). The type of event is distinguished by event code. There are fifteen different event codes. [27]

There are four event codes relevant to this project:

- Shut_Down - signals to a filter that it should terminate and release its resources
- Level - holds a signal value
- Information - holds an information string
- Error - reports an error, holds the error string

2.4.1.3 **SmartCGMS Filter**

Each SmartCGMS filter is managed as a dynamic library. It is loaded by the SmartCGMS framework and managed by calling exported functions: `do_create_filter` and `do_get_filter_descriptors`. A filter is described by its descriptor (`scgms::TFilter_Descriptor`). It is a structure that holds the filter's GUID, name, description, parameter information, and flags. Each descriptor member provides necessary information about the filter to the SmartCGMS framework or the developer. The filter must export the `do_get_filter_descriptors` function, which returns its descriptors in a continuous array. [27]

To create the filter object, SmartCGMS uses the other exported function `do_create_filter`. This function takes the filter GUID as a parameter. If the GUID matches the GUID of the filter, it creates the filter and returns a reference to it. [27]

Once a filter is created, it has to be configured before it can process events. The filter life cycle can be seen in figure 2.3. Each SmartCGMS filter must implement an interface defined by the SmartCGMS framework called `scgms::IFilter`. This interface is an abstract class with two pure virtual functions: `Configure` and `Execute`. A pure virtual function in C++ is a function that is not defined in the base class. This forces its classes to implement the function. [27]

The `Configure` function provides the filter parameters (loaded from the filter chain configuration) to the filter. It configures the filter and puts it in the operational state. When the filter is in the operational state, it can process events. The events are processed by calling the `Execute` function. When the `Execute` function is called, an event is passed to the filter. The filter can read and modify the event values. After the filter processes the event, it can then either pass it to the next filter or deallocate the event memory. The event processing is synchronous. Each `Execute` recursively calls the next `Execute` functions until the event is deallocated by a filter or the event reaches the end of the filter chain (where it is deallocated by a terminal filter). Then the `Execute` function call returns. The filter chain execution is protected by a recursive mutex. In the case of multiple threads calling `Execute` simultaneously, SmartCGMS only allows a single call to the `Execute` function and blocks the other threads. [27]

The filter is terminated when it receives a `Shut_Down` event. [27]

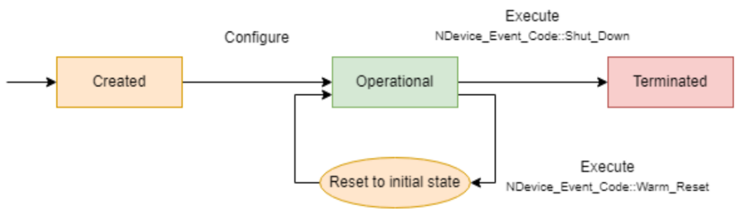


Figure 2.2: Life cycle of a SmartCGMS filter [27].

2.4.1.4 SmartCGMS Discrete Model

The SmartCGMS discrete model is an entity that can represent, for example, a physiological model of a diabetic patient (e.g., The Bergman Minimal Model [28]). The discrete model is implemented as a derived class from the `scgms::IFilter` interface. It implements two additional functions: `Step` and `Initialize`. The discrete model has a state. It has to be initialized prior to operation. The discrete model then performs discrete steps to advance its state. The discrete model is managed by a signal generator filter. Its operation can be seen in figure 2.2. The model parameters are specified under the signal generator filter parameters in the configuration file.

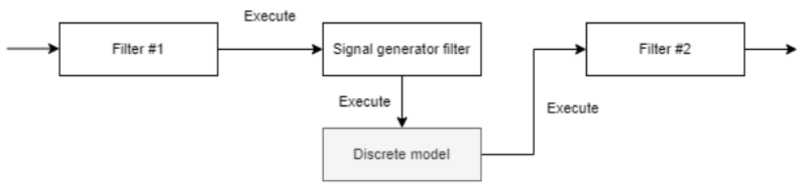


Figure 2.3: Operation of the SmartCGMS signal generation filter and a discrete model [27].

2.4.1.5 SmartCGMS Configuration

The SmartCGMS configuration is stored in an INI file. It defines the order of the filters in the filter chain and contains their parameters. The configuration can be generated by the SmartCGMS GUI application or by using a text editor. It is designed to be human-readable. A single filter can be configured, for example, as follows:

```
[Filter_001_{8FAB525C−5E86−AB81−12CB−D95B1588530A}]
Buffer_Size = 100
Log_File = Log.txt
```

The three-digit number after 'Filter_' denotes the filter's position in the filter chain, the curly brackets hold its GUID, and then follow the filter's parameters (specified in the filter descriptor). [27]

2.4.1.6 SmartCGMS Portability

SmartCGMS has been developed with portability in mind. The source code is written in C++ (C++17 standard) to make SmartCGMS portable, minimize its memory footprint, and maximize its computational performance [29]. As mentioned in section 2.4.1.1, SmartCGMS is available on Windows, MacOS, Debian, and Android. All of these platforms have a file system and support dynamic libraries. These features might not be available on embedded devices running a Real-time operating system (RTOS), and thus, the support for such devices would require modifications to the SmartCGMS code base. SmartCGMS has been compiled for Raspberry Pi, but it is not officially supported and documented.

2.4.2 FreeRTOS

To achieve predictability and reliability, a RTOS can be used to manage complex programs in a real-time system. Safety-critical real-time systems (e.g., medical embedded systems) require the highest level of predictability and reliability. This can be achieved by using a certified RTOS [30]. FreeRTOS is a free, widespread RTOS that offers a pre-certified version (SAFERTOS) with official instructions on how to port an existing application from FreeRTOS to SAFERTOS [31][32]. However, SAFERTOS does not use the same codebase as FreeRTOS. It is only based on its functional model [31]. If certified software is not required for a solution, FreeRTOS is very well supported by its community and also offers official technical support under the project OpenRTOS. The codebase of FreeRTOS and OpenRTOS is shared. Therefore, applications written for FreeRTOS can be ported to OpenRTOS without any modifications [33].

As many other RTOSs, FreeRTOS has low processing and memory overhead (typically around 6 to 12 KB). The core functionality of the kernel is located in only

3 files, which makes it simple to use and understand. Its documentation is extensive, and it offers official support for many architectures [32]. Additionally, many community ports are available online. FreeRTOS is also supported as a component in Espressif IoT Development Framework (ESP-IDF) [34].

FreeRTOS was selected as a base operating system for the approach of bringing SmartCGMS to a smartwatch proposed by [18]. It is also specified as a required RTOS for this project.

2.4.2.1 Tasks

Tasks are the threads of a RTOS. A task in FreeRTOS is a function that executes as an endless loop or must delete itself after finishing its execution. FreeRTOS scheduler manages the execution of tasks and guarantees that only one task runs at a given time. The tasks are executed according to their priority, which is set during the task creation. Other parameters can also be configured during the creation. These are the task's code, its name, stack depth, parameters, priority, and a task handle that refers to the created task. FreeRTOS additionally offers support for creating tasks statically. In this case, the task is not allocated from the FreeRTOS heap, but the memory needs to be provided at the task creation. This solution is slightly more complex. There are two extra parameters: the task's stack buffer (used for the task's stack) and the task buffer (used for storing runtime information about the task). However, the task's memory can be allocated at compile time. [32]

2.4.2.2 FreeRTOS-Plus-POSIX

FreeRTOS-Plus-POSIX is a FreeRTOS extension that implements a subset of the POSIX threading Application Programming Interface (API) [35]. This extension wraps the FreeRTOS tasks API with POSIX threads (pthreads) API, thus allowing usage of pthreads API to control the execution of FreeRTOS tasks. Only 20% of the pthreads API is implemented. Therefore, libraries written using pthreads can not be ported directly to FreeRTOS using only this extension. [36]

2.4.3 WebAssembly

The goal of this project is to create a real-time concept of SmartCGMS that could be compiled for embedded platforms (e.g., ESP32, Raspberry Pi Zero W) and general-purpose computers (e.g., an x86-64 machine running Windows) with minimal differences in the source code. A typical scenario of using the concept might look like this: generate a configuration using the SmartCGMS GUI, optimize its parameters on a general-purpose computer with higher computing power, and then upload the code to an embedded device receiving data from a Continuous Glucose Monitoring (CGM) system and controlling an insulin pump.

There are multiple ways to create portable software. One of them is CMake. CMake is a software to manage build systems. It can support multiple platforms, but it usually requires slight modifications to the source code for each platform (e.g., in the form of C/C++ preprocessor directives). WebAssembly (WASM), on the other hand, is a portable binary format for a virtual stack machine [22]. It is, therefore, possible to run the same compiled binary on any machine implementing the WASM virtual instruction set. It was originally intended for web browsers, but the virtual instruction set has also been implemented in non-web environments [37]. These standalone runtimes also support some embedded devices (e.g., the ESP32) [19] [20]. WASM supports the compilation of programs written in different programming languages (e.g., C, C++, Rust, C#) [38]. This makes it a good candidate for developing a multi-platform prototype of SmartCGMS, as suggested by Otta [18]. The WASM format is described in section 2.3.1. This section describes WASM tools and toolchains used in this project.

2.4.3.1 Emscripten

Emscripten is a complete compiler toolchain for WASM. It supports C and C++ and other languages that support Low Level Virtual Machine (LLVM) to WASM compilation. Emscripten targets mainly the web and Node.js WASM runtimes. Alongside the compiled WASM binary, it produces JavaScript glue code that ensures the WASM module can run (e.g., it imports required functions). Emscripten provides its own implementation of the standard C and C++ library. In addition to exporting functions from the compiled C/C++ code, Emscripten provides ways for the C/C++ code to execute JavaScript. All functions exported from the WASM module must be declared as C functions (using `Extern "C"` to prevent C++ name mangling). [39] [40]

2.4.3.2 WebAssembly System Interface

As WASM gained popularity outside of browsers, there is a need for standardization of the system interface. This is what WebAssembly System Interface (WASI) aims to resolve. WASM requires access to system resources (e.g., system calls) to utilize the full potential of the environment it is running in. WASI is a system interface between the WASM module and the host system. It provides a standardized API to the WASM module that can be used to access the system resources. It is up to each WASM environment to implement this interface.

2.4.3.3 WASI Software Development Kit

WASI Software Development Kit (SDK) is a toolchain that combines WASI C library implementation, Clang and LLVM to one SDK. It is used to compile C and C++ into WASM. It offers a Docker image for a portable way to build WASM modules. The WASI SDK supports most of C and C++ functionality. However, C++ exceptions are not yet supported, and threads are still an experimental feature without long-term stability. Regardless, WASI is in constant development, and more features are planned to be supported in the future. [37] [41]

2.4.3.4 Standalone WebAssembly Runtime

When running WASM outside of the browser environment, a standalone WASM runtime is required. Standalone WASM runtime is an application or a library that implements the WASM virtual stack machine. Such a runtime can then be embedded into an existing application outside of the web browser. There are multiple standalone WASM runtimes available. The most popular (based on GitHub stars) standalone WASM runtimes at the time of writing this project are:

- Wasmer [42]
- Wasmtime [43]
- WebAssembly Micro Runtime (WAMR) [19]
- Wasm3 [20]
- wazero [44]
- WasmEdge [45]

Of these runtimes, only two officially support real-time devices: WAMR and Wasm3. The other runtimes are targeted at systems running GNU/Linux, Microsoft Windows, or MacOS operating systems. Both WAMR and Wasm3 support Xtensa, ARM, and x86-64 Instruction Set Architectures (ISAs), and both support WASI. Wasm3 is currently in a phase of minimal maintenance. Therefore, WAMR has been selected as the standalone WASM runtime for this project.

2.4.3.5 WebAssembly Micro Runtime

WAMR is a standalone WASM runtime. It has a small memory footprint, high performance, and many configurable features [19]. Features that are relevant for this project are WASI threads support, pthread API, and out-of-the-box support for Xtensa (ESP-IDF) and x86-64 with prepared examples. It supports three modes to execute a WASM module:

- Ahead-Of-Time (AOT) compilation mode
- Interpreter mode
- Just-In-Time (JIT) compilation mode

When using the AOT mode, the WASM module is compiled into native machine code ahead of deployment to WAMR. The WASM module is compiled using `wamrc` compiler. It needs to be compiled separately for each different platform. Its advantage is nearly native execution speed and very quick startup. It also has a very small memory footprint. [46]

The interpreter mode is the slowest of the three modes. It has a small memory footprint. The WASM instructions are interpreted by WAMR during runtime. WAMR offers two interprets: classic and fast. The fast interpreter is 2 times faster but consumes more memory. The classic interpreter supports source code debugging. [46]

JIT mode offers the same execution speed as AOT and is platform-agnostic. The disadvantage is a long startup time as the WASM module is compiled during execution. [46]

Proposed System 3

The proposed system is a SmartCGMS concept that can be compiled for low-power devices and general-purpose computers with minimal modifications. The project requirements specify two low-power platforms: Raspberry Pi Zero W and ESP32. However, additional platforms might be required in the future. Therefore, the SmartCGMS concept should be easily portable. As suggested by Otta in [18], one of the supported platforms will be WebAssembly (WASM). Then, any platform implementing the WASM virtual stack machine will be able to execute the SmartCGMS concept WASM module. The scenario of compiling the proposed SmartCGMS concept application is displayed in figure 3.1. The SmartCGMS concept application comprises three parts: the SmartCGMS framework source code, a SmartCGMS filter chain, and a platform environment (e.g., FreeRTOS source code, drivers). Throughout this thesis, the `SmartCGMS concept application` will refer to the executable for a specific platform or its source code. The `SmartCGMS concept` will refer to the whole proposed system used to compile SmartCGMS concept applications. The current state of the SmartCGMS framework will be referred to as the `original SmartCGMS` or just `SmartCGMS`.

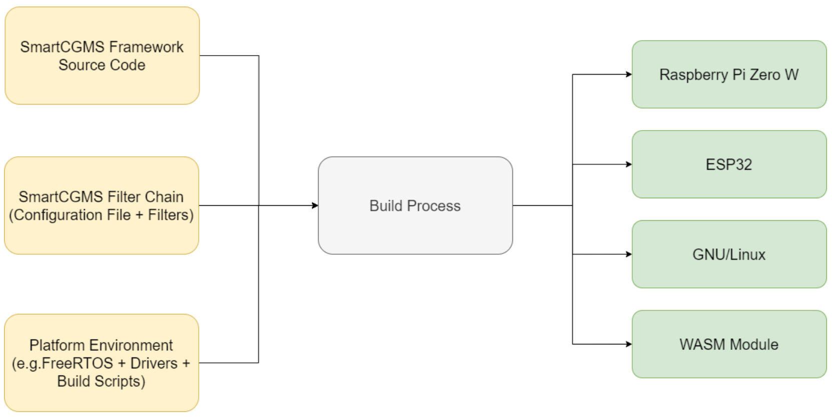

SmartCGMS Framework Source Code
SmartCGMS Filter Chain (Configuration File + Filters)
Platform Environment (e.g.FreeRTOS + Drivers + Build Scripts)
Build Process
Raspberry Pi Zero W
ESP32
GNU/Linux
WASM Module

Figure 3.1: Steps to deploy the SmartCGMS concept to different platforms

This chapter describes the proposed SmartCGMS concept, the design choices, and the platforms on which the SmartCGMS concept applications will be tested.

3.1 System Design

SmartCGMS is written in C++ using the C++17 standard. It relies heavily on the C++17 features, namely the C++17 standard filesystem library. Although C++ is considered portable, many low-power devices still lack the support for some C++17 features, such as the C++17 standard filesystem library. The SmartCGMS framework requires the filesystem library to load its components (the SmartCGMS core and common libraries) and to manage entities (e.g., filters) as each entity is implemented as a dynamic library.

3.1.1 Static Linking

Given the lack of support for the C++ standard filesystem library on low-power devices, it would be necessary to implement a way to manage dynamic libraries for each low-power device. This would complicate the introduction of new platforms that are compatible with the concept. Therefore, the SmartCGMS concept applications will be linked statically at compile time instead.

3.1.2 Filter Preprocessor

SmartCGMS uses functions exported by the filters to construct the filter chain. The exported function names of different filters in the original SmartCGMS are identical (`do_create_filter` and `do_get_filter_descriptors`). When linking the SmartCGMS concept applications statically, identical function names would lead to linker conflicts. To avoid this, each filter in the SmartCGMS concept applications is required to export functions with unique names.

This approach is not compatible with the original SmartCGMS. To prevent the need to develop separate filters for the SmartCGMS concept and the original SmartCGMS, a filter preprocessor tool is proposed. The purpose of the preprocessor is to modify the source code of the filters developed for the original SmartCGMS to avoid linker conflicts.

Conflicting function names are appended with a suffix to ensure each function has a unique signature. To minimize modifications to the SmartCGMS codebase, the preprocessor generates additional source code files. These files replace parts of the original SmartCGMS that manage the filters' dynamic libraries. Instead, the filter functions with modified names are called directly.

This tool ensures that SmartCGMS filters do not need to be implemented specifically for the SmartCGMS concept. A filter can be developed as a dynamic library satisfying the requirements of the original SmartCGMS. It is then modified by the preprocessor and used in the SmartCGMS concept without any modifications done by the developer.

3.1.3 Preserving Modularity

It is important to keep the SmartCGMS concepts modular. To achieve this, the separation between the SmartCGMS codebase and the filter chain is maintained. The proposed system can be split into three main components:

- SmartCGMS codebase
- Platform environment
- Filter chain

The SmartCGMS codebase includes the original SmartCGMS codebase (the core and common libraries). SmartCGMS offers a simple interface that allows using the SmartCGMS framework from other programming languages. It is a C-style Application Programming Interface (API). The simple API functions are: create a filter chain specified by a configuration file, shut the filter chain down, and inject events into the filter chain. This interface can be used for the SmartCGMS concept as it offers a basic set of functions to interact with SmartCGMS. Missing functionality can be implemented. The SmartCGMS codebase might remain unchanged for each platform except for one file. SmartCGMS uses a WinAPI mapping file to provide access to system resources. This file needs to be updated when a new platform is introduced.

The platform environment code defines the behavior of the SmartCGMS concept application. From the environment source code, the SmartCGMS API is called. Additionally, the FreeRTOS source code, drivers, and other necessary software components belong to the platform environment. Build scripts can also be provided to simplify the compilation process of the SmartCGMS concept application. CMake will be used to generate build files for all platforms.

When creating filters for the SmartCGMS concept, many can be implemented as platform-independent, such as filters computing a signal value. Whereas some filters are, by design, platform-dependent. This includes input filters, such as a filter listening on a peripheral device for incoming events, and output filters generating output. The reason for their platform dependency can be, for example, a display driver specific to a target platform. Any filter that requires asynchronous processing of information and needs to create a new thread will also need to be developed with a specific platform in mind (e.g., using FreeRTOS tasks in low-power devices running FreeRTOS versus using standard thread library from C++ in platforms that support it).

3.1.4 WebAssembly

In addition to targeting traditional Instruction Set Architectures (ISAs) like AMD64/x86-64, ARMv6, and Xtensa, the goal is to support WASM for maximum portability. There are no obstacles that would limit compiling a SmartCGMS concept application to WASM. The issue with WASM lies in preparing the platform environment to support WASM. A WASM standalone runtime must be ported to the platform to be able to execute WASM modules. Although WASM is not yet widely supported by many embedded platforms, there is a growing effort to standardize WASM interaction with the underlying platform (WebAssembly System Interface (WASI) - described in section 2.4.3.2) and bring it to new platforms.

Compiling the SmartCGMS concept application to WASM offers two main advantages. Firstly, it allows the software to be compiled just once and then transmitted in the WASM binary format to other platforms. This eliminates the need for platform-specific compilation. Secondly, it brings SmartCGMS to the web. The ability to use SmartCGMS from the web browser makes it more accessible to users and potentially speeds up its adoption. A drawback of WASM is the complex process of porting WASM runtimes to new platforms.

3.2 Target Architectures

This section describes the specific platforms representing each ISA specified by the project requirements.

3.2.1 Low-Power Devices

The ARMv6 and Xtensa ISA devices represent low-power devices that could be used in the field (e.g., worn by a diabetic patient). Raspberry Pi Zero W and ESP32 chips have been selected for this project.

3.2.1.1 ESP32

ESP32 is a microcontroller unit with ultra-low power consumption, robust design, and connectivity (Wi-Fi and Bluetooth) [47]. It is widely adopted in the Internet of Things applications due to its price and performance [48]. The board has an Xtensa LX6 Dual-Core 240 MHz CPU, 520 KB of SRAM, and 2 MB of FLASH (with support up to 64 MB) with an operating current of 80 mA. The development for the board is simple, thanks to the Espressif IoT Development Framework (ESP-IDF) -

ESP32 official development framework. It provides an easy way to build and flash ESP32 projects to the board. ESP-IDF is based on FreeRTOS, thus FreeRTOS API is available when developing applications for ESP32. ESP-IDF build system is based on CMake. It offers additional syntax to reduce boilerplate, but pure CMake can also be used to configure projects. ESP-IDF's build system uses a concept called components. Components are static libraries linked to the ESP32 application that are defined using CMake files. There are official components offered by ESP-IDF (e.g., FreeRTOS), but one can also implement their own components. The ease of use and availability make the ESP32 a good candidate for this project. Additionally, it has been selected by Otta in [18] as the hardware of choice. [47][34]

3.2.1.2 Raspberry Pi Zero W

Raspberry Pi Zero W is a compact and cheaper alternative to the Raspberry Pi microcontroller board. The 'W' represents connectivity, namely Wi-Fi and Bluetooth. It has a BCM2835 ARMv6 1Ghz single-core CPU, 512 KB of SDRAM, and support for a microSD card. Its operating current is 100 mA [49]. An official port of FreeRTOS [32] is not available for the board. The Raspberry Pi Zero W offers more computing power and memory than the ESP32 at the cost of increased energy usage. It has been selected for this project due to its availability and similarity to the ESP32.

Raspberry Pi Foundation offers an operating system for the Raspberry Pi Zero W - the Raspberry Pi OS (previously called Raspbian). It is based on Debian, and it provides a desktop environment called PIXEL (Pi Improved Xwindows Environment, Lightweight) [49]. It is not an Real-time operating system (RTOS). Therefore, FreeRTOS has been ported to Raspberry Pi Zero W and used instead.

3.2.2 High Performance Computing

The AMD64/x86-64 ISA represents computers with high processing power compared to the Raspberry Pi and ESP32. The reason why SmartCGMS is targeted at these platforms is to run computationally intensive simulations.

The SmartCGMS console application was developed to preserve system resources and can be used for computationally extensive tasks [27]. The SmartCGMS concept does not aim to offer better processing performance than the existing SmartCGMS solutions. The reason the proposed SmartCGMS concept is compatible with general-purpose computers (e.g., x86-64 running GNU/Linux) is to be able to verify the correct functionality of a filter chain before deployment to a low-power device.

Implementation 4

This chapter presents the implementation of the SmartCGMS concept proposed in chapter 3. The first section describes modifications to the SmartCGMS codebase. The second section describes the implementation of the proposed filter preprocessor tool. The third section describes the SmartCGMS concept Application Programming Interface (API). The fourth section describes filters implemented to test and showcase the proposed concept. The last section describes the platforms for which the concept application has been compiled and the differences between the platforms.

4.1 SmartCGMS Codebase

As already discussed in chapter 3, to make the SmartCGMS concept easily portable to new embedded platforms, the SmartCMGS concept application needs to be linked statically instead of dynamically. This section describes the necessary changes to the SmartCGMS codebase to compile the SmartCGMS concept application for low-power platforms.

4.1.1 Functions Incompatible with Embedded Environments

The original SmartCGMS codebase contains functions that can not be supported in a portable embedded environment and functions related to SmartCGMS's dynamic linking nature. These functions were either removed or modified to avoid any unwanted side effects (e.g., by removing the function body and returning a default value). The decision of whether to remove a function or modify it was made individually for each function. The goal was to keep the code readable and minimize modifications to the source code. Most of the modifications are implemented using preprocessor `#if defined()` directives, with the macro name `EMBEDDED`.

4.1.1.1 SmartCGMS Interfaces

The SmartCGMS common library defines interfaces that each entity must implement (e.g., `scgms::IFilter` for filters). Some of the interfaces contain functions that rely on filesystem API. As the low-power platforms selected for the SmartCGMS concept do not support C++ standard filesystem API, these functions were removed from the interfaces.

4.1.2 Resolving Symbols

When a SmartCGMS application is started, it loads all the dynamic libraries from its working directory. To resolve functions between the SmartCGMS core and common dynamic libraries, it uses a function called `resolve_symbol`. It is a wrapper for the `GetProcAddress` function, which retrieves the address of an exported function from a dynamic library. In the SmartCGMS concept, the `resolve_symbol` function can call the functions directly as the concept applications are compiled as monolithic executables. The function in the SmartCGMS concept codebase is defined in a new file `staticLink.cpp` and is called `resolve_symbol_static`. To be able to resolve the functions, the headers declaring the functions need to be included in the file implementing the `resolve_symbol_static` function. An example of how the functions are resolved can be seen in figure 4.1.

```cpp
void* resolve_symbol(const char *symbol_name) noexcept
{
  if (strcmp(symbol_name, "create_device_event") == 0)
  {
    return reinterpret_cast<void*>(create_device_event);
  }
  if (strcmp(symbol_name, "get_filter_descriptors") == 0)
  {
    return reinterpret_cast<void*>(get_all_descriptors);
  }
  if (strcmp(symbol_name, "execute_filter_configuration") == 0)
  {
    return reinterpret_cast<void*>(execute_filter_configuration);
  }
  return nullptr;
}
```

Source code 4.1: Example implementation of the resolve_symbol_static() function

4.1.3 Managing Entities

To manage entities, SmartCGMS uses a class called `CLoaded_Filters`. This class stores information about each entity (e.g., its descriptors, handle to its dynamic library). On startup, SmartCGMS checks each dynamic library in its project folder if it exports functions defined for a SmartCGMS entity. If a library exports the defined functions, it is then stored in this class. When an action is requested on a specific entity, the `CLoaded_Filters` class can then call the corresponding function by using the dynamic library handle. The entities are identified by a globally unique identifier (GUID). The `CLoaded_Filters` class does not store the GUIDs of the entities. When executing a function (e.g., `do_create_filter`), it calls the function from each dynamic library until one of the calls returns a success return code (each entity checks the GUID passed as an argument to the function and executes the function only if the GUID matches its own). The `CLoaded_Filters` class is replaced in the SmartCGMS concept by a class with the same name. The class is completely generated by the preprocessor tool.

Filter Descriptors

The newly implemented class stores the filter descriptors (`TFilter_Descriptor`) in an `std::vector`. These descriptors are loaded when the class is instantiated. To load the descriptors, the class defines a lambda function to copy the descriptors of each filter to the `CLoaded_Filters` class, see 4.2.

```cpp
using FilterDescriptorFunction =
HRESULT(*)(scgms::TFilter_Descriptor**, scgms::TFilter_Descriptor**);

auto get_descriptors = [&](FilterDescriptorFunction get_filter_descriptors)
{
  scgms::TFilter_Descriptor *desc_begin, *desc_end;
  bool result = get_filter_descriptors(&desc_begin, &desc_end) == S_OK;
  if (result)
  {
    std::copy(desc_begin, desc_end, std::back_inserter(mFilter_Descriptors));
  }
};
```

Source code 4.2: Lambda function to replace `do_get_filter_descriptors` calls to dynamic libraries

The exported `do_get_filter_descriptors` function of each filter is passed as a parameter to the lambda function and then the filter descriptors are copied to the `std::vector` `mFilter_Descriptors` in `CLoaded_Filters` class.

Creating Filters

The original SmartCGMS codebase contains a function called `create_filter_body`. This function is called when a filter needs to be constructed. The filter is specified by its GUID. The function iterates through the filters' dynamic libraries stored in `CLoaded_Filters` and calls the `do_create_filter` functions until one of the calls returns signaling success. As the SmartCGMS concept's `CLoaded_Filters` class only stores the filter descriptors and can not call the function of the filters, the `create_filter_body` function also needs to be replaced. The new implementation simply unrolls the loop of the original function and in each step calls the `do_create_filter` function for a different filter as seen in listing 4.3.

```
scgms::SFilter create_filter_body(const GUID &id, scgms::IFilter *next_filter)
{
  scgms::SFilter result;
  scgms::IFilter *filter;

  if (do_create_filter_data_filter(&id, next_filter, &filter) == S_OK)
  {
    result = refcnt::make_shared_reference_ext<scgms::SFilter,
         scgms::IFilter>(filter, false);
    return result;
  }
  if (do_create_filter_ema_filter(&id, next_filter, &filter) == S_OK)
  {
    result = refcnt::make_shared_reference_ext<scgms::SFilter,
         scgms::IFilter>(filter, false);
    return result;
  }
  if (do_create_filter_watchdog_filter(&id, next_filter, &filter) == S_OK)
  {
    result = refcnt::make_shared_reference_ext<scgms::SFilter,
         scgms::IFilter>(filter, false);
    return result;
  }
  if (do_create_filter_print_filter(&id, next_filter, &filter) == S_OK)
  {
    result = refcnt::make_shared_reference_ext<scgms::SFilter,
         scgms::IFilter>(filter, false);
    return result;
  }

  return result;
}
```

Source code 4.3: Example implementation of the create_filter_body() function

4.2 Filter Preprocessor

As the original SmartCGMS is a set of dynamic libraries and each filter is its own dynamic library, the exported functions of each filter can have the same function signature and not cause conflicts at the linking stage when building a SmartCGMS application. In the SmartCGMS concept, the function names are modified, as seen in figure 4.3. This is to prevent linking conflicts. To prevent developers from having to write separate filters for the original SmartCGMS and the SmartCGMS concept and to keep SmartCGMS entities' API unchanged, a filter preprocessor tool has been proposed and implemented.

The source code modifications described in section 4.1.3 all come from the same source and header files `filters.cpp` and `filters.h`. These files are not included in the SmartCGMS concept codebase. Instead, they are generated by the filter preprocessor tool and added for each filter chain subsequently. The reason is that each additional filter in a filter chain requires its functions to be added to the `filters.cpp` source code file.

The filter preprocessor tool takes a folder containing an original SmartCGMS filter chain source code as input and generates a folder containing a SmartCGMS concept filter chain source code as output. The expected input folder structure is that each subfolder of the input folder contains a single filter implementation. The name of the root folder of each filter is used to identify the filters in the generated files. The source code of each filter is searched and then modified by appending the root folder name as a suffix to the functions that are exported by the filters (`do_create_filter` and `do_get_filter_descriptors`). The definitions of the new functions are added to the generated header file. All the modified names of the functions are stored and then used to generate the body of the `create_filter_body` function and the function to gather filter descriptors to `CLoaded_Filters` using the lambda function from listing 4.2.

SmartCGMS offers a way to load the filter chain configuration from memory. The filter preprocessor tool copies the .ini configuration file to a C++ raw string literal from which it can then be loaded in the SmartCGMS concept.

The filter preprocessor does not check if the provided filter chain contains all filters specified by the configuration file. If there is a filter specified in the configuration but the source code files for it are not provided to the preprocessor, the issue will be discovered at runtime after building the SmartCGMS concept application. In case there is an error in the configuration of the filter chain, it can be resolved by uploading a new configuration at runtime. If the error comes from a missing filter, then the whole application has to be rebuilt.

4.3 SmartCGMS Concept API

To interact with the SmartCGMS concept, the original SmartCGMS API can be used. Only parts of the original SmartCGMS API have been implemented for the SmartCGMS concept due to time constraints. Additionally, a new minimal C-style API (similar to the simple interface of SmartCGMS) has been implemented to test the functionality of the proposed system. The introduced API allows usage of the SmartCGMS concept without any underlying knowledge of SmartCGMS. There are three interface functions:

- `int build_filter_chain(const char* configuration_input)`
- `const char* get_config_data()`
- `void create_event(const SCGMS_Event* event)`

The `build_filter_chain` function is used to construct the filter chain. It takes a C string as input. If an input string is not provided, the filter chain is built using the configuration with which the SmartCGMS concept application has been compiled. The default configuration can be retrieved as a C string using the `get_config_data` function. This allows an application using the SmartCGMS concept to easily edit the default filter parameters. The configuration can be retrieved before the filter chain is built. The last function `create_event()` injects events into the filter chain. This function can be used to send data to the filter chain when it is received outside of the SmartCGMS filter chain. The SmartCGMS filter chain can be shut down by creating a shutdown event.

4.4 Filters

Four filters have been developed for the SmartCGMS concept demonstration:

- Data reading filter
- Data transformation filter
- Data visualization filter
- Watchdog filter

The filters have been developed to support all the platforms specified in section 4.5. In cases where the implementation needed to differ, preprocessor directives were used. The filters were implemented as dynamic libraries to maintain compatibility with the original SmartCGMS. Filters that do not rely on platform-specific features can be used as dynamic libraries on platforms supported by the original SmartCGMS. The following sections describe the implemented filters.

4.4.1 Data Reading Filter

The data reading filter is meant to be an input filter of a filter chain, but it can also be used to store statistics at the end of a filter chain. The common part for all platforms acts as a simple event statistic counter. The filter logs the number of all events passing through it. The filter additionally logs the number of error events, warning events, level events, and information events. When the filter receives a shutdown event, it prints the gathered statistics to the console. An example scenario of how the data reading filter could be used is depicted in figure 4.1. One data reading filter can be set up to log incoming events into the filter chain and one can be positioned at the end of the filter chain. On the filter chain shutdown, the statistics can be compared and, for example, a number of error events generated by the filter chain can be calculated.

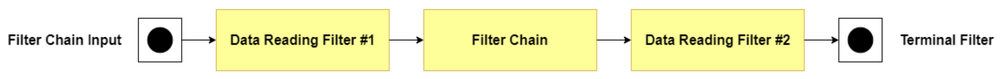


Figure 4.1: Example of a SmartCGMS filter chain containing data reading filters at the beginning and the end to calculate event statistics.

The data reading filter for the ESP32 platform creates a new task that listens on the UART interface. It expects to receive numbers in the double-precision format on the UART interface. Whenever a number is received, a SmartCGMS level event is generated using the received number as the level value and sent into the filter chain. The UART task is created during the configuration of the data reading filter. The filter can still process events originating from previous filters or outside the filter chain using the `Execute` function. The task creation during the filter configuration is optional and is set using a `create_task` parameter in the configuration file to either `true` or `false`. On the Raspberry Pi Zero W platform, the data reading filter can create a task that reads level values from an `input.h` file that is compiled with the filter source code. When a shutdown event is received by the data reading filter and the task has been started, the filter shuts it down synchronously.

The WebAssembly (WASM) platforms do not support creating a task (or a POSIX thread (pthread)) for reading or generating data since the selected environments for WASM offer only experimental support for pthreads (WebAssembly Micro Runtime (WAMR)) or since it would greatly increase resource management and synchronization complexity (using JavaScript multithreading on the web). The parameter `create_task` on these platforms is ignored.

4.4.2 Data Transformation Filter

The data transformation filter implementation is portable across all the platforms supported by the SmartCGMS concept and the original SmartCGMS. The filter computes the Exponential Moving Average (EMA) value of the event levels. An EMA is a moving average that applies decreasing weight to older values. The formula for the EMA calculated in the filter can be seen in listing 4.4. The accumulator value is the EMA value. The alpha is a constant parameter that modifies the ratio at which the weight of older data decreases. When constructing the filter the alpha constant is read from the configuration file using an `alpha` parameter that can be set to a value between 0 and 1. The filter computes the EMA from the level value of SmartCGMS level events. All other events are passed through unmodified.

```
accumulator = (alpha * event.level()) + (1.0 - alpha) * accumulator;
```

Source code 4.4: Exponential moving average calculation

4.4.3 Data Visualization Filter

The data visualization filter has a specific implementation for each platform. It does not have any parameters that can be specified in the configuration file and it does not modify the passing events in any way. On embedded platforms and in the WASM environment targeted at standalone WASM runtimes, the filter prints the event level values and information event strings. For each event, it also prints the device time. On the embedded platforms the data is sent to UART. In the WASM standalone runtime, the WebAssembly System Interface (WASI) is used to print the data to the console on the hosting system.

For the WASM environment targeting the web, Emscripten is used to call a JavaScript function from the C++ source code. Emscripten provides a C-style function `emscripten_run_script` that takes a C string as an argument and executes it as JavaScript code in the host environment. The call to the `emscripten_run_script` function is shown in listing 4.5. The generated JavaScript code can be seen in listing 4.6. The generated JavaScript code creates a message object and calls a function `postData` providing the object as an argument. This approach allows developers to tailor the `postData` function implementation to a specific application and it does not expose the implementation to the data visualization filter. The message object can be extended with more information about the event, but for demonstration purposes, it has been kept simple.

```
std::string js_code = R"(
  var message = {
  data_level: )" + std::to_string(event.level()) + R"(,
  data_device_time: )" + Rat_Time_To_Local_Time_Str(event.device_time(),
      "%H:%M:%S") + R"(,
  data_logical_time: )" + std::to_string(event.logical_time()) + R"(
  };
  postData(message);
  )";
emscripten_run_script(js_code.c_str());
```

Source code 4.5: Calling JavaScript to post event data to the web environment running the SmartCGMS WASM module.

```
var message = {
    data_level: 101.348,
    data_device_time: 13:45:22,
    data_logical_time: 76
};
postData(message);
```

Source code 4.6: An example of JavaScript code contained in the `js_code` string from listing 4.5

The SmartCGMS concept application can be executed asynchronously on the web as a JavaScript web worker (JavaScript thread equivalent). The `postData` function then has to be changed to `self.postMessage`. This posts the message object to the thread that spawned the SmartCGMS web worker. The object can then be handled inside `worker.onmessage` handler in the thread that controls the SmartCGMS web worker.

4.4.4 Watchdog Filter

The purpose of the watchdog filter is to ensure a correct run of the filter chain, to notify about any unusual behavior, and possibly to shut the filter chain down. The implementation monitors two things. Firstly, if the filter chain is active (an event passing through before a specified timeout). Secondly, if the events in the filter chain are chronologically ordered (the logical event time is increasing). The first condition is monitored with a task that runs asynchronously. The task has a set timer, when it expires the watchdog checks whether any events passed through. If the watchdog hasn't been reset by a passing event, it triggers an action (function `Trigger` is called). When the `Do_Execute` function is called (upon receiving an event), it calls a `Kick` function. This function takes the event's logical time as a parameter and it compares it to the logical time of the previous event. If the new event's logical time is lower than the previous event's time (the new event is older) the watchdog also triggers an action.

The watchdog task is implemented as a pthread for the WASM platforms and the ESP32. For Raspberry Pi Zero W the watchdog task is implemented as a FreeRTOS task. In case of porting this filter to a new platform, it is required to implement the `Trigger` function and to create the watchdog task during the filter configuration. The watchdog filter has one parameter - the timeout timer interval in milliseconds.

4.5 Supported Platforms

This section describes platforms for which an environment was created to support the SmartCGMS concept. There are 4 platforms (in brackets are specified the platform's Instruction Set Architectures (ISAs)):

- Raspberry Pi Zero W (ARMv6)
- ESP32 (Xtensa)
- WASM targeted at the web (x86-64)
- WASM targeted at a standalone runtime (portable)

The Raspberry Pi Zero W and ESP32 represent embedded devices. Their environment is based on FreeRTOS. The ESP32 environment uses the Espressif IoT Development Framework (ESP-IDF), which offers additional software components. The WASM platforms use the standard C library built on top of WASI system calls. SmartCGMS uses a `winapi_mapping.cpp` file to provide a wrapper around functions from different platforms (e.g., function to allocate memory). Functions to allocate memory for each platform were added to this file.

The directory layout is similar for all platforms. The SmartCGMS codebase and filter chain are separated as proposed in section 3.1.3. The platform environment from section 3.1.3 refers to the whole directory of the SmartCGMS concept application for a specific platform (including build scripts, CMake files, and FreeRTOS source code).

A Python script has been implemented to assist the development. First, it downloads the platform environment from a remote repository, followed by the SmartCGMS codebase. The filter chain is taken from an input directory. The Python script then uses the preprocessor tool to modify the filters' source code and generate the required files. Finally, all the source code files are combined in the downloaded platform environment directory. Each platform uses CMake to simplify the build process of the SmartCGMS concept.

The platform environments and the Python script generating the build directories are targeted at GNU/Linux. Therefore, the Python script has been Dockerized to allow development on other platforms. The Python script generates an additional CMake file that contains environment variables specific for each platform (e.g., path to ARM embedded toolchain for Raspberry Pi Zero W or path to the ESP-IDF for the ESP32). The SmartCGMS concept application can then be built using a simple build shell script that is also provided for each platform. To avoid local installation of multiple toolchains, Docker images that are provided for the WASM and ESP32 platforms on their official repositories can be used.

4.5.1 Raspberry Pi Zero W

The ARMv6 ISA requires an implementation of the C++ atomic library helper functions to be provided, as it does not implement the atomic library [50]. This step has been omitted due to project time constraints and atomic variables in the SmartCGMS codebase have been replaced by their non-atomic equivalents. Preprocessor directives have been used to maintain atomicity on other platforms.

A custom port of FreeRTOS has been used. FreeRTOS-Plus-POSIX extension has been added to allow all platforms to use pthreads to manage tasks. To allocate memory, FreeRTOS `pvPortMalloc` needs to be called.

When creating a SmartCGMS concept application for the Raspberry Pi Zero W the data for the filter chain (e.g. blood glucose levels from a sensor sent over UART) can be received and processed directly in the SmartCGMS filter chain using a specialized filter. The data can also be received and processed in a separate FreeRTOS task that sends the events into the SmartCGMS filter chain. Using the second approach, the SmartCGMS concept can be easily implemented into an existing FreeRTOS application.

4.5.2 ESP32

The ESP32 environment uses the ESP-IDF. The SmartCGMS concept application is divided into three distinct parts: the SmartCGMS codebase, the filter chain, and the main application. The first two parts are implemented as ESP-IDF components. The main application then uses these components. The SmartCGMS codebase and the filter chain components depend on each other and can not be used on their own. The filter chain component's name is `filters`. The SmartCGMS codebase component is called `scgms_embedded`. The main application is expected to interact with the `scgms_embedded` component. The API functions described in section 4.3 are defined in this component in a header file `scgms.h`.

Similarly to the Raspberry Pi Zero W, the ESP32 SmartCGMS concept can be used in two ways. Either the main application can receive data and then send it into the SmartCGMS filter chain, or it can build a filter chain that creates its own FreeRTOS task for receiving data (e.g., by using the UART data reading filter from section 4.4).

4.5.3 WASM Standalone Runtime

For the standalone WASM runtime, the SmartCGMS concept application is compiled to the WASM binary format using Clang. To provide C standard library implementation and the WASI, WASI Software Development Kit (SDK) is used for the build. The platform environment has a similar structure to the embedded environments. It contains two directories: the SmartCGMS codebase and the filter chain, plus additional files like CMake files, build scripts, and a *main.cpp* source code file. To control what functions are exported from the WASM module, the `main.cpp` and the `CMakeLists.txt` file can be edited. Currently, the WASM module exports the main function, which is started directly after loading the module. The main function builds the filter chain specified by the configuration file provided by the preprocessor. The WASM module can be optionally set up as a library (without a main function). The WASM compiled SmartCGMS concept supports asynchronous filters. A filter can create a thread using pthreads. Although WASI SDK offers only experimental support for threads. The WASM module compiled for the standalone WASM runtime can be executed even on the web. To run the WASM module, the platform needs to support WASI calls used in the module. The WASI calls may differ for each WASM SmartCGMS concept application depending on the used filter chain (e.g., if using a filter with pthreads).

The standalone WASM runtime selected for this project is WAMR. Relevant features that are supported by WAMR are pthreads and WASI. WAMR also supports multiple ISAs including Xtensa, x86-64, and ARM. WAMR provides a command line interface application called `iwasm`. The application for this project has been compiled and used on GNU/Linux, though other x86-64 systems are also supported (e.g., MacOS and Windows). A WAMR application for the ESP32 is also available. It is based on the ESP-IDF and loads the WASM module from a C-style byte array from memory. To be able to execute the WASM module on the ESP32, the build script in the WASM standalone runtime environment also generates a C/C++ header file containing a byte array with the WASM module. As WASM is a binary format, this is achieved by hex dumping the WASM into the byte array. Although WAMR supports ARM ISA, the Raspberry Pi Zero W platform is not officially supported, and neither is FreeRTOS.

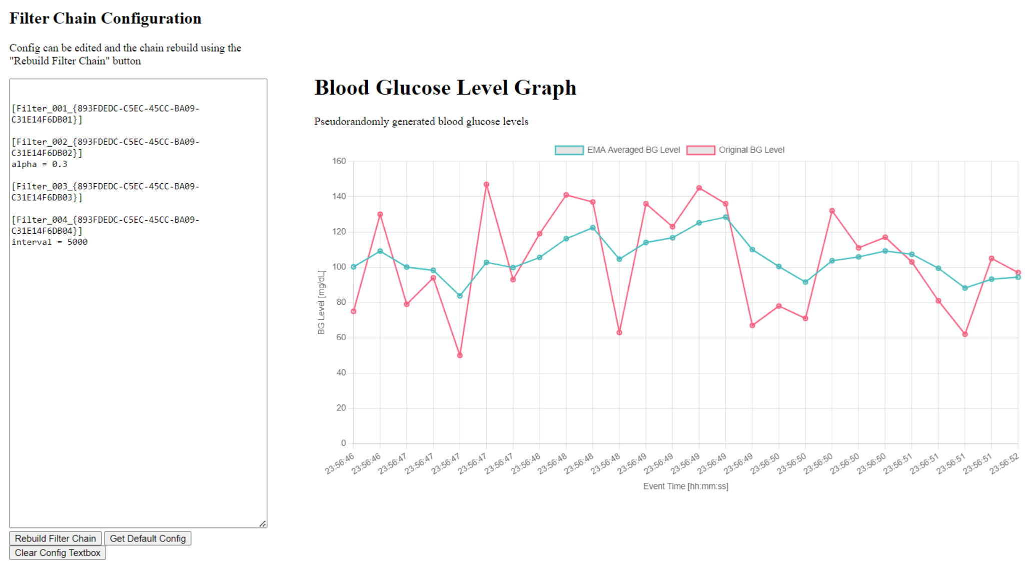
4.5.4 WASM Targeted at The Web

While the WASM module compiled using WASI SDK can be executed in a web browser, it is easier to use a WASM module that was compiled using the Emscripten toolchain. The reason is that Emscripten also generates JavaScript wrapper code to simplify the module's loading. Emscripten can also directly generate an HTML file that loads and executes the WASM module. The platform environment directory structure is the same as the directory structure for compiling WASM for standalone runtimes. One folder contains the SmartCGMS codebase, one contains the filter chain, and then there is an additional `main.cpp` file, which interacts with the SmartCGMS concept API and defines functions that are exported by the WASM module. The `CMakeLists.txt` does not have to be edited when exporting new symbols, as the Emscripten toolchain can automatically handle symbol exports from C++ code to JavaScript, see listing 4.7. Emscripten features that are relevant for this project are multi-threading support and the ability to execute JavaScript code inside the WASM module.

```cpp
extern "C" {
EMSCRIPTEN_KEEPALIVE
const char* get_config()
{
    return get_config_data();
}
}
```

Source code 4.7: An example of exporting symbols from WASM module using Emscripten

The WASM module compiled in the environment targeted at the web can also be executed in standalone runtimes, though WAMR does not recommend running Emscripten compiled WASM modules. Therefore, it is recommended to use WASM modules built in this environment only in the web browser. When the WASM SmartCGMS concept application is built, two files are created: the WASM module and a JavaScript code to load and execute the module. Additionally, a third file might be generated: a JavaScript web worker code. This file is generated when threads are created inside the SmartCGMS concept application. Using the JavaScript code, the module can be easily embedded into a webpage. To demonstrate the functionality of the WASM SmartCGMS concept, a demo webpage was created. The webpage uses JavaScript to load the compiled SmartCGMS concept WASM module. It provides buttons to build the filter chain and then injects events into it. The events are displayed using `Chart.js` (JavaScript data visualization library) in a line graph. The graph displays the original event level value injected into the filter chain and then the modified value provided by the data visualization filter (using the `postData` function), which is described in section 4.4.3. The filter chain can be built either with the configuration it was compiled with or with a custom configuration supplied from the webpage. The webpage has an editable text field to edit the configuration, which can then be sent to the SmartCGMS WASM module. To simplify the creation of a custom configuration, the default configuration can be loaded from the SmartCGMS WASM module (using the API function: `get_config_data()`). A filter chain configuration and event data in the line graph can be seen in figure 4.2. The data to the filters are pseudo-randomly generated, and the filter chain consists of the four filters from section 4.4 in the same order as they are described. Additional output from the SmartCGMS concept application can be seen in the web browser developer console, shown in figure 4.3.



Filter Chain Configuration

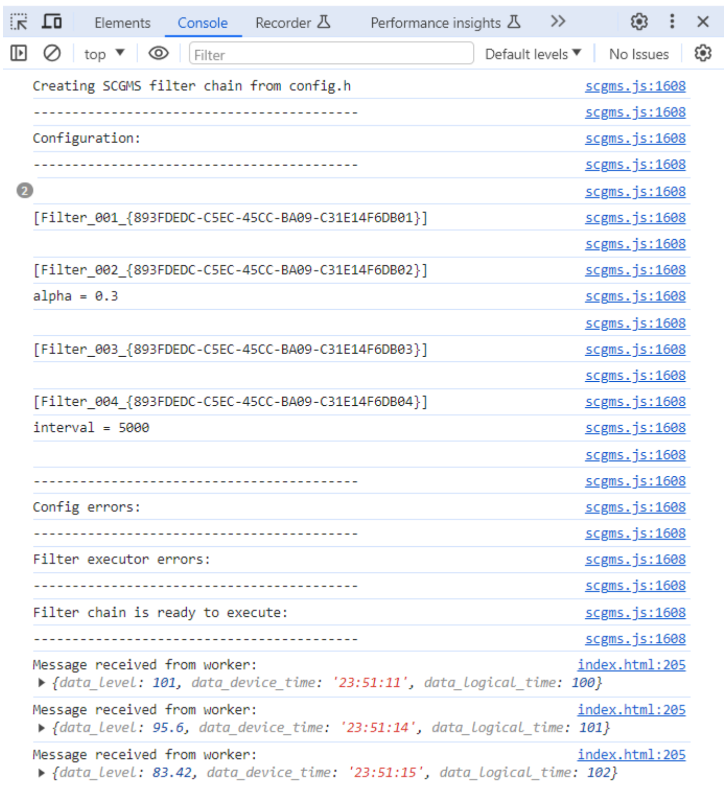
Config can be edited and the chain rebuild using the "Rebuild Filter Chain" button

```
[Filter_001_{893FDEDC-C5EC-45CC-BA09-
C31E14F6DB01}]

[Filter_002_{893FDEDC-C5EC-45CC-BA09-
C31E14F6DB02}]
alpha = 0.3

[Filter_003_{893FDEDC-C5EC-45CC-BA09-
C31E14F6DB03}]

[Filter_004_{893FDEDC-C5EC-45CC-BA09-
C31E14F6DB04}]
interval = 5000
```

Blood Glucose Level Graph

Pseudorandomly generated blood glucose levels

Figure 4.2: SmartCGMS WASM demo webpage: Filter chain configuration and pseudo-randomly generated data processed by the filter chain displayed in a `Chart.js` graph.



```
Creating SCGMS filter chain from config.h                      scgms.js:1608
------------------------------------------                     scgms.js:1608
Configuration:                                                 scgms.js:1608
------------------------------------------                     scgms.js:1608
2                                                              scgms.js:1608
[Filter_001_{893FDEDC-C5EC-45CC-BA09-C31E14F6DB01}]            scgms.js:1608
                                                               scgms.js:1608
[Filter_002_{893FDEDC-C5EC-45CC-BA09-C31E14F6DB02}]            scgms.js:1608
alpha = 0.3                                                    scgms.js:1608
                                                               scgms.js:1608
[Filter_003_{893FDEDC-C5EC-45CC-BA09-C31E14F6DB03}]            scgms.js:1608
                                                               scgms.js:1608
[Filter_004_{893FDEDC-C5EC-45CC-BA09-C31E14F6DB04}]            scgms.js:1608
interval = 5000                                                scgms.js:1608
                                                               scgms.js:1608
------------------------------------------                     scgms.js:1608
Config errors:                                                 scgms.js:1608
------------------------------------------                     scgms.js:1608
Filter executor errors:                                        scgms.js:1608
------------------------------------------                     scgms.js:1608
Filter chain is ready to execute:                              scgms.js:1608
------------------------------------------                     scgms.js:1608
Message received from worker:                                 index.html:205
 ▸ {data_level: 101, data_device_time: '23:51:11', data_logical_time: 100}
Message received from worker:                                 index.html:205
 ▸ {data_level: 95.6, data_device_time: '23:51:14', data_logical_time: 101}
Message received from worker:                                 index.html:205
 ▸ {data_level: 83.42, data_device_time: '23:51:15', data_logical_time: 102}
```

Figure 4.3: SmartCGMS WASM demo webpage: Additional output printed to the Google Chrome developer console by the SmartCGMS WASM module

SmartCGMS Web Demo

The SmartCGMS concept application compiled as a WASM module was extended to create simulation of a diabetic patient's blood glucose levels running on the web. Currently, it is deployed only locally.

The demo webpage was modified to allow input of meals, insulin boluses, and basal rate (simulated by small bolus doses). Each event sent to the filter chain advances the simulation time by 5 minutes. The filter chain configuration can be edited to modify the parameters of the filters or the model. Additionally, the parameters can be saved to a file from which they can be loaded. The file is stored in a virtual file system created by Emscripten. Multiple files can be created. This allows us to use filters that require files (e.g., log filter). However, the files stored in the Emscripten virtual file system are only temporary and are deleted when the webpage is reloaded. To maintain the content of the files, it needs to be copied when the simulation is over.

The model is managed by a signal generator filter. This filter is required when configuring the filter chain in the demo. Another required filter is the data visualization filter that sends the events' information back to the JavaScript code to update the graph. Additional filters can be added to the filter chain.

The SmartCGMS web demo supports two models: Samadi model based on [51] and S2013 model based on [52]. The models can be changed just by modifying the model's GUID in the configuration.

Evaluation 5

The goal of the project was to implement a portable SmartCGMS concept. To achieve this, the SmartCGMS codebase has been modified to use static linking during compilation, and C++ features that are not widely available in Real-time operating systems (RTOSs) have been removed. With these modifications, it is possible to compile the SmartCGMS concept as a native application for FreeRTOS or any platform that supports C++17.

To evaluate the SmartCGMS concept, it has been compiled for five different platforms. The five platforms are:

- Raspberry Pi Zero W running FreeRTOS (ARMv6)
- ESP32 running FreeRTOS (Xtensa)
- GNU/Linux (x86-64)
- WAMR (compiled using WASI SDK)
- SmartCGMS demo webpage (compiled using Emscripten)

The first three platforms use a native application for specified ISA and operating system, while the last two are compiled to WASM. A set of four filters has been developed (described in section 4.4. The filters can be compiled for any of the mentioned platforms.

The evaluation is divided into two parts. First, the correct functionality of the modified SmartCGMS framework and the developed filters is verified. Second, the operational characteristics of the SmartCGMS concept applications are measured and compared for different platforms. The next sections outline the experimental setup for each measurement, followed by the presentation and discussion of the gathered results.

5.1 Correctnes of Execution

To verify the correct functionality of the modified SmartCGMS framework, a test scenario consisting of several interactions with the SmartCGMS framework was created. The test scenario was executed on each platform, and the results were collected and compared. The goal of this test was to evaluate if the modified SmartCGMS framework can load the developed filters, build a filter chain, and process events with the same results as the original SmartCGMS. This test also aimed to verify that the implemented filters behave as intended.

5.1.1 Experimental Setup

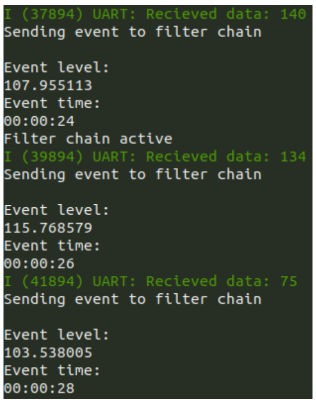
The experimental setup includes an array of 100 double-precision floating-point values. The values were pseudo-randomly generated and do not represent any real data. The same array was used on all platforms. These values were then used to create level events and were processed by the SmartCGMS concept applications. The filter chain processing the events was the same for all platforms. It includes these filters in the specified order:

- Data reading filter
- Data transformation filter
- Data visualization filter

First, the filters were compiled as Microsoft Windows dynamic-link libraries for the original SmartCGMS. The filters' source code files were then preprocessed by the implemented filter preprocessor. The modified source code files of the filters were then compiled for the SmartCGMS concept platforms.

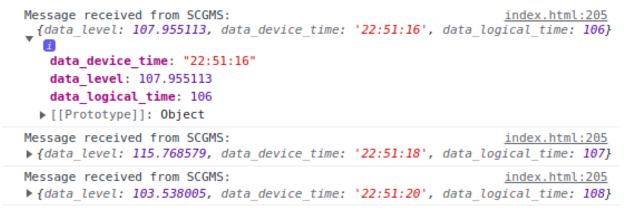
The configuration remained identical for all platforms except for the `create_task` parameter of the data reading filter. On the ESP32, a task to listen for incoming data on UART was created. The data from the array was then transmitted to the ESP32 using UART. The other platforms used the provided SmartCGMS concept API to create events directly in the same task/thread that built the filter chain. The results were collected using the data visualization filter. After each execution of the test scenario, the printed event levels were saved to a file. The same test scenario was carried out using the SmartCGMS console application on Microsoft Windows 10. After all the results were gathered, the events' level values were compared.

The watchdog filter has been tested separately. The watchdog has been implemented to indicate that the filter chain is inactive. To test its correct functionality a filter chain was built but no events were sent into the chain. After the timeout period, the watchdog was expected to trigger. For the test, the `Trigger` function has been



```
I (37894) UART: Recieved data: 140
Sending event to filter chain

Event level:
107.955113
Event time:
00:00:24
Filter chain active
I (39894) UART: Recieved data: 134
Sending event to filter chain

Event level:
115.768579
Event time:
00:00:26
I (41894) UART: Recieved data: 75
Sending event to filter chain

Event level:
103.538005
Event time:
00:00:28
```

Figure 5.1: ESP32



```
Message received from SCGMS:                                                index.html:205
 {data_level: 107.955113, data_device_time: '22:51:16', data_logical_time: 106}
▼ i
    data_device_time: "22:51:16"
    data_level: 107.955113
    data_logical_time: 106
  ▶ [[Prototype]]: Object
Message received from SCGMS:                                                index.html:205
▶ {data_level: 115.768579, data_device_time: '22:51:18', data_logical_time: 107}
Message received from SCGMS:                                                index.html:205
▶ {data_level: 103.538005, data_device_time: '22:51:20', data_logical_time: 108}
```

Figure 5.2: The SmartCGMS webpage

Figure 5.3: The event level values being collected on different platforms

implemented as a print statement to the console, signaling the chain is inactive. The test scenario was executed, and the console was observed to see whether the watchdog was triggered or not. This test has been carried out only on the SmartCGMS concept platforms.

5.1.2 Results and Discussion

The events' level values were printed with a precision of six decimal places. The level values were identical on all platforms, including the original SmartCGMS. This suggests that the modifications to the SmartCGMS codebase didn't influence its functionality. Additionally, the performed tests indicate that the proposed and implemented filter preprocessor functions correctly as it was used during the setup for the tests. One of the biggest limitations of the SmartCGMS concept is its lack of filesystem support. Multiple filters that are developed for the original SmartCGMS use files. Implementing a simple filesystem for the SmartCGMS concept-supported platforms would allow us to bring the concept even closer to the original SmartCGMS. Emscripten provides a virtual filesystem that allows us to use more features from the original SmartCGMS (e.g., deferring model parameters to files), see section 4.5.4

The filters' behavior (described in section 4.4) verification was also successful. The data filter's task successfully received/generated the data on platforms that used the `create_task` parameter set to `true` (ESP32 and Raspberry Pi Zero W). The correct calculation of the EMA by the data transformation filter has been presented by the modified level values printed to the console. The data visualization filter's functionality was verified by collecting the level values from the console. The watchdog filter was successfully triggered on all SmartCGMS concept platforms except WAMR. The reason was that WAMR support for pthreads is only experimental, and attempts to create a separate pthread for the watchdog timer were unsuccessful. The SmartCGMS web demo requires the browser to support `SharedArrayBuffer` for correct functionality of the watchdog timer thread.

SmartCGMS defines multiple kinds of entities that can be used within the framework. The SmartCGMS concept focused on SmartCGMS filters as the top-level entity. Support for other entities was not implemented in the SmartCGMS concept and, therefore, was not tested. The gathered results suggest that the SmartCGMS concept is capable of building a filter chain (potentially with asynchronous filters) and executing events on all supported platforms. Only the most basic functionality of the SmartCGMS concept has been tested. Thorough testing will be required along with further development of the concept.

5.2 Memory Usage

The memory usage of an application for low-power devices is an important characteristic. Low-power devices often have a limited amount of Random-access memory (RAM). When writing an application for a low-power device, it is important to keep the memory requirements as low as possible. On the other hand, predictability is also an important factor, and it may be preferable to have an application with a constant higher memory consumption than an application with very low memory requirements that has sudden peaks of memory usage. In this test, the memory footprint of the SmartCGMS concept application is measured. This test has two objectives. The first objective is to provide insight into how the SmartCGMS concept application manages memory allocations during runtime. The second objective of the test is to compare the SmartCGMS concept's memory footprint compiled as a native application and a WASM application.

5.2.1 Experimental Setup

The memory consumption has been measured on four different platforms:

- Raspberry Pi Zero W running FreeRTOS (ARMv6)
- ESP32 running FreeRTOS (Xtensa)
- GNU/Linux (x86-64)
- WAMR (running on GNU/Linux (x86-64))

The first two platforms were used to gather data about how much memory the SmartCGMS concept allocates when building a filter chain and processing events.

The last two platforms were used to compare the memory footprint of the native and WASM applications.

The same array of event levels from the test scenario in section 5.1 was used. All four implemented filters were used. In the following order:

- #1 Data reading filter
- #2 Data transformation filter
- #3 Data visualization filter
- #4 Watchdog filter

A FreeRTOS `xPortGetFreeHeapSize` API was used to gather the dynamic memory usage on the Raspberry Pi Zero W. The ESP-IDF provides its own API for requesting heap information `heap_caps_print_heap_info`. Both of these functions provide information about the current free heap size. The data from these functions were collected before and after building the filter chain. Then, the heap information was sampled for each second during the execution of the test scenario. The events were injected every two seconds into the filter chain.

To compare the native and WASM applications, GNU/Linux was used. Originally, the goal was to use ESP32 for the comparison, but the compiled SmartCGMS WASM module was too large for its limited RAM. On the GNU/Linux platforms, both versions of the SmartCGMS concept application were measured using the same tools. For measuring the heap memory usage, the Valgrind Massif heap profiler tool was used. Valgrind measures how much memory the program allocates from the heap [53]. The static memory footprint of the SmartCGMS concept application was measured using the `size` GNU utility. The `size` tool displays the sizes of each section and the total size of a specified object [54]. The total memory footprint of the application was then computed as a sum of the total size of the executable and the maximum heap usage. For the WASM version, both the size of the WASM module and the WASM runtime were summed. To make the comparison as fair as possible, the SmartCGMS concept application has been compiled using the same build environment. The only difference was that the WASM module was compiled using WASISDK cross-compile toolchain file. The cross-compile toolchain file specifies the correct compiler, linker, and additional tools required for the compilation to WASM.

5.2.2 Results and Discussion

Figure 5.4 presents the allocated heap space during the SmartCGMS concept application runtime on the Raspberry Pi Zero W and ESP32 platforms. The actual value of the allocated heap space is not relevant. Rather, the focus is on the shape of the graph lines. The concrete values can change depending on the specific parameters of the filters (e.g., the watchdog task's stack size) and their implementation. From the graph, it can be observed that once the filter chain is built and the first event is injected into the filter chain, the allocated heap space does not change. The constant memory usage is desired on embedded devices. It is achieved by the SmartCGMS framework's device event pool. SmartCGMS implements its own statically allocated pool of device events. When an event is created, SmartCGMS first tries to create it from the event pool, and only if it fails will it allocate the event on the heap.

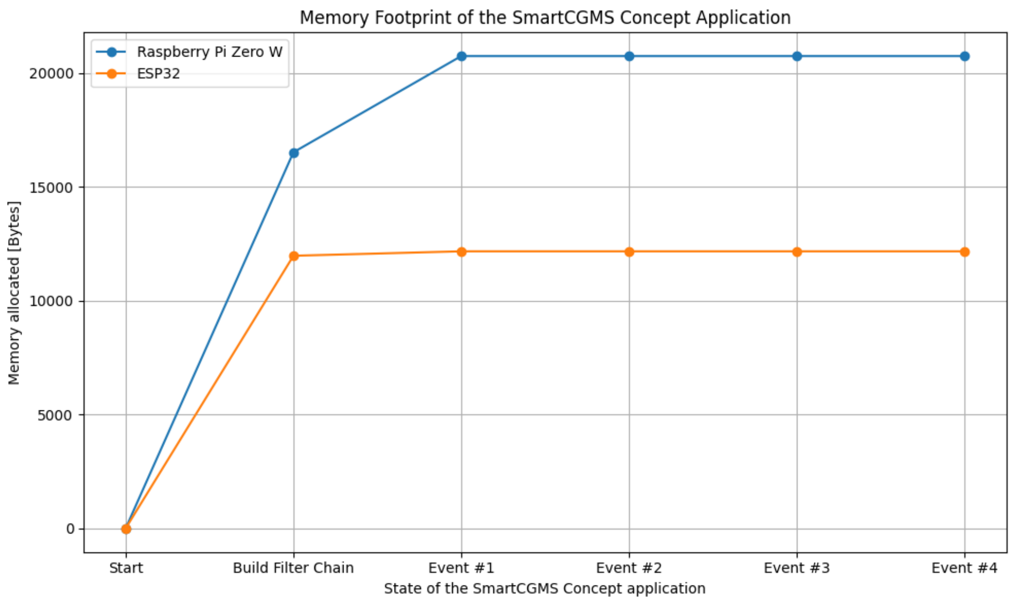


Figure 5.4: Dynamic memory usage of SmartCGMS concept application executed as a FreeRTOS native application on Raspberry Pi Zero W and ESP32. Only the first four events are displayed since the allocated memory remains constant afterward.

The initial rise of allocated memory is caused by the construction of the filter chain. The SmartCGMS concept applications are using dynamic task allocation. The tasks' stacks are therefore allocated from the heap and this causes a relatively big increase in memory usage. While static allocation is generally preferred for programming low-power devices due to its predictability, dynamic allocations occur only during the initialization phase of the application - when configuring the filters in the filter chain. Ensuring the application shuts down safely in case of insufficient memory during the filter chain configuration is relatively easy to control. Compared to encountering a depletion of free heap space during event processing. Regardless, there are not any constraints that would prevent allocating the tasks' stacks statically (FreeRTOS offers the `xTaskCreateStatic` API).

The figure 5.5 displays the dynamic memory usage of the SmartCGMS concept native application on GNU/Linux. The WASM module's memory usage is presented in figure 5.6. The maximum values are presented in table 5.1. As the goal was to compare the memory requirements of the two approaches, the exact values are relevant. The heap memory usage of the WASM version of the SmartCGMS concept application is approximately 85x higher than the usage of the native version. The main reason is that the WASM standalone runtime must load the whole WASM module into a buffer, and then additional memory is required to load the module sections and instantiate the module as the operations are not performed in place. The native application's memory usage consists mainly of allocations performed by the standard C++ library.

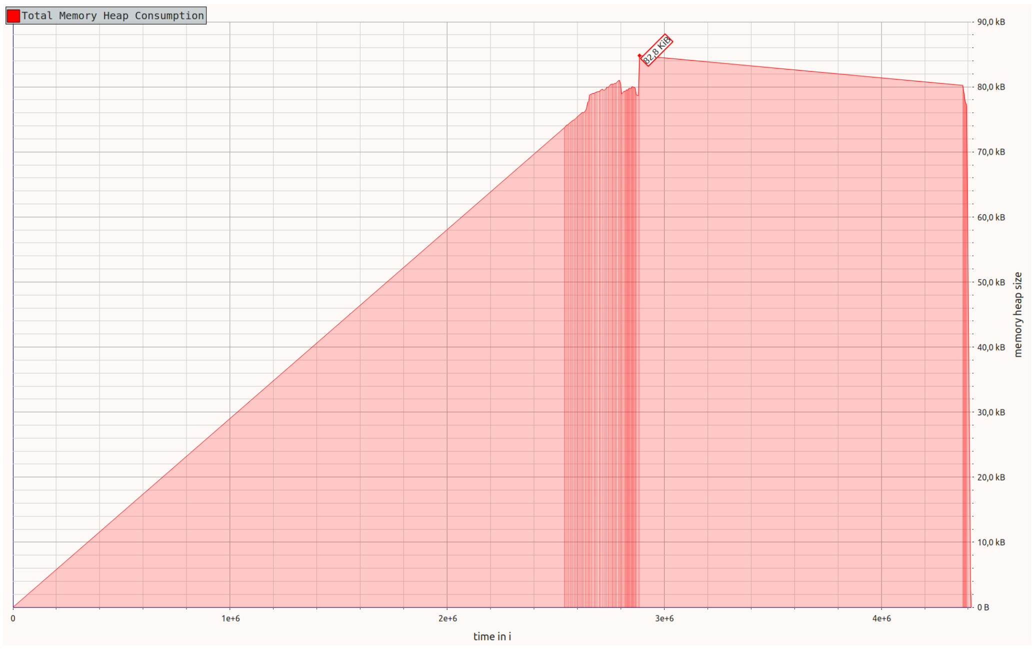


Figure 5.5: Dynamic memory usage of SmartCGMS concept application executed on GNU/Linux as a native application, only the total memory consumption is displayed. The time displayed on the x-axis is measured in executed instructions.

When comparing the static memory footprint of the two versions, the native SmartCGMS application has a smaller footprint. The specific values are presented in table 5.1. The footprint of the WASM module and the WAMR application combined is almost twice as big as the footprint of the native application. When comparing the total memory footprint of the two approaches (table 5.1), the SmartCGMS native application's memory footprint is approximately 9x lower than that of the WASM version. The results for the WASM module were gathered from WAMR in the fast interpreter mode.

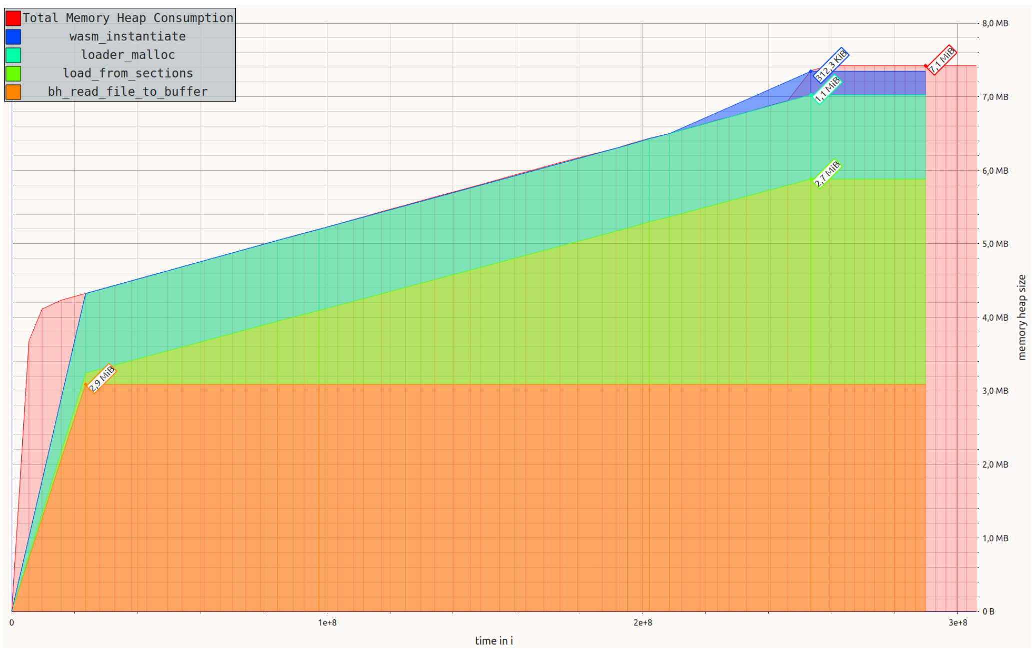


Figure 5.6: Dynamic memory usage of SmartCGMS concept application executed as a WASM module in WAMR on GNU/Linux. The time displayed on the x-axis is measured in executed instructions.

Table 5.1: Memory Footprint of the SmartCGMS Concept Application

	WASM	**Native**
Executable Size	1223KB	880KB
Runtime Size	360KB	-
Maximum Heap Usage	7085KB	83KB
Total Size	8668KB	963KB

WAMR also offers AOT mode that requires the WASM code to be compiled for a specific architecture before it can be loaded and executed. This mode offers increased execution speed as the Just-In-Time (JIT) LLVM mode but without the added memory overhead of the LLVM compilation during runtime. This mode appears to be the best fit for the anticipated usage of the SmartCGMS concept, but attempts to compile and run it on the GNU/Linux platform were unsuccessful.

As mentioned earlier, the WASM memory footprint of the current approach rules out low-power devices with less than 10 MB of RAM (e.g., ESP32). The WASM module size could be further reduced with code optimizations and different compile options, but without a major redesign of the SmartCGMS Concept, it is impossible to fit within the memory limit of ESP32 (320 KB of Data RAM).

The WASM module was linked with the C/C++ standard library. When compiled with the option `-nostdlib` (and then using the host environment's C/C++ `stdlib` implementation), the size of the executable is reduced to 168KB. However, from observing the memory allocation of WAMR in figure 5.6, the ESP32 would still be unable to execute the SmartCGMS concept WASM module. WAMR allocates at least twice the memory of the WASM module when loading and instantiating it. Alternative runtime targeted at memory-constrained devices (such as the one proposed in [55]) might allow execution of the SmartCGMS concept application compiled to WASM on the ESP32. The Raspberry Pi Zero W is a viable candidate for the WASM version of the SmartCGMS concept application as it has sufficient RAM (512 MB). At the writing of the project, a ready-to-execute WASM standalone runtime was not available for the Raspberry Pi Zero W, and creating a custom port of available WASM runtimes was out of the scope of the project. Therefore, it is only a suggestion for further research.

5.3 Execution Time

Execution time might not appear as a critical parameter when considering the use case of the SmartCGMS concept application as an insulin pump controller. However, it is directly related to energy consumption. The more time the device spends processing events, the more energy it consumes. Given that WASM can run on embedded devices, a test scenario was created to compare the SmartCGMS concept application compiled to WASM for WAMR and a native application in terms of the time required to process a specific number of events. This test aims to provide more insight into the WASM modules versus native applications debate and help steer the future development of the SmartCGMS concept.

5.3.1 Experimental Setup

The execution speed of the SmartCGMS concept application was evaluated on GNU/Linux. A native SmartCGMS concept application and WASM module for WAMR were compared. The tested application consisted of building a filter chain and then executing a specified number of events. The events were created using the level values from the array from previous experiments. The filter chain included these filters:

- #1 Data reading filter
- #2 Data transformation filter
- #3 Watchdog filter

The data visualization filter was not used in this evaluation as the I/O operations would probably be the main bottleneck of the execution speed. To obtain accurate results, the event set was iterated multiple times. Different numbers of iterations through the set were measured: 1000, 10 000, and 100 000. The native SmartCGMS concept application was compared to the WASM application running in WAMR in the fast interpreter mode and the LLVM JIT mode. Each test was performed 10 times, and the times were averaged. Multiple runs of the test should ensure better accuracy of the data by reducing the influence of random variations caused by, for example, cache, I/O operations, memory management, or background processes. The execution time was measured using the `clock_gettime` function from the GNU C library, defined in `time.h` header. The function takes two parameters. The ID of the clock that it should obtain the time from and a pointer to a `timespec` struct, in which the current time of the clock is returned. A high-resolution clock (`CLOCK_PROCESS_CPUTIME_ID`) was used for measuring the execution time of the tests. The time required for the execution was computed by subtracting two timestamps. In the native application, the initial timestamp was recorded before building the filter chain, and the ending timestamp was recorded after all the events were processed. The WASM modules execution time was measured from WAMR. The initial timestamp was recorded before calling the main function of the WASM module, and the ending timestamp was recorded right after the main function returned. The main function of the WASM module was identical to the main function of the native application and consisted only of building the chain and then executing the defined set of events.

5.3.2 Results and Discussion

The averaged results are displayed in Figure 5.7, with precise numeric values provided in Table 5.2. The findings are similar to the ones published in [56]. The execution is approximately 10 times slower in WAMR fast interpreter mode compared to the native application. The JIT mode performs the best when the WASM module performs a large number of operations. The disadvantage of JIT mode is that the initialization of the WASM module includes its compilation for the target architecture. This creates a large memory and execution time overhead when the module is loaded. From the table 5.2, it can be seen that the execution time of the WASM module executed in JIT mode increased in similar increment to the native approach between the two largest number of iterations. The WAMR Ahead-Of-Time (AOT) mode could be used as an alternative to the JIT mode in the SmartCGMS concept. It has the advantage of faster execution with platform-specific code and the compilation overhead does not affect the execution as it is compiled AOT.

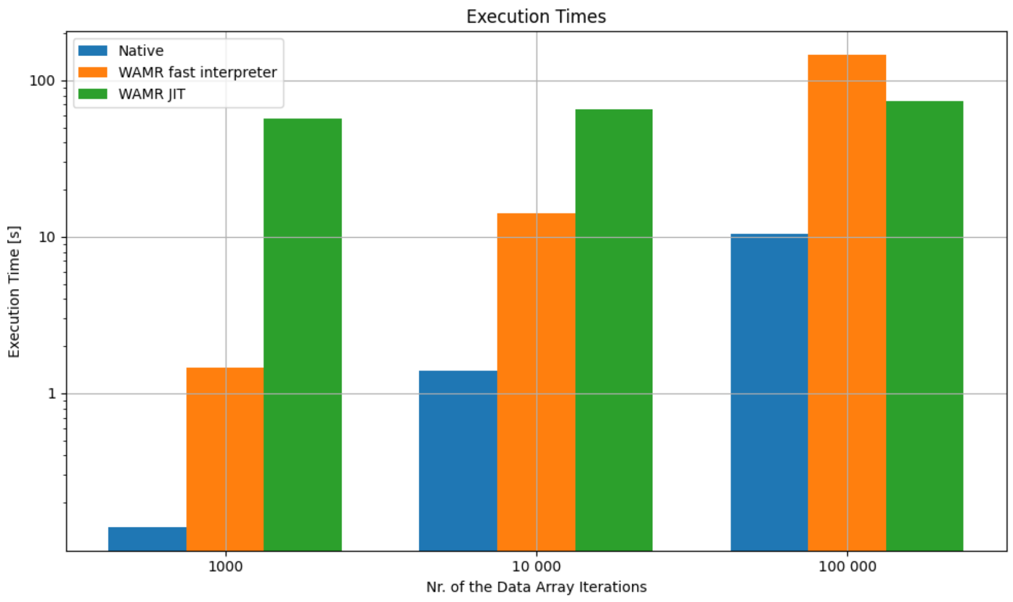


Figure 5.7: Comparison of the SmartCGMS concept application execution times on different platforms: Native GNU/Linux, WAMR fast interpreter, and WAMR JIT mode. The y-axis displays time on a logarithmic scale.

Table 5.2: Execution Times of the SmartCGMS concept application on different platforms (in seconds). The top line refers to the number of processed events by the filter chain.

Execution Type	10^5 events	10^6 events	10^7 events
Native Application	0.14	1.41	10.38
WAMR Fast Interprer	1.39	14.32	145.54
WAMR JIT Mode	57.67	65.87	74.94

Conclusion and Future Work

6

The goal of this thesis was to propose and implement a SmartCGMS concept and evaluate its execution on low-power devices. The main requirements were to use FreeRTOS and retain compatibility with the x86-64 Instruction Set Architecture (ISA).

The proposed SmartCGMS concept uses static linking instead of dynamic linking. To complement the changes in the SmartCGMS codebase, a preprocessor tool was implemented that automatically generates source code files to overcome issues introduced by changing the linking strategy. Static linking has been chosen for its ease of portability across low-power platforms. The work results in a prototype SmartCGMS concept for low-power devices. The implementation was successfully executed on five different platforms: ESP32, Raspberry Pi Zero W, GNU/Linux, WebAssembly Micro Runtime (WAMR), and as aWebAssembly (WASM) module embedded in a webpage.

The evaluation focused on the correct functionality of the SmartCGMS concept. Four filters were implemented to compare the original SmartCGMS and the SmartCGMS concept. The results were gathered by executing the same test scenario in the original SmartCGMS framework and on all the SmartCGMS concept supported platforms. The verification was successful, the results from the SmartCGMS concept applications were the same as from the original SmartCGMS. Furthermore, the evaluation included a comparison of the execution and memory overhead between the SmartCGMS concept compiled as a native application and as a WASM module. This comparison not only assessed the performance of the WASM and native approaches for the SmartCGMS framework on low-power devices but also highlighted the need for low overhead WASM runtimes for Internet of Things (IoT) devices.

This research has proposed and implemented two approaches for executing SmartCGMS on low-power devices. Both solutions have their advantages and disadvantages. For future SmartCGMS research, it is important to focus on evaluating WASM runtimes regarding their memory requirements.

Bibliography

[1] *Medtronic MiniMed 670G*. URL: https://www.medtronic.com/ca-en/diabetes/home/products/insulin-pumps/minimed-670g.html. (Accessed: 2024-04-24).

[2] *OpenAPS*. URL: https://openaps.org/. (Accessed: 2024-04-24).

[3] *AndroidAPS*. URL: https://androidaps.readthedocs.io/. (Accessed: 2024-04-24).

[4] *Loop*. URL: https://loopkit.github.io/loopdocs/. (Accessed: 2024-04-24).

[5] T. E. Group. *Diabetes Mellitus Metabolic Simulator for Research (DMMS.R)*. URL: https://tegvirginia.com/dmms-r/. (Accessed: 2024-04-24).

[6] M. Ubl, T. Koutny, A. Della Cioppa, I. De Falco, E. Tarantino, and U. Scafuri. "Distributed Assessment of Virtual Insulin-Pump Settings Using SmartCGMS and DMMS.R for Diabetes Treatment". In: *Sensors* 22.23 (2022). ISSN: 1424-8220. URL: https://www.mdpi.com/1424-8220/22/23/9445.

[7] L. Petruzelkova, J. Soupal, V. Plasova, P. Jiranova, V. Neuman, L. Plachy, S. Pruhova, Z. Sumnik, and B. Obermannova. "Excellent Glycemic Control Maintained by Open-Source Hybrid Closed-Loop AndroidAPS During and After Sustained Physical Activity". In: *Diabetes Technology & Therapeutics* 20.11 (2018), pp. 744–750.

[8] D. M. Lewis, R. S. Swain, and T. W. Donner. "Improvements in A1C and time-in-range in DIY closed-loop (OpenAPS) users". In: *Diabetes* 67.Supplement_1 (2018).

[9] J. Künzler, T. Züger, C. Stettler, M. Laimer, and A. Melmer. "Comparing the technical reliability and insulin dosing of a" do-it-yourself artificial pancreas" with a commercial hybrid closed-loop system in a" shadow-mode" scenario: An exploratory study." In: *Diabetes, obesity & metabolism* (2023).

[10] *OpenAPS determine-basal.js*. URL: https://github.com/openaps/oref0/blob/master/lib/determine-basal/determine-basal.js. (Accessed: 2024-05-15).

[11] A. Melmer, T. Züger, D. M. Lewis, S. Leibrand, C. Stettler, and M. Laimer. "Glycaemic control in individuals with type 1 diabetes using an open source artificial pancreas system (OpenAPS)". In: *Diabetes, Obesity and Metabolism* 21.10 (2019), pp. 2333–2337.

[12] D. C. Klonoff. "Cybersecurity for connected diabetes devices". In: *Journal of diabetes science and technology* 9.5 (2015), pp. 1143–1147.

[13] M. E. Horowitz, W. A. Kaye, G. M. Pepper, K. E. Reynolds, S. R. Patel, K. C. Knudson, G. K. Kale, M. E. Gutierrez, L. A. Cotto, and B. S. Horowitz. "An analysis of Medtronic MiniMed 670G insulin pump use in clinical practice and the impact on glycemic control, quality of life, and compliance". In: *Diabetes Research and Clinical Practice* 177 (2021), p. 108876. ISSN: 0168-8227. DOI: `https://doi.org/10.1016/j.diabres.2021.108876`. URL: `https://www.sciencedirect.com/science/article/pii/S0168822721002357`.

[14] L. H. M. Aria Saunders and G. P. Forlenza. "MiniMed 670G hybrid closed loop artificial pancreas system for the treatment of type 1 diabetes mellitus: overview of its safety and efficacy". In: *Expert Review of Medical Devices* 16.10 (2019), pp. 845–853.

[15] T. Knebel and J. J. Neumiller. "Medtronic MiniMed 670G Hybrid Closed-Loop System". In: *Clinical Diabetes* 37.1 (Jan. 2019), pp. 94–95. ISSN: 0891-8929. DOI: `10.2337/cd18-0067`. eprint: `https://diabetesjournals.org/clinical/article-pdf/37/1/94/501043/94.pdf`. URL: `https://doi.org/10.2337/cd18-0067`.

[16] F. Kuhl, R. Weatherly, and J. Dahmann. *Creating computer simulation systems: an introduction to the high level architecture*. Prentice Hall PTR, 1999.

[17] T. Koutny and M. Ubl. "SmartCGMS as a Testbed for a Blood-Glucose Level Prediction and/or Control Challenge with (an FDA-Accepted) Diabetic Patient Simulation". In: *Procedia Computer Science* 177 (2020). The 11th International Conference on Emerging Ubiquitous Systems and Pervasive Networks (EUSPN 2020) / The 10th International Conference on Current and Future Trends of Information and Communication Technologies in Healthcare (ICTH 2020) / Affiliated Workshops, pp. 354–362. ISSN: 1877-0509. DOI: `https://doi.org/10.1016/j.procs.2020.10.048`. URL: `https://www.sciencedirect.com/science/article/pii/S1877050920323164`.

[18] M. Otta. "Towards a health software supporting platform for wearable devices". In: *Procedia Computer Science* 210 (2022), pp. 112–115. ISSN: 1877-0509. DOI: `https://doi.org/10.1016/j.procs.2022.10.126`. URL: `https://www.sciencedirect.com/science/article/pii/S1877050922015836`.

[19] Bytecode Alliance. *WebAssembly micro runtime*. URL: https://github.com/bytecodealliance/wasm-micro-runtime. (Accessed: 2024-02-22).

[20] *Wasm3*. URL: https://github.com/wasm3/wasm3. (Accessed: 2024-02-22).

[21] *WebAssembly Core Specification*. Version 2.0. W3C, Apr. 19, 2022. URL: https://www.w3.org/TR/wasm-core-2/.

[22] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien. "Bringing the web up to speed with WebAssembly". In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2017, pp. 185–200.

[23] E. Wen and G. Weber. "Wasmachine: Bring iot up to speed with a webassembly os". In: *2020 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*. IEEE. 2020, pp. 1–4.

[24] N. Mäkitalo, T. Mikkonen, C. Pautasso, V. Bankowski, P. Daubaris, R. Mikkola, and O. Beletski. "WebAssembly modules as lightweight containers for liquid IoT applications". In: *International Conference on Web Engineering*. Springer. 2021, pp. 328–336.

[25] P. P. Ray. "An Overview of WebAssembly for IoT: Background, Tools, State-of-the-Art, Challenges, and Future Directions". In: *Future Internet* 15.8 (2023), p. 275.

[26] P. Kocian. *Dynamic Software Image Update on ESP32*. URL: https://github.com/PetrKocian/ESP32-WASM-Dynamic-Update/. (Accessed: 2024-04-24).

[27] *diabetes.zcu.cz*. URL: https://diabetes.zcu.cz/. (accessed: 2022-02-23).

[28] R. N. Bergman. "Toward physiological understanding of glucose tolerance: minimal-model approach". In: *Diabetes* 38.12 (1989), pp. 1512–1527.

[29] T. Koutny and M. Ubl. "Parallel software architecture for the next generation of glucose monitoring". In: *Procedia Computer Science* 141 (2018), pp. 279–286. ISSN: 1877-0509. DOI: https://doi.org/10.1016/j.procs.2018.10.197. URL: https://www.sciencedirect.com/science/article/pii/S1877050918318507.

[30] M. Nanda, S. Dhage, and J. Jayanthi. "An approach to formally qualify commercial RTOS for safety application". In: *2015 2nd International Conference on Computing for Sustainable Global Development (INDIACom)*. 2015, pp. 816–821.

[31] *SAFERTOS*. URL: https://www.highintegritysystems.com/safertos/. (accessed: 2022-02-24).

[32] *FreeRTOS*. URL: http://www.freertos.org. (accessed: 24.02.2024).

[33] *OpenRTOS*. URL: https://www.highintegritysystems.com/openrtos/. (accessed: 2022-02-24).

[34] E. Systems. *ESP-IDF: Espressif IoT Development Framework*. Espressif Systems. URL: `https://github.com/espressif/esp-idf`.

[35] *The Open Group Base Specifications Issue 7, IEEE Std 1003.1.* URL: `https://pubs.opengroup.org/onlinepubs/9699919799/basedefs/pthread.h.html`. (Accessed: 2024-03-22).

[36] *FreeRTOS-Plus-POSIX*. URL: `https://www.freertos.org/FreeRTOS-Plus/FreeRTOS_Plus_POSIX/index.html`. (accessed: 23.03.2024).

[37] *Standardizing WASI: A system interface to run WebAssembly outside the web*. URL: `https://hacks.mozilla.org/2019/03/standardizing-wasi-a-webassembly-system-interface/`. (Accessed: 2024-02-22).

[38] *WebAssembly*. Accessed: 2022-02-24. 2024. URL: `https://webassembly.org/`.

[39] *Emscripten*. URL: `https://emscripten.org/`. (Accessed: 2024-02-22).

[40] A. Zakai. "Emscripten: an LLVM-to-JavaScript compiler". In: New York, NY, USA: Association for Computing Machinery, 2011. ISBN: 9781450309424. DOI: `10.1145/2048147.2048224`. URL: `https://doi.org/10.1145/2048147.2048224`.

[41] *WASI-SDK*. URL: `https://github.com/WebAssembly/wasi-sdk`. (Accessed: 2024-02-22).

[42] *Wasmer*. URL: `https://github.com/wasmerio/wasmer`. (Accessed: 2024-03-22).

[43] *Wasmtime*. URL: `https://github.com/bytecodealliance/wasmtime`. (Accessed: 2024-03-22).

[44] *wazero*. URL: `https://github.com/tetratelabs/wazero`. (Accessed: 2024-03-22).

[45] *WasmEdge*. URL: `https://github.com/WasmEdge/WasmEdge`. (Accessed: 2024-03-22).

[46] Bytecode Alliance. *Introduction to WAMR running modes*. 2023. URL: `https://bytecodealliance.github.io/wamr.dev/blog/introduction-to-wamr-running-modes/`. (Accessed: 2024-04-24).

[47] Espressif Systems. *ESP32 Overviewl*. Accessed: 2022-01-24. 2024. URL: `https://www.espressif.com/en/products/socs/esp32`.

[48] A. Maier, A. Sharp, and Y. Vagapov. "Comparative analysis and practical implementation of the ESP32 microcontroller module for the internet of things". In: *2017 Internet Technologies and Applications (ITA)*. 2017, pp. 143–148. DOI: `10.1109/ITECHA.2017.8101926`.

[49] *RaspberryPi*. Accessed: 2022-01-24. 2024. URL: `https://www.raspberrypi.com`.

[50] *Arm Documentation*. section: ISO C library implementation definition. URL: `https://developer.arm.com/documentation`. (Accessed: 2024-04-04).

[51] M. Rashid, S. Samadi, M. Sevil, I. Hajizadeh, P. Kolodziej, N. Hobbs, Z. Maloney, R. Brandt, J. Feng, M. Park, et al. "Simulation software for assessment of nonlinear and adaptive multivariable control algorithms: glucose–insulin dynamics in type 1 diabetes". In: *Computers & Chemical Engineering* 130 (2019), p. 106565.

[52] C. D. Man, F. Micheletto, D. Lv, M. Breton, B. Kovatchev, and C. Cobelli. "The UVA/PADOVA Type 1 Diabetes Simulator: New Features". In: *Journal of Diabetes Science and Technology* 8.1 (2014). PMID: 24876534, pp. 26–34. DOI: `10.1177/1932296813514502`. URL: `https://doi.org/10.1177/1932296813514502`.

[53] *Valgrind Documentation*. Massif: a heap profiler. URL: `https://valgrind.org/docs/manual/ms-manual.html`. (Accessed: 2024-04-12).

[54] *Linux manual*. page: size(1). URL: `https://man7.org/linux/man-pages/man1/size.1.html`. (Accessed: 2024-04-12).

[55] B. Li, H. Fan, Y. Gao, and W. Dong. "Bringing WebAssembly to Resource-constrained IoT Devices for Seamless Device-Cloud Integration". In: *Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services*. 2022, pp. 261–272.

[56] W. Wang. "How Far We've Come – A Characterization Study of Standalone WebAssembly Runtimes". In: *2022 IEEE International Symposium on Workload Characterization (IISWC)*. 2022, pp. 228–241. DOI: `10.1109/IISWC55918.2022.00028`.

User Manual

This appendix chapter includes details on how to use and extend the developed SmartCGMS concept. The project is split into three repositories:

- SmartCGMS Embedded Codebase
- Build Environments
- Build System

The SmartCGMS Embedded Codebase repository contains the current modified codebase of SmartCGMS. When the concept is officially adopted by SmartCGMS, this repository will be replaced by the official SmartCGMS repositories `core` and `common`.

The Build Environments repository contains files and directories used to build the SmartCGMS concept for all supported platforms. Each platform has its directory in the root directory of the repository. Each currently supported platform uses CMake to simplify the build process. Each platform provides a `build` script that generates the compiled binary. Some platforms contain additional `run` script that uploads the binary to the platform. Directories of low-power supported platforms contain the source code of FreeRTOS and required drivers.

The Build System repository is targeted at GNU/Linux. It contains a Python script to pull the SmartCGMS code and the build environment for a specified platform from the repositories. It also contains the filter preprocessor tool. When executing the Python script, the filter preprocessor tool is used to modify the source code files of filters in the `input` folder. The repository contains a Dockerfile to build an image that can be used to execute the Python script.

How to Develop a SmartCGMS Concept Application

The build system and the build environments are targeted at `GNU/Linux`. It is therefore recommended to use `GNU/Linux` when building the SmartCGMS concept application.

Toolchains

The dependencies differ for each platform. The table 1 presents the required tools and toolchains for each platform.

Table 1: Tools required by the SmartCGMS concept

Tool	Required by
Git	The Build System
Docker	The Build System
CMake	All platforms
ESP-IDF	ESP32
ARMv6 Cross-compiler	RasperryPi Zero W
Emscripten	WASM targeted at the web
WASI SDK	WASM for WAMR

Developing Filters

The SmartCGMS concept supports only the filters from the SmartCGMS entities. To develop a custom application using the SmartCGMS concept, the user must first implement custom filters for their desired platforms. To develop a filter, follow the official SmartCGMS documentation. There are four experimental filters provided with the SmartCGMS concept. They are described in the chapter 4.

Cloning the Build System Repository

Once the filters are implemented. The next step in creating the SmartCGMS concept application is to clone the Build System repository. To do so, the user can use the command displayed in listing 6.1. The Build System is also provided in the folder `Aplication_and_libraries/Build_System_Linux/build_script`.

Listing 6.1: Cloning the Build System repository

```
petrk@pc:$ git clone https://github.com/PetrKocian/SCGMS-Build-System.git
```

Creating a Folder for a Supported Platform

The Build System repository contains the filter preprocessor tool and a Python script to create build directories for supported platforms.

The build directory-generating script can be used in the provided Docker image or the local GNU/Linux environment. The dependencies for the build script are defined in the Dockerfile. To use the Docker image, it has to be built first. It can be done by running the `build_docker.sh` script or executing directly the command from listing 6.2:

Listing 6.2: Building a Docker image for executing the Python build script

```
petrk@pc:$ docker build docker -t scgms-builder
```

After the Docker image is built, the supported platforms can be displayed by executing the `run_docker.sh` script without any parameters. To generate a build directory for a desired platform, invoke the script with the name of the platform as the only parameter. The names of the supported platforms are:

- esp32
- rpizerow
- wasm_emcc
- wasm_wamr

The directories for all platforms can be generated at once by providing the parameter `all` (seen in listing 6.3).

Listing 6.3: Running the Python build script inside using provided Docker script.

```
petrk@pc:$ ./run_docker.sh all
```

The Python script `paths_to_toolchains.py` contains the paths to required toolchains used by individual platforms. These paths are used to generate CMake files or shell scripts that are then used by the platforms' build scripts. If the installation location of the toolchains differs for the user, these paths should be edited directly in the script to ensure the correct generation of the toolchain files in the platform build directories.

```
# PATHS TO TOOLCHAINS
esp_idf_path = "~/esp/esp-idf"
wasi_path = "/opt/wasi-sdk"
emsdk_path = "~/emsdk"
```

The script also invokes the filter preprocessor. The original filter source code files are taken from the `input` directory. The filter preprocessor then generates files that are ready for compilation in the platform directory.

The script does a clean build whenever its invoked - the directory where the platform build files are generated is deleted and then generated again. It is therefore important to copy the generated folder to a different location or rename it, in case it should not be deleted.

Alternative Approach without Docker

The Python build script can be invoked directly. To use it, it is necessary to install:

- Python
- GitPython

The script can then be called directly. See listing 6.4.

Listing 6.4: Invokind Python build script directly.

```
petrk@pc:$ python prepare_build.py all
```

Filter Preprocessor

The filter preprocessor can be used as a standalone application. To preprocess the filters, copy their source code files to `input` directory where the filter preprocessor executable is located. Then run the `preprocessor` executable. If the preprocessing is successful, the modified filter files are generated to `temp` directory.

Building the SmartCGMS Concept Executable

Once the build directory for a platform is generated, it should be possible to build an executable for the target platform. Each platform build directory contains a shell script `build.sh`. If the toolchain paths have been specified correctly in the previous step, the `build.sh` should compile the SmartCGMS concept executable to a `build` directory.

Executing the SmartCGMS Concept Application

Raspberry Pi Zero W

To execute the SmartCGMS concept application on Raspberry Pi Zero W, ensure that the Raspberry Pi has a SREC bootloader on its microSD card. Once the executable is ready, restart the Raspberry Pi Zero W (e.g., unplug and plug back in power) and execute the `start.sh` script. The script uploads the `rpiscgms.srec` SREC image to the Raspberry Pi Zero W over UART and then starts listening on UART. The upload is slow - approximately 5 minutes - but it can be uploaded without using a microSD card reader.

ESP32

To execute the prepared SmartCGMS concept application on the ESP32, the user can use the `start.sh` shell script. The project uses the Espressif IoT Development Framework (ESP-IDF). To use the `idf.py` command to interact with the project, the user can export the required variables and use the command directly, instead of using the provided shell scripts. The ESP32 application listens on UART for incoming data. A UART sender application is provided in folder `UART_Event_Sender` that periodically sends data to the UART. It can be executed by running the `run.sh` script.

WASM for WAMR

Once the SmartCGMS concept application for WAMR is built, it can be executed using the `iwasm` executable binary. WAMR provides a guide on how to build the `iwasm` binary on its GitHub page. The SmartCGMS concept application WASM module can then be executed by using the command in listing 6.5.

Listing 6.5: Executing SmartCGMS concept application WASM module using `iwasm`.

```
petrk@pc:$ ./iwasm scgms
```

GNU/Linux

To compile the SmartCGMS concept application for GNU/Linux, use the WASM for WAMR environment. Do not use the provided `build.sh` script, but instead use this series of commands:

```
rm -rf build
mkdir build
cd build
cmake ..
make
```

WASM for the Web

To execute the WASM module compiled using Emscripten, a demo webpage is provided. To run the compiled WASM SmartCGMS module, copy the `scgms.js`, `scgms.wasm`, and `scgms.worker.js` files to the root directory of the `SCGMS_Web_Demo`. To start the server with the demo webpage, use the `scgms_server.py` Python script. To be able to start threads in the WASM web SmartCGMS demo, `SharedArrayBuffer` needs to be enabled in the web browser as specified in Emscripten documentation for pthreads.

Introducing New Platforms

Once a SmartCGMS concept code has been successfully compiled for a new platform, it can be introduced to the build system using the following steps.

Create a Build Environment for the Platform

In the Build Environments repository, create a new folder with the name of the platform. In this folder, all the required drivers, Real-time operating system (RTOS), build script, CMake files, and other necessary files are located.

Modify winapi_mapping.cpp

Using preprocessor macros, specify functions for allocating and freeing memory in the `winapi_mapping.cpp` file.

Modify prepare_build.py script

Add the platform name to `supported_archs` variable in the `prepare_build.py` script. Create a function to copy the SmartCGMS concept source code and the build environment for the platform to a specified folder. Optionally, add a function to create a script/CMake file to specify required toolchains. Finally, add these functions to the `main` function in the script behind an `elif` statement. The functions to create the build directory of the new platform should also be added to the `elif` statement specifying all platforms.

List of Figures

2.1 Example of a SmartCGMS filter chain [27]. 10
2.2 Life cycle of a SmartCGMS filter [27]. 12
2.3 Operation of the SmartCGMS signal generation filter and a discrete model [27]. 12

3.1 Steps to deploy the SmartCGMS concept to different platforms 18

4.1 Example of a SmartCGMS filter chain containing data reading filters at the beginning and the end to calculate event statistics. 29
4.2 SmartCGMS WASM demo webpage: Filter chain configuration and pseudo-randomly generated data processed by the filter chain displayed in a `Chart.js` graph. 36
4.3 SmartCGMS WASM demo webpage: Additional output printed to the Google Chrome developer console by the SmartCGMS WASM module 37

5.1 ESP32 41
5.2 The SmartCGMS webpage 41
5.3 The event level values being collected on different platforms 41
5.4 Dynamic memory usage of SmartCGMS concept application executed as a FreeRTOS native application on Raspberry Pi Zero W and ESP32. Only the first four events are displayed since the allocated memory remains constant afterward. 44
5.5 Dynamic memory usage of SmartCGMS concept application executed on GNU/Linux as a native application, only the total memory consumption is displayed. The time displayed on the x-axis is measured in executed instructions. 45
5.6 Dynamic memory usage of SmartCGMS concept application executed as a WASM module in WAMR on GNU/Linux. The time displayed on the x-axis is measured in executed instructions. 46

5.7 Comparison of the SmartCGMS concept application execution times on different platforms: Native GNU/Linux, WAMR fast interpreter, and WAMR JIT mode. The y-axis displays time on a logarithmic scale. . . . 49

List of Tables

5.1 Memory Footprint of the SmartCGMS Concept Application 46

5.2 Execution Times of the SmartCGMS concept application on different platforms (in seconds). The top line refers to the number of processed events by the filter chain. 49

1 Required Tools for the SmartCGMS Concept . . . .

List of Listings

4.1 Example implementation of the resolve_symbol_static() function . 24
4.2 Lambda function to replace `do_get_filter_descriptors` calls to dynamic libraries 25
4.3 Example implementation of the create_filter_body() function . . . 26
4.4 Exponential moving average calculation 30
4.5 Calling JavaScript to post event data to the web environment running the SmartCGMS WASM module. 31
4.6 An example of JavaScript code contained in the `js_code` string from listing 4.5 31
4.7 An example of exporting symbols from WASM module using Emscripten 35
6.1 Cloning the Build System repository . . . .
6.2 Building a Docker image for executing the Python build script . .
6.3 Running the Python build script inside using provided Docker script. . . . .
6.4 Invokind Python build script directly. . . . .
6.5 Executing SmartCGMS concept application WASM module using `iwasm`. . . . .

Acronyms

AAPS AndroidAPS.

AOT Ahead-Of-Time.

API Application Programming Interface.

CGM Continuous Glucose Monitoring.

DIY Do-It-Yourself.

EMA Exponential Moving Average.

ESP-IDF Espressif IoT Development Framework.

FDA Food and Drug Administration.

GUID globally unique identifier.

HLA High Level Architecture.

IoT Internet of Things.

ISA Instruction Set Architecture.

JIT Just-In-Time.

LLVM Low Level Virtual Machine.

pthread POSIX thread.

RAM Random-access memory.

RTOS Real-time operating system.

SDK	Software Development Kit.
WAMR	WebAssembly Micro Runtime.
WASI	WebAssembly System Interface.
WASM	WebAssembly.