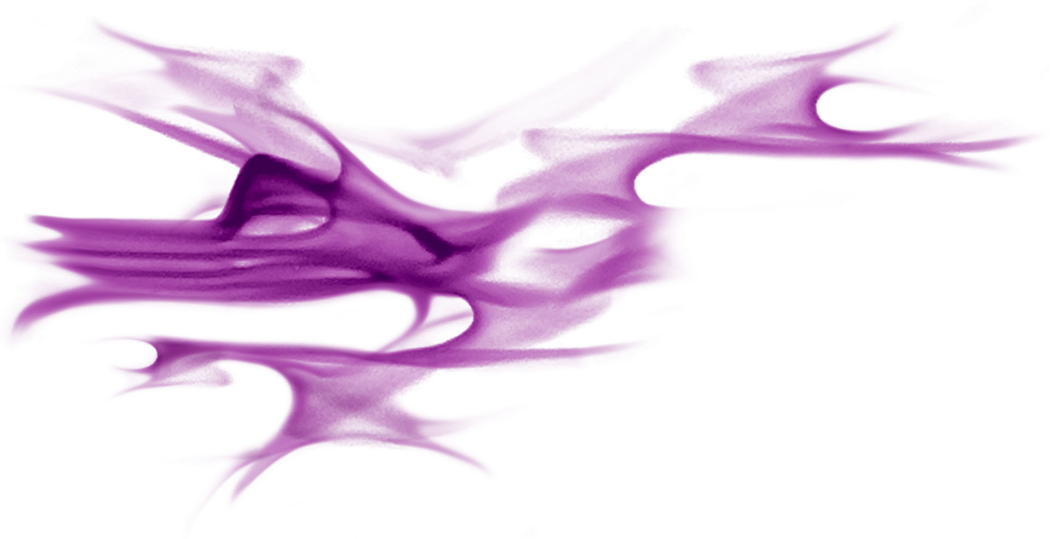
MATLAB to Fixed-Point C Code Generation and its Application to Real-Time Heartbeat Detection

Master's Thesis in Embedded Systems



Torfinn Berset

MATLAB to Fixed-Point C Code Generation and its Application to Real-Time Heartbeat Detection

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

EMBEDDED SYSTEMS

by

Torfinn Berset
born in Ålesund, Norway

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

TiP2: Body Area Networks
Human++ Program
Holst Centre/imec
Eindhoven, the Netherlands
www.imec-nl.nl

Cover picture: The imec “swirl”.

MATLAB to Fixed-Point C Code Generation and its Application to Real-Time Heartbeat Detection

Author: Torfinn Berset
Student id: 1531697
Email: torfinn@berset.no

Abstract

MATLAB is a popular very-high-level-language used for visualizing, prototyping and performing design-space exploration of algorithms. But, this flexibility comes at the price of high memory consumption and slow execution times, making it unsuitable for use in an embedded system.

The possibility of using Embedded MATLAB (EML), a small subset of the MATLAB language compatible with code generation code, is investigated to generate production level C code from a MATLAB algorithm. The hypothesis of this thesis is that this has the potential of bringing together the best of two worlds; a flexible design interface coupled with a resource-constrained and optimized implementation.

In this thesis, a workflow is defined to transform an existing algorithm from an unconstrained MATLAB algorithm to a constrained EML implementation. A new tool is also presented for assisting in automated floating-to-fixed-point conversion of MATLAB algorithms. This conversion will also transfer to the generated C code, thus potentially generating production ready C code.

To demonstrate the workflow, a case study on a *Real-Time Heartbeat Detection* algorithm is presented. It is concluded that although this technology is very promising it still has performance and usability problems that is keeping it from reaching its fullest potential, even if existing workarounds for these problems are included. However, Embedded MATLAB might yet serve its purpose as a bridge between software engineers and scientists, providing a common platform for research and development.

Thesis Committee:

Chair:	prof. A. v. Deursen, Faculty EEMCS, TU Delft
University supervisor:	dr. phil. H. G. Gross, Faculty EEMCS, TU Delft
Company supervisor:	ir. B. Grundlehner, imec, Holst Centre
Committee Member:	dr. S. Dulmann, Faculty EEMCS, TU Delft

Preface

This thesis is dedicated to Shen Jie, whose love and support made this work possible.

Thanks to Bernard Grundlehner, who has been my supervisor and friend through my internship and my thesis at imec/Holst Centre. Through countless discussions, I have gained a whole new appreciation for signal processing.

I would like to thank professor Hans-Gerhard Gross, who has challenged me to push my research further and yielded results beyond my expectations.

I would like to thank the following people for reviewing my thesis and giving constructive comments: Fabien Massé, Iñaki Romero, Julien Penders, Geir Berset, Jos Hulzink, Annika Maas, Michael di Fazio, Alex Young, Frank Bouwens, Michael de Nil, Dilpreet Buxi and Maryam Ashouei.

Finally, I would like to thank my family: Lillian, Finn, Geir and Trond, for all their support throughout my studies in Delft and Eindhoven.

Torfinn Berset
Eindhoven, the Netherlands
July 18, 2011

Contents

List of Figures

Chapter 1

Introduction

Many algorithm designers use the flexibility of MATLAB for design-space exploration, but for implementation of their algorithm in an embedded system, they must translate their algorithms to the low level C language. However, this path is bound with many difficulties.

There are fundamental differences between the two languages and each of them serves their distinctive purpose; on one hand, MATLAB is an interpreted language with a vast function library that allows interactive program development and debugging. This is beneficial for prototyping purposes, as the developer can defer to specify the size of the data or the data types, allowing the developer to focus on the science, not the implementation. But this flexibility comes at the cost of speed and memory consumption being several orders of magnitude worse than compiled languages.

On the other hand, low level languages such as C can be highly efficient and has widespread industry acceptance, a large range of supported platforms, and can meet the stringent operating requirements of a heavily resource-constrained embedded system, such as an autonomous wireless sensor node. However, algorithm development is difficult and tedious as memory must be pre-allocated, the code must be compiled after each modification and there are many more possible sources of error in the code due to increased complexity. Also, there are limited visualization tools and a limited function library, which are invaluable for rapid prototyping and testing.

MATLAB uses double-precision floating-point by default for all its calculations. Unfortunately, this is most often not possible to run on an embedded system, unless floating-point emulation libraries are used. However, these libraries most often give an unacceptable hit in performance and memory consumption. Fixed-point arithmetic is a good alternative to floating-point for a low-power embedded system, but it comes with its own set of design challenges that must be handled. In this thesis, these challenges will be addressed, and a fixed-point conversion tool compatible with the code generation workflow will be introduced.

Interpreted languages are a good choice when development time trumps execution time. Therefore, the concept of code generation of interpreted languages to compiled languages is an approach that might have the potential benefit of bringing together the best of the two worlds. A prototype could first be rapidly developed and validated in a high level interpreted language. Then, code generation tools could be used to translate it into production level

code in a compilable language; however, as we will see, such translation is far from trivial to accomplish.

In this thesis, Embedded MATLAB is examined, which is a subset of the MATLAB language that can be automatically translated into C code with MATLABs `emlc` code generator.

1.1 Research Questions

The research question investigated in this thesis is: can a generic and highly automated process for converting MATLAB algorithms to C code sufficiently efficient for deployment to resource-constrained real-time embedded systems be defined? We can split this question into sub-questions that will be answered in this thesis:

- Can a MATLAB to embeddable C code conversion with a high degree of automation be achieved?
  - What tools are available, and what are the limitation of these tools?
  - What steps are required to enable the automated generation of fixed-point C code from a MATLAB algorithm?
  - How can we (partially) automate the process of fixed-point conversion?
- How well does this method perform?
  - How does automatically generated code compare to manually crafted code?
  - What are the limitations and bottlenecks of this process?
  - What are the additional possibilities that open up by applying this method?

1.2 Contributions

The main contributions in this thesis are two-fold:

- A method for converting MATLAB algorithms to deployable C code:
  - A method of systematically creating Embedded MATLAB components that can be generated into embeddable C code.
  - An automated fixed-point conversion and data analysis tool;
  - An overview of the current shortcomings of the current tools.
- The embedded implementation, optimization and deployment of a *real-time heartbeat detection* algorithm, using the proposed MATLAB-to-C code generation workflow.

1.3 Thesis outline

In Chapter 2, the traditional workflow of manually translating MATLAB algorithms to C is investigated, and a new workflow based on automatic code generation is proposed. In Chapter 3, the steps required for the transformation of a MATLAB algorithm to the Embedded MATLAB subset is shown. In Chapter 4 data type conversion on the algorithm is performed and the *FixIT* tool for assisting in the automation of the process is introduced. To validate the proposed workflow, a case study on the transformation and optimization of a *heartbeat detection* algorithm is presented in Chapter 5.

Finally, the results are discussed in Chapter 6 and the thesis is concluded together with suggestions for future work in Chapter 7.

Chapter 2

Background and Code Generation Workflow

The traditional workflow of manually transferring algorithms from MATLAB to C has been difficult and error-prone. MATLABs flexible nature is a double-edged sword that makes it a useful tool for design-space exploration, but at the same time makes it increasingly difficult to transform into the much stricter boundaries of the C language. In this chapter, the context of the thesis will be given, and the traditional algorithm development and the manual translation process from MATLAB to C will be investigated. Finally, a new translation process is proposed, which is based on performing a series of transformations on the MATLAB code, leading to conformity with the `emlc` MATLAB-to-C code generator.

2.1 Background

2.1.1 Body Area Networks

A body area network (BAN) can provide medical, lifestyle, assisted living, sports or entertainment functions for the user [29]. The network consists of a series of small sensor nodes, each of which has its own power supply, consisting of storage and energy scavenging devices. Each node has enough intelligence to perform its own task autonomously. Furthermore, each node can communicate wirelessly with other nodes or with a central node worn on the body. The central node, *e.g.* a mobile phone, in turn communicates with the outside world using a standard telecommunication infrastructure such as a WLAN or a cellular network. The energy consumption of each building block must be drastically reduced to allow energy autonomy [14].

At Holst Centre/imec, the Human++ research project [29] tackles key technology challenges associated to micro-power generation and storage, ultra-low-power radios, ultra-low-power DSPs, sensors and actuators [28].

The information gathered can be used to provide services to the user, such as management of chronic disease, medical diagnostic, home monitoring, biometrics and sport and fitness tracking. An envisioned BAN platform for imec's Human++ program is shown in

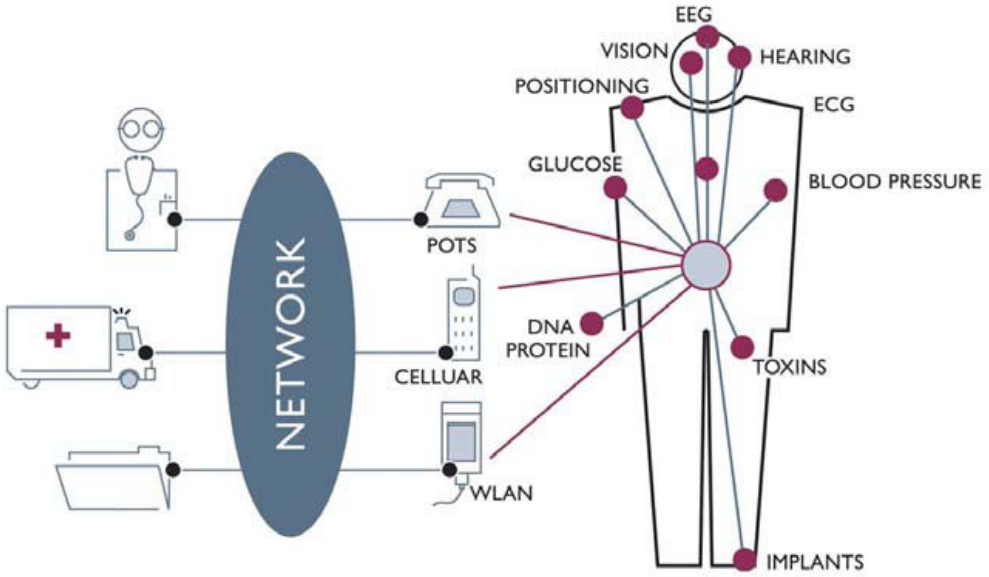


Figure 2.1: An envisioned Body Area Network

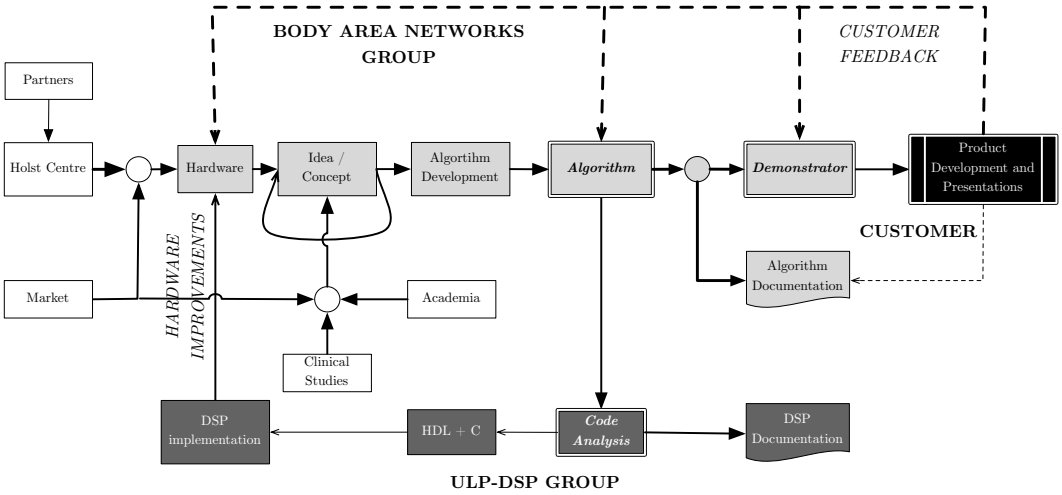


Figure 2.2: The life cycle of algorithmic development at imec/Holst Centre

Figure 2.1. In this thesis, the proposed workflow will be evaluated on the optimization and implementation of a heart-beat detection algorithm used in a Body Area Network.

2.1.2 Algorithm Development at imec/Holst Centre

At Holst Centre/imec, algorithms in the Wireless Autonomous Transducer Solutions (WATS) BAN group are designed for the purpose of producing demonstrators. These demonstrators display the capabilities of hardware developed at Holst Centre and are shown to its partners. The algorithm concepts are, in addition to internal initiative, derived from the market, clinical studies and academia. The concept iterates through several phases before it is ready to be implemented in a real-world context. After the concept MATLAB code is written, it is manually translated into C code to embed on internally developed hardware platforms. During the manual translation, additional changes in the C code are introduced to meet non-functional requirements such as execution time and memory usage.

The final MATLAB and C code is transferred to the Ultra-Low Power Digital Signal Processing (ULP-DSP) group for analysis and for creating the foundation for the requirements of future DSP development. The developed DSPs are integrated into future hardware platforms and new demonstrators are created. After presenting the demonstrators, feedback is gathered from the customers are looped back into the different phases to improve future generations of hardware and algorithms. This development process is outlined in Figure 2.2.

2.2 Manual MATLAB-to-C Conversion

MATLAB is by many considered to be the *de facto* platform for algorithmic development, but stepping out of MATLAB and implementing the same algorithms in C is a path bound with difficulties. For example, MATLAB is an interpreted language with no explicitly required data typing, while C is a statically typed compiled language. By default, MATLAB uses 64-bit floating-point arithmetic, however this does not translate well into the embedded computing domain; floating-point calculations are expensive in terms of processing power and memory. Furthermore, on embedded micro-controllers and DSPs, FPUs are most often not available, and computations must be performed using integer arithmetic.

In the process of translating algorithms from MATLAB to run on an embedded microcontroller, engineers face many constraints from the target platform, as well as difficulties in translating the inherent functionality of MATLAB. This translation process is often error-prone, time consuming and trial-and-error based due to the lack of proper guidelines. In this section, manual and automatic translation of MATLAB code to C is reviewed, along with their respective challenges.

2.2.1 Background

MATLAB syntax is based on compact matrix notation; it is designed to perform expressions with vectors and matrices in single-line expressions similar to their corresponding mathematical formulas. In equivalent C code, such functionality is accomplished by using iterators, such as nested `for` loops, to express matrix operations as a sequence of scalar computations. To create a model of the MATLAB algorithm in C, developers should put a hold on major algorithmic modifications in order to keep the verification of the transla-

tion manageable. Thus the C conversion usually happens late in the design process, which typically prolongs the development process and hence increase the cost of the project.

2.2.2 Example of Manual MATLAB-to-C translation

In this section, an example of manual MATLAB-to-C translation is studied, together with sources of translation error and methods of verifying the correctness of manually translated code.

Consider the MATLAB program[1] in Listing 2.1. This program has two inputs, a variable `x` and a variable `threshold`.

```
1  function [ y ] = trivialExample( x, threshold )
2
3    y = x( x < threshold ); % Return values of x less than threshold
4    y = sort(y);            % Sort the values ascendingly
5    y = y(2:end-1);         % Remove the first and the last value
```

Listing 2.1: A simple MATLAB function

If this example is run on a reversed sequence of prime numbers ranging from 17 to 2 with a threshold of 8, as seen in Listing 2.2, it will first discard values above 8 ($\{17,13,11\}$), secondly sort the remaining numbers ascendingly ($\{2,3,5,7\}$) and thirdly remove the first and last value ($\{2,7\}$).

```
6  >> trivialExample( [17 13 11 7 5 3 2] , 8 )
7
8  ans =
9
10       3      5
```

Listing 2.2: Results of MATLAB function

Although this algorithm is simply implemented in MATLAB with 3 lines of code, there are some considerations to be made and problems to be solved when translating it to C:

Problem 2.1 *The statement* `x( x < threshold )` *returns an array of undetermined size. It equals the values of* `x` *that is less than* `threshold`.

Proposed Solution The number of elements returned by the statement must be determined, so that the memory can be allocated accordingly.

Problem 2.2 `sort` *may not be available in the C library.*

[1] Example adapted from http://www.eetimes.com/design/automotive-design/4017563/MATLAB-to-C-translation-part-1-Pitfalls-and-problems

Proposed Solution Decide upon and implement a sorting routine in C.

Problem 2.3 *The variable* `y` *is reused several times in different contexts, sometimes with different variable lengths.*

Proposed Solution Introduce additional variables with their own unique names and allocated memory.

Problem 2.4 *The indexing* `(2:end-1)` *must be translated carefully.*

Proposed Solution In C, values starting with index `1`, not `2` must be kept. Also, C does not know the length of the input from only examining the array. The length of the input `x` must be passed as an argument to the function. Having multi-dimensional arrays exacerbates the problem even further.

Obviously, the trivial MATLAB algorithm rapidly transforms into a non-trivial implementation when written in C. This shows part of the inherent complexity of translating algorithms from a higher level language to a lower level language, and conversely: the added ease of development in higher level languages due to the abstraction of such implementation details. This allows the development to shift the focus from the *details* to the *concept*.

2.2.3 Challenges during manual MATLAB-to-C conversion

As shown, translating MATLAB to C can be a complex process. Here, some sources of complexity that can commonly lead to errors during the manual translation process are discussed:

- Array indices start with 1 in MATLAB, but with 0 in C. Although a simple change, it can easily introduce errors in the translation process, especially when mixed with bitwise-operations and boolean expressions.
- Much of the inherent complexity of manually converting MATLAB to C is due to the fact that MATLAB is an interpreted language. Hence, unless pre-allocated, the data types and variable dimensions are only known at run-time and cannot be deduced from simply examining the code.
- Storing of data in MATLAB is *column* major, whereas C is *row* major. This can be seen from the following example: Consider the matrix in (2.1). MATLAB stores this matrix in 2-dimensional memory as shown in (2.2), while C would store this matrix as shown in (2.3).

$$M = \begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix} \tag{2.1}$$

$$M_{ML} = \begin{bmatrix} a\,d\,g\,b\,e\,h\,c\,f\,i \end{bmatrix} \tag{2.2}$$

$$M_C = \begin{bmatrix} a\,b\,c\,d\,e\,f\,g\,h\,i \end{bmatrix} \tag{2.3}$$

The translator must consider the different storage orientations when translating algorithms that use matrices so that they do not operate on data along the wrong dimension.

- MATLAB supports reusing the same variable in different contexts with different data types, dimensions and sizes. In C all variables must be cast to a specific data type before use, and after it is initialized it cannot be changed. Thus, additional variables might have to be introduced.

- MATLAB supports vectorized notation, such as `sum(vector(10:20))` (that sums up the values between index 10 and 20 in the vector' array), which is not supported in C, but can be resolved by using traditional loops such as `while` or `for`.

The sum of all these discrepancies between MATLAB and C surmounts to making the translation process a substantial engineering effort.

During and after the conversion from MATLAB to C has been made, the correctness of the translation must be verified. There are several methods to verify the correct conversion of MATLAB to C: One method is to create a library as an interface to the compiled C code, then this library can be loaded in MATLAB and verified against the test bench. A second method is to translate the existing MATLAB test bench to C, and run the resulting code as a stand-alone application. A third method is to exchange data files between MATLAB and C.

2.3 Automated MATLAB-to-C conversion

As an alternative to translating MATLAB code to C manually, tools have been introduced for automatic MATLAB-to-C code generation. Code generation is the process of automatically transforming code from a source language to a target language through a specialized compiler, or *code generator*. In this thesis, the `emlc` MATLAB-to-C code generator is evaluated.

2.3.1 Motivation and Challenges

One of the main motivations for automatic code generation from a high level language to a lower level language is that the designer can focus on creating a solid algorithmic concept in the high level language, and let a code generator handle the implementation details of the lower level language. Hence, algorithm developers might be able to create embeddable code without stepping out of their comfort zones, which in this case is the MATLAB environment.

Another advantage of code generation is that developers only need to maintain one version of the algorithm. This is particularly important in the research industry, as new

hardware and algorithm developments can occur frequently, and are often just proof-of-concepts rather than a consumer ready implementation. In reality, the development of the algorithm itself often continues after the translation process has begun, hence the MATLAB algorithm and the C implementation must be realigned in order to produce comparable results. Most often, either the changes in the implementation code is not reflected in the concept algorithm is due to time-constraints (or negligence), or changes in the concept algorithm fails to propagate to the implementation in fear of "breaking the system". Code generation might provide an effective solution to this problem and can thus be of great value for certain applications.

Although code generation seemingly promises to be an effective solution, it is not necessarily that it will lead to an efficient solution as well. Jeff Bier, of technology analyst firm BDTI, remarks the following [41] on the advent of automatic code generation for MATLAB:

> "*This is a long-awaited development that will be welcomed by many, but it isn't a silver bullet. The considerations that engineers focus on when they do algorithm development are different from the key considerations in embedded software or hardware development. For example, a code translation tool isn't going to figure out how to segment your data and schedule your computations for efficient implementation, nor is it going to figure out where to apply parallelism*"

Although the last statement is debatable, it is easily arguable that code generation is not a panacea, and manual efforts are still required to create an efficient solution. In this thesis, the manual steps required to generate efficient C code from MATLAB code is examined, and it is it shown that it requires substantial effort from the developers.

2.3.2 Requirements for code generation

To generate code from a high-level source language, it is often required to constrain the usage of the language to a subset of the language, which is compatible with automatic translation tools. Though this thesis focuses on proprietary code generation tools by The MathWorks for generating C code from MATLAB code, it is believed that many of the concepts and methods shown here are valid beyond this particular combination of tools and languages.

Previously, third-party conversion tools were available in the form of *MATLAB-to-C Synthesis* (MCS) by Agility Design Solutions and *AccelDSP* by Xilinx; however, at the time of writing, their developers no longer support these tools.

In the remainder of this section, the different components required to generate C code from MATLAB code together with their dependencies, as seen in Figure 2.3, are discussed.

Embedded MATLAB

Embedded MATLAB [38] is a subset of the MATLAB language that can be automatically converted to C code using code generation software. This subset of features consists of

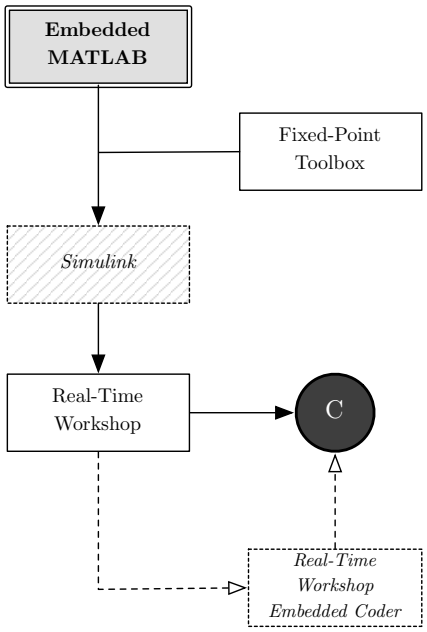


Figure 2.3: An overview of *MathWorks* products related to code generation

many of the most commonly used numerical MATLAB functions. On the other hand, Embedded MATLAB does *not*, amongst other, support objects, cell arrays, nested functions, visualization or `try`/`catch` statements. However, it is important to note that the full MATLAB environment is still available for development, visualization, debugging and prototyping; it just cannot be translated into C code.

The MathWorks organizes add-ons to the core MATLAB functionality into *toolboxes*, which are packaged individually. To generate C code from Embedded MATLAB, the following toolboxes are needed: Real-Time Workshop[2], Fixed-Point Toolbox, Simulink and optionally – the Real-Time Workshop Embedded Coder[3]. An overview of the dependency between the toolboxes can be seen in Figure 2.3. These toolboxes will be briefly described in the following sections.

Simulink

Simulink is a toolbox for MATLAB for multi-domain simulation and Model-Based Design for dynamic and embedded systems. Simulink supports both C (with Real-Time Workshop) and HDL (with Simulink HDL Coder) code generation for prototyping and implementation of embedded software and hardware. Embedded MATLAB code can with certain modifications be imported as a block in a Simulink model. This will open up more choices for code generation. This option was considered outside the scope of this thesis as it is not a common tool in use by algorithm designers.

[2]MATLAB Coder as of MATLAB 2011a
[3]Embedded Coder as of MATLAB 2011a

Real-Time Workshop

Real-Time Workshop [40] (RTW) is a toolbox for MATLAB and Simulink that allows the user to create automatically generated code from Embedded MATLAB functions and Simulink models.

Fixed-Point Toolbox

The Fixed-Point Toolbox [39] allows for fixed-point arithmetic from within MATLAB. The toolbox is not required for Embedded MATLAB per se, but it is required to generate *fixed-point C code* with Embedded MATLAB.

Real-Time Workshop Embedded Coder

Real-Time Workshop Embedded Coder is a *Simulink* extension of Real-Time Workshop primarily for code generation optimized for embedded systems. However, this toolbox also provides a few new options for C code generation from Embedded MATLAB models, but they are of lesser importance.

2.4 Workflow for MATLAB-to-C Code Generation

In this section, a workflow for MATLAB-to-C code generation is presented.

2.4.1 Functional and Non-Functional Requirements

There are additional non-functional requirements to be considered that have to be met that are not part of simply computing the correct result. For example, the algorithm has to execute within a certain time limit while also keeping the power consumption low. Furthermore, the application will run in a severely resource constrained environment, which limits the program to a minimal memory footprint. Last but not least, a certain degree of portability of the code should be maintained so that the code can migrate to a new hardware platform when more advanced technology becomes available.

The functional requirements are for the most part specified in the initial development of the algorithm. However a revision of the requirements is a sound practice at this stage to minimize the chances of unnecessary reengineering in the later stages of development due to functional changes.

2.4.2 Outline of Proposed Workflow

The target is to tailor a workflow that can be used by algorithm designers and embedded software engineers alike to transform MATLAB algorithms to C code implementable in an embedded system. The workflow will combine common software engineering practices, digital signal processing techniques, and steps specifically related to modifying MATLAB algorithms to a form that produces more desirable results when processed by a C code generator.

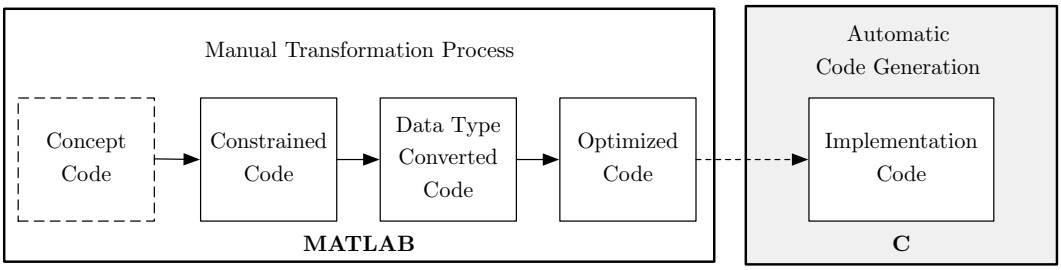


Figure 2.4: High-level view of proposed code generation workflow

As suggested by Cockburn [11], both an incremental and an iterative development process will be used. *Incremental* development is a staging and scheduling strategy in which various parts of the system are developed at different times or rates and integrated as they are completed. *Iterative* development is a rework scheduling strategy in which time is set aside to revise and improve parts of the system. Cockburn concludes that incremental development gives opportunities to improve the development process, as well as to adjust the requirements to the changing world, while iterative development helps to improve the product quality.

To meet non-functional requirements such as execution time, memory consumption and power usage, several iterations might be needed. After each iteration, the functional requirements must remain satisfied. This verification is performed through tests of the system.

It is important to note that this is a non-linear process; feedback loops exist within the workflow so that the code is continuously tested and refactored as the conversion proceeds.

Algorithmic Restructuring and Constraining

The first phase of the conversion process is started by preparing a test bench to verify the continued functional correctness and performance of the algorithm. Secondly, the existing code is cleaned up and reorganized through a series of defined code transformations. Thirdly, the code is conformed to the syntax required by the code generation software. Fourthly, blocks and buffers to process data in real-time are introduced. As the final step in the first phase, static memory allocation is introduced instead of MATLABs default dynamic memory allocation. At the end of this phase, floating-point C code can be automatically generated from the MATLAB algorithm. This process is described in Chapter 3.

Fixed-Point Conversion Phase

After constraining the code and defining the algorithms data flow in the previous phase, the workflow proceeds with converting the data types used in the algorithm from MATLABs default floating-point representation to a more computationally efficient fixed-point representation. There are two main steps in fixed-point conversion, namely determining the algorithms precision requirements and defining suitable fixed-point representations. To aid in the automation of these steps, a tool named *FixIT* has been developed, which will be discussed in Chapter 4.

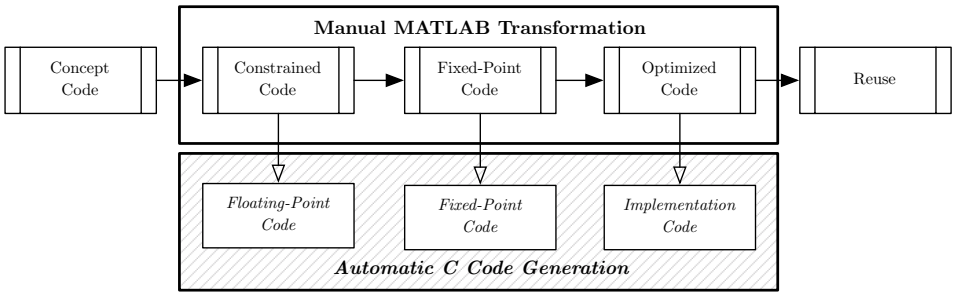


Figure 2.5: Code Generation Possibilities at Different Implementation Steps

Optimization

After the data type conversion, the resulting C code can be deployed on the target platform and profiling of the code can begin. This profiling information can be used to identify bottlenecks in the algorithm that must be improved in order to meet the non-functional requirements. To demonstrate the optimization process and evaluate the workflow, a case study is presented on the MATLAB-to-C conversion of a heartbeat detection algorithm in Chapter 5.

2.4.3 Code Generation Possibilities

At the end of each phase, it is possible to generate C code from the algorithm. After the constriction phase, *floating-point* C code can be generated. Accordingly, after the fixed-point conversion phase, *fixed-point* C code can be generated. After the final phase, *optimized* fixed-point C code can be generated, which is the goal of this workflow. An overview of the different possibilities for C code generation can be seen in Figure 2.5.

2.5 Summary and Discussion

In this chapter, the way algorithm development is performed at Holst Centre/imec was shown. Furthermore, the traditional workflow for translating algorithms from MATLAB to a C implementation on an embedded system was examined. This method was contrasted by a new emerging method that automatically translates MATLAB code into C. However, there are strict prerequisites that must be fulfilled before code generation can be performed, and even further refinements that must be carried out in order to create an efficient implementation.

To generate embeddable C code from MATLAB code, an iterative and incremental workflow is proposed that restructures and constrains the original code, converts the algorithm to fixed-point and performs iterative optimizations. This proposed workflow is detailed in the following chapters, before presenting a case study where the workflow is applied and evaluated.

Chapter 3

Algorithmic Restructuring and Constraining

In this chapter, the algorithm is constrained and transformed from MATLAB to Embedded MATLAB, the code generation compliant subset of MATLAB functions, without changing the native MATLAB data type of double precision floating-point.

During this process, unsupported functions for code generation must be removed or rewritten and computational complexity should be reduced to a level that is feasible for implementation on an embedded system. Furthermore, the algorithm is restructured and elements used for real-time data processing such as pipelining, buffering and static memory allocation is introduced.

At the end of this stage, floating-point C code can be generated from the Embedded MATLAB algorithm; however, this is not efficient enough for the intended target of a low-power implementation on an embedded system. Thus, in Chapter 4, data type conversion from floating-point to fixed-point is performed before optimizations and optimized C code is generated from the resulting MATLAB code. An outline of this step and the subsequent steps can be seen in Figure 3.1.

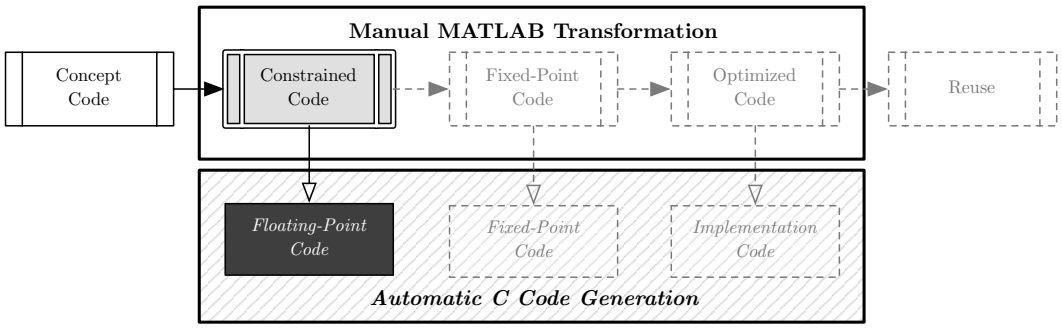


Figure 3.1: MATLAB-to-C flow — Constraining the Concept Code

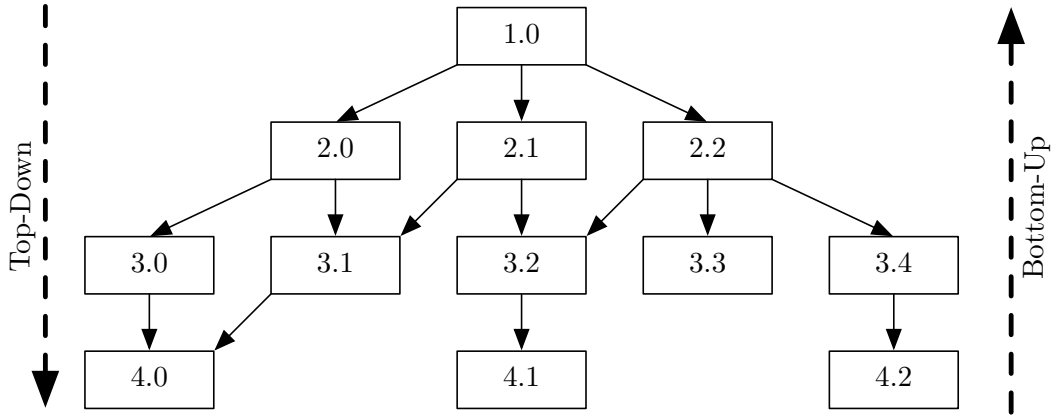


Figure 3.2: Two MATLAB-to-C conversion strategies

3.1 The Test Bench

In this section, a test bench is set up that will be used to verify the continued correctness and performance of the system as the transformations that will enable the generation of efficient C code are performed.

3.1.1 Background

Testing is an essential activity of software engineering [6], and testing for scientific software and automatic code generation is no exception. Several journal papers have been retracted because of bugs [12], emphasizing the importance of software testing in algorithmic development. Wilson [43] argues that journals must start insisting that scientists' computational work meet the same quality standards and reproducibility as their laboratory work.

Verification is the demonstration that the application correctly solves the equations embodied in the solution algorithm, while *validation* is the demonstration that the application accurately models all the important effects [8]. The testing strategy will only focus on *verifying* the results, not on the algorithm's validation, which is left to the original algorithm designer.

3.1.2 Testing Strategies

The test-bench should in addition to verifying the logical correctness of the algorithm also measure the algorithmic performance, as it can change when performing optimizations, approximations and during the conversion of the algorithm to fixed-point. The test-bench is usually application-specific; for example a heartbeat detection algorithm's test-bench should include detection accuracy, while a de-noising algorithm should measure improvement in signal-to-noise ratio.

There are two main approaches for converting a MATLAB algorithm into Embedded MATLAB compliant code, namely *bottom-up* transformation and *top-down* transformation, as seen in Figure 3.2.

The first method is to start at the leaf nodes and iterate towards the top node of the function hierarchy. This enables the testing of each component in isolation, thus simplifying the detection of Embedded MATLAB syntax violations. This method is suitable for Test-Driven Development [4] and unit-testing every component in the application.

The second method reverses the process by starting at the top node of the function hierarchy and traverse down to the leaf nodes. The advantage of this method it allows for the retention of the system-level tests [38].

3.2 Refactoring

This section examines how *refactoring* can be used to prepare the algorithm for further transformation towards the Embedded MATLAB subset.

3.2.1 Background

Here, refactoring will be used as a tool for cleaning up existing code in anticipation of further transformations required for code generation. Scientists often develop algorithms in an incremental, iterative manner to match the output with the theory in a trial-and-error fashion [8, 34, 35]. As such, the requirements are developed as the algorithm development progresses, which often leads to unstructured code. Evans [13] states that such processes can come to a halt unless frequent refactoring is applied to take advantage of newly gained insights to improve the model and the design.

Though an initial refactoring is necessary before starting the transformation process, refactoring is a continuos process that should be revisited regularly as the algorithm development transpires. This will aid in increasing the algorithm's maintainability and the reusability of its components.

3.3 Code Constraining

In this section, functions will be constrained to the Embedded MATLAB (EML) subset and unsupported functions will be rewritten within the boundaries of EML.

3.3.1 Background

MATLAB does not require the developer to specify the size of the data or the data type; this helps in allowing the developer to focus on the proving a concept, while deferring implementation details. However, if the target is to create efficient, embeddable, C-code, constraints must be imposed on the MATLAB code in terms of memory footprint and computational complexity.

Furthermore, MATLAB has a vast function library, but only a small subset of this library is compatible with the available code generation tools.

3.3.2 Writing Embedded MATLAB compliant code

While the Embedded MATLAB subset encompasses code generation support for basic arithmetic and programming constructs, it lacks support for visualization, objects, cell arrays, nested functions or `try`/`catch` statements.

Moreover, as only a relatively small subset of MATLAB's massive function library is supported for code generation with Embedded MATLAB, non-supported functions must be recreated within the Embedded MATLAB subset and be made compatible with the code generation tools. If the complexity of the functions is too high for an embedded implementation, approximation-functions can often be used.

3.3.3 Code Profiling

A technique that is helpful in determining bottlenecks in the implementation is to use the *profiling* tool available in MATLAB. As execution times for various functions on the development platform undoubtedly differs from the target platform, it should not be assumed that there is necessarily any relation between the two. However, the profiler can be used to track the execution count of each line and function, which be used as a decision basis for where to apply optimizations first.

3.4 Pipelining

In this section, the algorithm's data flow is restructured and *pipelining* based on block processing and buffers is introduced. This is suitable for processing data in real-time on an embedded system.

3.4.1 Background

MATLAB's default method of processing data is *batch-based*, however the target is to process data from sensor-inputs continuously. One method to achieve real-time processing is to split the data into consumable pieces, called blocks, and process them in a pipelined fashion. Once a block has been processed, the results are propagated through the system for further processing.

A *buffer* holds one or more blocks of data. The allocation of buffers is closely related to the block sizes. Here, *simple buffering* [30] is applied, thus buffer memory is allocated for the lifetime of the system.

3.4.2 Block Size

Small block sizes require less RAM as the data can be discarded more quickly. The system's *responsiveness* can be increased as well, as the time from the data enters the system until it is processed is shorter. However, this is assuming that the system can throughput the data fast enough without congesting.

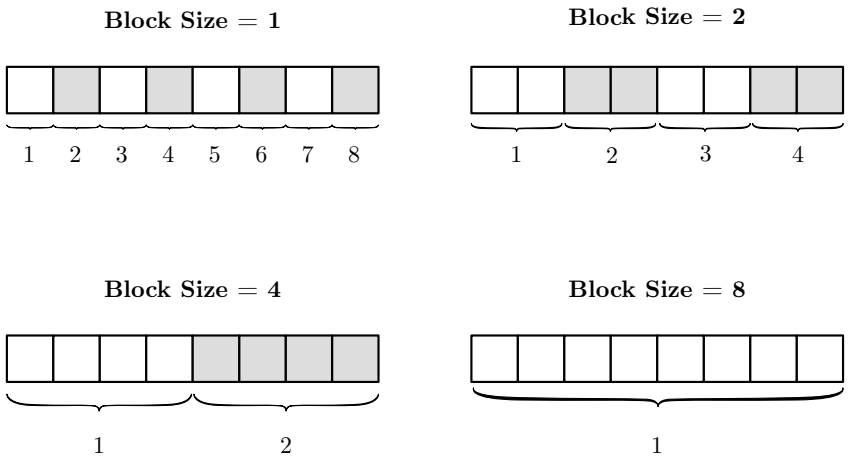


Figure 3.3: Four different example block sizes for processing eight samples

Larger block sizes usually lead to less overhead due to loops and function calls, but require more RAM as more data needs to be kept in memory at any moment in time. Furthermore, the system's response time to external changes will be slowed down. Grouping computations together allows embedded systems with low-power modes to hibernate for longer time-periods in-between blocks.

Typically, block sizes are allocated in sizes in powers of 2 to allow various optimizations, but there are no restrictions on the block size other than it must fit in the available memory. Usually, the block sizes are selected in relation to the monitored signals' sampling frequencies. In Figure 3.3, four examples on how different block sizes can be applied to a set of 8 input samples are shown.

To meet the application's requirements, appropriate block sizes for the different stages of the system's data flow are selected. The selection of the block size is often a trade-off between having a responsive, low-memory application and a computationally efficient, low-power application.

3.4.3 Buffers

There are several types of software buffers available, but here the focus will be on using *circular buffers*, as they are a suitable choice for streaming data in a real-time system. For now, assume that data is processed in a first-in, first-out (FIFO) fashion. The indexing of the buffer operates on modulo arithmetic, and thus it is very computationally efficient to select a buffer size that is a power of two. In this way, the modulo operation can be replaced with a simple bit-masking AND operation. The advantage of this buffer is that is very simple to implement, but a drawback is the imposed limitation in the selection of buffer-sizes if a power of two buffer length is opted.

In Figure 3.4, an example partitioning of buffers within an application is shown. Each of the functions operates on different block sizes, as discussed in the previous section.

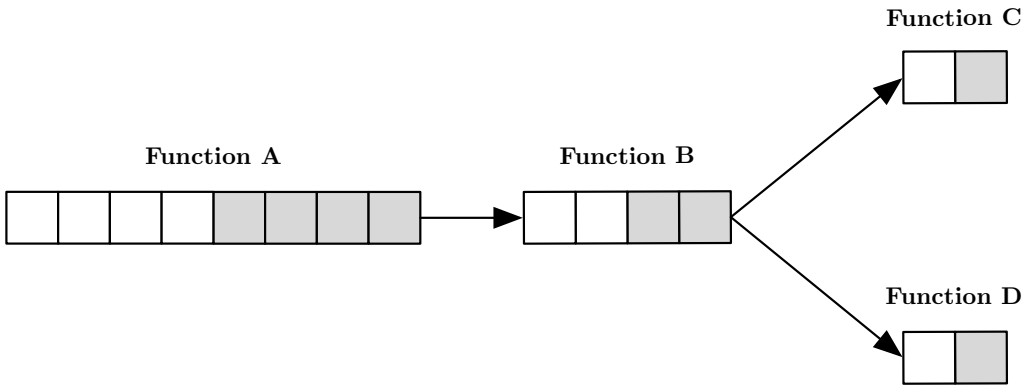


Figure 3.4: Example of buffering and pipelining within an embedded application

3.5 Static Memory Allocation

In this section, static memory allocation to the algorithm is introduced. Static memory allocation means that the memory is allocated at *compile-time* for the application's life time. This is in contrast to dynamic memory allocation, where allocation is performed at *run-time*, and memory can be allocated or deallocated as desired.

3.5.1 Background

In MATLAB, memory allocation is dynamic as it is an interpreted language. When transferring an MATLAB algorithm to a real-time embedded system, static memory allocation will be introduced as a step toward guaranteeing real-time operation.

In MATLAB, it is allowed to initialize an array and make it grow element by element. This means that each time the array expands, a new memory segment is allocated to fit both the previous array and the new values. This new memory segment is then copied with values from the previous array together with the added values. After copying, the previous array is freed up from the memory. This imposes an unacceptable overhead on the embedded system.

Embedded MATLAB uses in-depth analysis to calculate the upper bounds at compile time [47]. However, they must be specified explicitly when the analysis fails to detect an accurate upper bound.

3.5.2 Array Boundaries

Here, the bounding of array sizes by means of pre-allocation is discussed, which is a very important step in preparing the MATLAB code for C code generation. Pre-allocating arrays has the added benefit of greatly accelerating MATLAB simulations.

In Listing 3.1 is a valid MATLAB program; this program creates a variable that is initialized to an empty array. The statement inside the `for` loop sets the variable to the concatenated value of itself plus the loop index `k`.

```
11 a = []; % Define empty array
12
13 for k=1:5
14     a = [a, k]; % Concatenate array with itself plus 'k'
15 end
16
17 % Gives the output
18 >> a =
19
20       1      2      3      4      5
```

Listing 3.1: MATLAB – Dynamic Memory Allocation

The array `a` is concatenated with the value of loop index `k` for each iteration. Hence, the array is *growing*; to avoid this phenomenon, which could be either impossible or potentially disastrous on an embedded system, memory can be pre-allocated by using MATLAB's `zeros` function which creates a matrix or vector of 0s, as seen in Listing 3.2.

```
21 a = zeros(1, 5);  % Pre-allocate
22
23 for k=1:5
24     a(k) = k;
25 end
26
27 % Gives the output
28 >> a =
29
30       1      2      3      4      5
```

Listing 3.2: Embedded MATLAB - Static Memory Allocation

Enough memory must be allocated with `zeros` to fit all the required data. The resulting matrix of zeros can be used to insert the output data. Note that while both methods are valid in MATLAB, Embedded MATLAB only supports the second option of pre-allocation. The Embedded MATLAB compiler will produce a compilation error if the array is still growing after pre-allocating memory with `zeros`. It is also possible to use `eml.nullcopy( zeros(m,n) )` in order to skip the initialization of the memory to 0.

3.5.3 Variable Sized Data

Variable sized data is defined as data whose size might change at run-time [38]. In MATLAB, functions return variable sized data by default; this means MATLAB will dynamically allocate the minimum amount of memory needed to represent the result.

Consider the MATLAB function in Listing 3.3; in this example, the size of `y` is unknown; it is dynamically allocated by MATLAB to store the output. However, since Em-

```
31 x = rand(1,100); % Create an array of 100 random values
32
33 % Return indexes of every variable in x larger than 0.5
34 y = find(x > 0.5);
```

Listing 3.3: Dynamic Allocated Memory for Variable Sized Data

bedded MATLAB only supports static memory allocation, an alternate solution must be produced. One method to resolve the problem is to pre-allocate sufficient memory for the worst-case scenario, and also introduce an auxiliary variable to identify the last valid index in the allocated memory. To mitigate the *worst-case*, memory requirements can be reduced by choosing smaller data-types, which will be discussed in Chapter 4, or use smaller *block-sizes*, as discussed in Section 3.4. In Listing 3.4, a statically allocated solution is shown after modifying the example in Listing 3.3.

```
35 x = rand(1,100);          % Create an array of 100 random values
36 y = zeros(1,length(x)); % Pre allocated for worst-case
37 index = 0;                % Index of last valid value
38
39 for k=1:length(x)
40     if x(k) > 0.5
41         index = index + 1;
42         y(index) = x(k);
43     end
44 end
```

Listing 3.4: Statically Allocated Memory for Variable Sized Data

The valid results of the computation can now stored in `y(1:index)`.

3.6 Optimization

Next, high level optimizations can be performed on the algorithm, before proceeding to the fixed-point conversion which may require a freeze in the numerical dynamic range used. After each optimization, the functionality should be verified to remain the same, which can be done by using the test bench. After the code is verified and generated, it can be compiled for the target platform and benchmarks can be performed to measure execution time and memory consumption. The way the non-functional constraints affect the code is difficult to predict accurately without hands-on experience, therefore the more complex the software-hardware interactions are, the more the developers need to experiment [33].

The Embedded MATLAB language offers some special constructs and pragmas to aid in optimizing the code for the target platform. This includes loop unrolling, function inlining

and the generation of look-up tables. However, in many cases, either the MATLAB-to-C code generator or the C-compiler will perform these optimizations on their own.

3.6.1 Passing values by reference

One disadvantage of developing in MATLAB is that the MATLAB language itself does not have the concept of pointers, thus MATLAB handles arrays and matrices in a pass-by-value fashion. In Embedded MATLAB, pass-by-reference can be induced in the generated C code by using identical variable names for pairs of inputs and outputs. An example of how pass-by-reference can be achieved in the generated C code by modifying the MATLAB code is shown in Listing 3.5:

```
45 function [x] = manipulatePointer(x) %#eml
46     x.one = x.one .* 21;
47     x.two = x.two ./ 2;
48 end
49
50 function [y] = passByReferenceExample()
51     y = struct('one',2,'two',84);
52     y = manipulatePointer(y);
53 end
```

Listing 3.5: Pass by reference example

In this example, the function `passByReferenceExample` passes the struct `y` to the `manipulatePointer` function. The `manipulatePointer` function uses the argument `x` as both input and output, thus instructing the Embedded MATLAB compiler to pass the value by reference. Note that MATLAB will still perform pass-by-value during simulations, but the generated C code will now use pass-by-reference.

3.6.2 Loop unrolling

Loop unrolling can in some cases be performed to improve register and cache locality. The branch overhead due to for-loops can be reduced or removed, allowing deeper pipelining and decreased execution time. Unrolling loops requires more code memory, as the expressions to be executed are duplicated multiple times for different sets of data. Unrolling should be treated carefully as it can also lead to slower execution times depending on the actual code and the target hardware platform. For example for certain DSP implementations it is often an undesired feature as they can have highly optimized hardware loops without overhead. In Embedded MATLAB, the process of loop unrolling can be controlled by using the `eml.unroll` pragma.

3.6.3 Function Inlining

Inlining is the action of inline expansion of a given function. This means that the compiler inserts the complete body of the function in every place of the code where the function is

called. Inline expansion removes the procedure call overhead.

The `emlc` compiler uses internal heuristics during C code generation to determine whether or not to inline functions in the generated code. Configuration of the automatic process can be done by changing the inline settings in `emlcoder.RTWConfig`.

The `eml.inline(mode)` pragma is used to manually control the inlining of Embedded MATLAB functions. This has two modes, namely `always` and `never`. The compiler automatically decides whether or not to inline the function if the pragma is not specified. Inlining functions might give performance benefits in certain situations, but it is best to leave the decision of inlining to the compiler in the majority of the cases.

3.7 Summary and Discussion

After this stage, floating-point C code with the `emlc` code generator can be generated; however this is not sufficient for an efficient implementation on an embedded system. On the other hand, reflecting back on the algorithm development process in Section 2.1, it can be seen that the current state of the algorithm can be used as the starting point for code analysis for generating requirements for future DSP design. Furthermore, an added bonus of refactoring the code and introducing pre-allocation of memory is that the simulation time can be drastically reduced and bugs in the algorithm are easier to find.

In this chapter, the workflow of transforming an existing MATLAB algorithm into the Embedded MATLAB subset was described. During this process, pipelining, buffering and static memory allocation and initial optimizations was also performed and the algorithm was tested against unintended changes in the output. Converting the algorithm to Embedded MATLAB is only the first stage in the proposed workflow towards implementation ready embeddable C code; next, data type conversion is performed in Chapter 4 before the workflow is evaluated in Chapter 5.

Chapter 4

Fixed-Point Conversion

In the previous chapter, the foundation was laid for the MATLAB to embeddable C code generation by introducing constraints on the MATLAB code in order to meet some of the non-functional requirements of using the algorithm in the target system. In this chapter, floating-point to fixed-point conversion will be performed. Digital signal processing algorithms are usually specified with floating-point data types but they are most often implemented in embedded systems with fixed-point arithmetic to minimize hardware cost, memory usage and power consumption.

Conversion of the algorithm to fixed-point is the most demanding part of the transformation process of going from the initial concept code to the final implementation code. For DSP design, floating-point to fixed-point conversion has been in some studies [24, 15] shown to represent between 30% and 50% of the total implementation time. Thus, a methodology that automatically establishes the fixed-point definition is required to reduce the algorithms time-to-market [24].

Although MATLAB is a very popular language for algorithmic development, according to Müellegger [27], it lacks the equivalent of a conversion assistance tool such as the Fixed-Point Advisor that exists in Simulink. In this chapter, the Fixed-Point Integration Tool, a.k.a. *FixIT*, is introduced. This toolbox can be used to fill two gaps in the MATLAB-to-C conversion process by providing semi-automated floating-to-fixed-point conversion and a

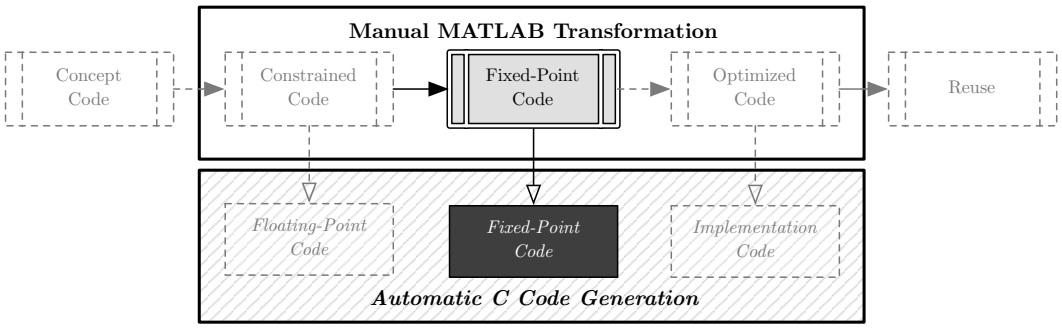


Figure 4.1: MATLAB-to-C flow — Fixed-Point Conversion

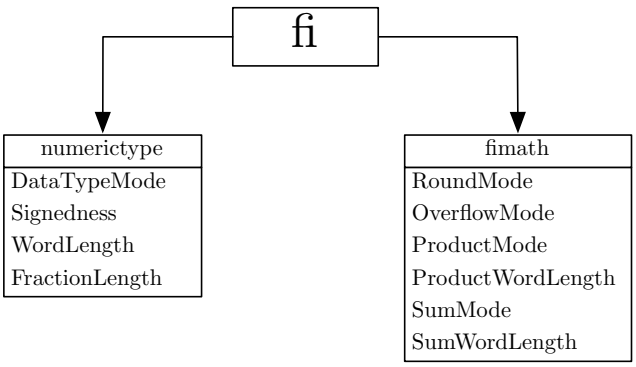


Figure 4.2: The Fixed-Point Toolbox fi object

means to log and visualize intermediate signals in the system without resorting to *ad hoc* solutions. Finally, some of the problems that currently haunt MATLABs data types for integer math and fixed-point will be discussed.

4.1 Background

MATLAB uses by default double precision floating-point for all data processing. Most embedded systems do not have floating-point units to perform floating-point arithmetic, as it is expensive in terms of power, speed and area. Fixed-point math provides a small, fast alternative to floating-point numbers at the expense of dynamic range. Converting from floating-point to fixed-point can be a tedious process, as there are many design considerations to take care of, such as rounding, overflows and underflows. Furthermore, an even smaller subset of functions the Embedded MATLAB subset is also compatible with the Fixed-Point Toolbox for code generation; thus unsupported functions must again be rewritten in order to compile to fixed-point C code.

4.1.1 Fixed-Point in MATLAB

The Fixed-Point Toolbox is used to instantiate fi objects in MATLAB. fi objects are fixed-point representations of real numbers containing word length, fractional length, signedness and arithmetic rules. The `fi` object is divided into two parts, `numerictype` and `fimath`, as can be seen in Figure 4.2.

The `numerictype` object contains all pertinent data related to the data type of the fixed-point object. This includes signedness, word length and fraction length. The `fimath` object contains the rules for performing arithmetic with other fixed-point numbers, such as saturation mode, rounding mode, product word length and summation word length.

The `fi` object also supports "Slope and Bias scaling", which replaces FractionLength with a scaling factor (most computationally efficient if is a power of 2) on the data, as well as a specifiable offset. This can be used to represent data that is either too large or too

small to fit in the normal word length. During code generation, `fi` objects and arithmetic is automatically translated into C code integer arithmetic.

4.1.2 Manual Fixed-Point Conversion with MATLAB

The MathWorks suggests the following workflow [39] for fixed-point conversion:

1. Implement the algorithm using fixed-point objects, using initial ”best guesses” for word lengths and scaling.
2. Activate floating-point override (overrides fixed-point objects with floating point numbers).
3. Turn the log overflow property on (provides warning and information on overflow in simulation).
4. Log the maximum and minimum values achieved by the variables in the algorithm in floating-point mode.
5. Deactivate floating-point override.
6. Use the information obtained in step 4 to set the fixed-point scaling for each variable in the algorithm such that the full numerical range of each variable is representable by its data type and scaling.

However, this process can be quite cumbersome because the developer (1) have to convert the algorithm to use `fi` objects first, thus slowing down simulation speed considerably, (2) have to manually extract the minimum and maximum values of each `fi` object with the `minlog` and `maxlog` functions and (3) still have to define the fixed-point scaling manually based on the results from step 2 and enter it back into the system. If the system changes its characteristics, steps two through six have to be manually repeated all over again to determine new data types.

4.1.3 Automatic Fixed-Point Conversion

Most fixed-point precision optimization methods are analytical, simulation based or a hybrid of the two. The current state of research on analyzing the quantization effects of floating-point to fixed-point conversion can be roughly divided into three groups [45, 9]:

The first group focuses on bit-true simulations, which strength lies in the ability to be able to model both Linear Time Invariant (LTI) and non-LTI systems. These methods guarantee that no overflow will occur, but lead to a conservative estimation [24]. Another disadvantage is the long simulation time and amount of input data that must be made available.

The second group is operating on statistical finite word length analysis methods [19] that has the advantage of being faster in simulations. Shi [37] examines two statistical methods, Monte-Carlo and analytical, for floating-to-fixed-point conversion. He concludes that

the methods can reduce simulation time by a factor of 100 compared to bit-true optimizations, and perhaps more importantly can achieve optimizations beyond what is achievable by bit-true simulation alone. This approach produces an accurate estimation of the dynamic range from signal characteristics, and it provides a guarantee against for overflow for signals *sharing the same characteristics*. Still, overflows can occur for signals with different statistical properties [24]; natural processes tend to have a Gaussian pdf, which has, by default, a range of $< -\infty, \infty >$. Therefore, overflows cannot be completely avoided, except for measured input signals, which are *saturated* by the ADC.

The third group of approaches is a combination of the two discussed above.

The fixed-point precision optimization methods can also be characterized by the degree of user interaction. Chang and Hauck [9] introduced a method that allows the user to contribute with guided choices (*User-Centric Automation*). They argue that designers should provide their input in all stages of the design process because they possess essential knowledge that cannot be deduced by today's tools.

To the author's knowledge, there are currently no tools specifically targeted for optimizing fixed-point calculations directly for Embedded MATLAB with Fixed-Point toolbox for the purpose of generating C code. Most works focus either on DSP design, usually for Simulink [9, 15, 37], or operate directly on C/C++ code [10, 19].

FixIT is a combined visualization tool together with a fixed-point word- and fraction-length optimization algorithm. This is where *FixIT* fills a niche in the current market.

4.2 Method

This section will show logged data can be used to automatically create a type definition file that can be integrated with existing MATLAB code.

4.2.1 FixIT overview

FixIT is currently constructed as a set of three modular objects: (1) the front-end DataInspector object, which logs the data to memory (2) the back-end NumericAdvisor object, which calculates a suitable data type for the data and (3) the FixIT object, which is the business logic that bridges the DataInspector and the NumericAdvisor and automatically generates MATLAB fixed-point definition wrappers from the results. In Figure 4.3, the flow of the fixed-point conversion tool is examined. The figure is annotated as follows:

1. Run algorithm with sample input and log intermediate calculation data in DataInspector;
2. Save data from DataInspector to storage (file) with the FixIT object;
3. Use FixIT to retrieve all logged data from storage;
4. Feed storage data into NumericAdvisor;
5. Get numerictypes suggested by NumericAdvsior;
6. Generate data type files;

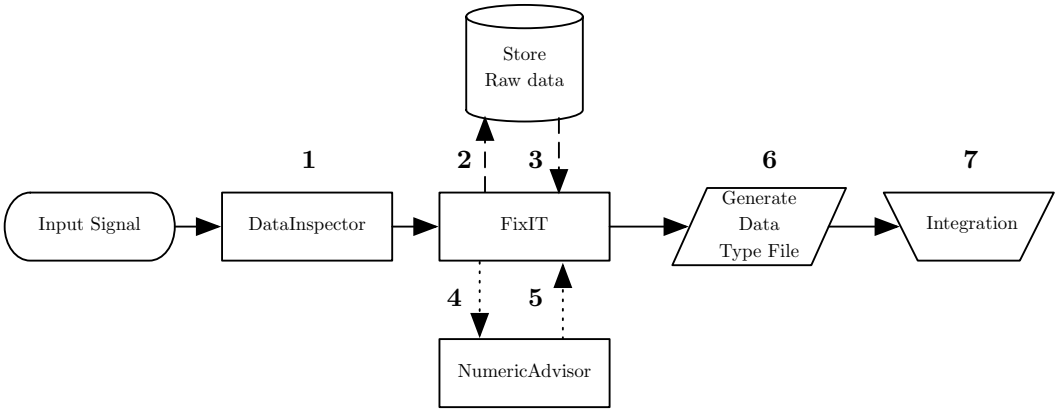


Figure 4.3: Steps involved of logging input signals and outputting data definition files

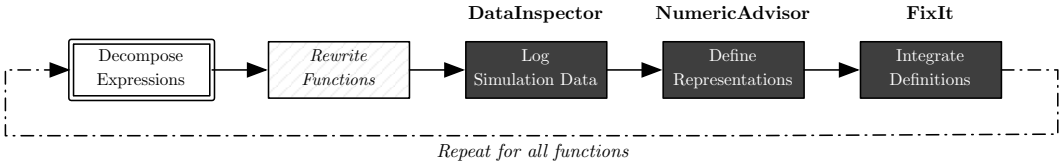


Figure 4.4: Floating-Point to Fixed-Point Conversion Workflow with FixIT

7. Integrate data type files with the MATLAB code.

4.2.2 Target Hardware

To generate fixed-point definitions, the target hardware for the calculations must be defined using the `eml.HardwareImplementation` object. To create compact and fast executing code, the native word length of the hardware target is used where possible.

4.3 Fixed-Point Conversion Workflow

This section describes the workflow for translating an algorithm from floating-point to fixed-point using the FixIT tool. The workflow is outlined in Figure 4.4.

4.3.1 Decomposing Expressions

In order to understand the properties of the system, the calculations must be decomposed to atomic expressions for numerical analysis. Consider the expression in Listing 4.1:

Here, only the final data type used for `a` can be derived, while the properties of `b`, `c`, `d` and `e` remains unknown. The fixed-point definition for the intermediate products must also be defined, thus the statement can be further decomposed as seen in Listing 4.2.

```
54      a = (b * c) + (d / e);
```

Listing 4.1: Composite Expression

```
55      bc = b * c;
56      de = d / e;
57      a = bc + de;
```

Listing 4.2: Decomposed Expression

After the decomposition, the behavior of each calculation can be studied individually and a suitable fixed-point definition for can be determined for each step.

4.3.2 Rewrite Functions

Many MATLAB functions are not supported for fixed-point C code generation, though they might be supported for floating-point C code generation. Thus rewriting of more functions in Embedded MATLAB syntax, as discussed in Chapter 3, might be necessary before proceeding.

4.3.3 Logging Simulation Data

As a proposed solution to logging and visualizing data, the *DataInspector* is introduced. Though the DataInspector still requires additional code, it can more easily be excluded from code generation and globally disabled if no further data logging is required. The collected data can be visualized or stored to disk through the textual user interface.

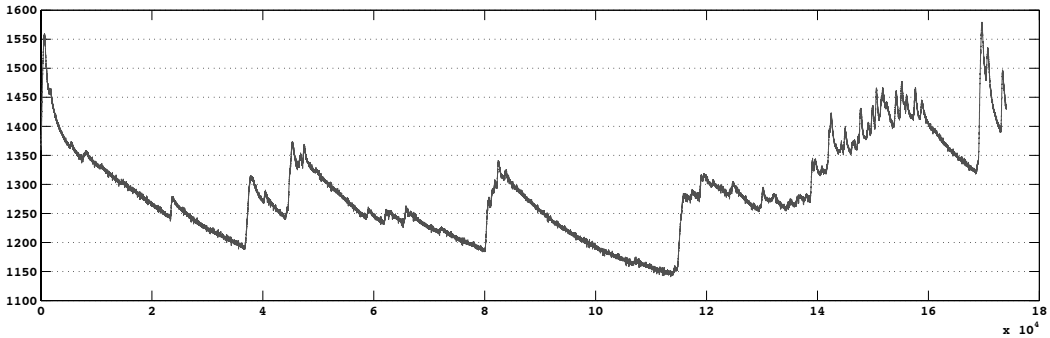


Figure 4.5: Example of GSR signal logged with DataInspector

For logging, it is advised to use several input signals to give a realistic representation of the system [19]. An example of a signal logged with the DataInspector can be seen in Figure 4.6.

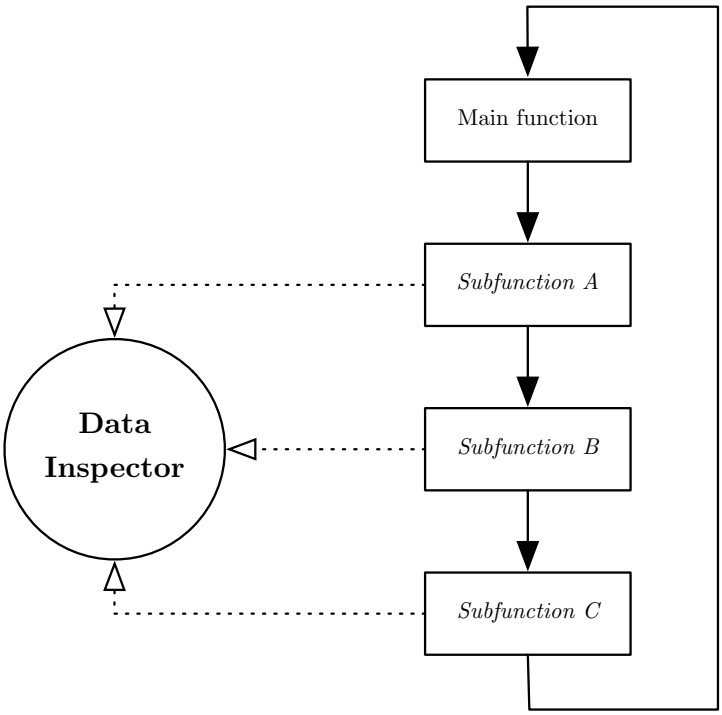


Figure 4.6: Logging data from the algorithm to the DataInspector object

To reduce taxation on memory, DataInspector logs the data in single-precision floating-point format by default, but can be set to any MATLAB data type.

4.3.4 Define Representations

Storage of logged data is performed by the FixIT class, which retrieves the data logged in the DataInspector and stores it in a folder hierarchy beneath the logged functions root folder. An additional identification parameter may be given at storage time to create a named data set.

The FixIT class works as a link between the DataInspector, which logs the data, and the NumericAdvisor, which selects appropriate data types. The FixIT object will retrieve all data in the DataInspector object and save it to disk. Next, the FixIT collates any previously logged data with the current data and dispatches it to the NumericAdvisor for analysis.

The NumericAdvisor is an object that contains a data type selection algorithm. Based on the input signals characteristics, such as dynamic range and signedness, a suitable `numerictype` is defined. It uses a simple algorithm to determine a suitable fraction-length to represent the data:

1. Determine dynamic range;

2. Set sign bit;

3. Determine if all data is integer valued or not;
4. Find a suitable word length;
5. Determine if a slope is required to scale the signal;
6. Determine fraction length or slope;
7. Add optional guard bits to prevent overflow;

A flow-chart of the algorithm can be found in Figure 4.7.
The minimum integer width required can be defined [19] as:

$$min(I_L) = \lceil log_2 R(x) \rceil \tag{4.1}$$

The NumericAdvisor has an option to add additional guard bits to the integer length to prevent overflow. This can be used to prevent overflow if it is uncertain how representative the input data is.

The most commonly used metric to evaluate the computation accuracy is the signal-to-quantization-noise ratio (SQNR) [24]. This metric defines the ratio between the desired signal power and the quantization noise power.

The NumericAdvisor is an object that contains a data type selection algorithm. Based on the input signals characteristics, such as dynamic range and signedness, a suitable `numerictype` is defined.

After logging the signals with DataInspector, the FixIT object gathers the logged data and stores it to disk. This data is then collated with existing data and fed into the Numeric-Advisor object to produce a set of suitable data types. These data types are then used for MATLAB code generation of data definition files, called `ntype` files. Data definitions will be loaded from the `ntype` file after integrating the `ntype` files with the existing MATLAB code.

For fixed-point optimization, it is paramount that the selected data set is representative. A meaningful base for analysis should represent the common case as well as boundary and extreme cases [9].

4.3.5 Integrate Definitions

After determining the data type definition with the NumericAdvisor, the FixIT class stores the definition in a *ntype* file (See Figure 4.8). This file is in fact an executable specification of the data type definition that can be integrated with the original code. A very important advantage is that the definition file can be updated without performing additional changes to the algorithm. Thus, if either new data becomes available or if the hardware platform changes, the fixed-point definitions can be updated while leaving the algorithm unchanged.

The `ntype` function files are used by the `nt` function. The `nt` function is a shorthand for extracting the fixed-point definitions stored in the `ntype` files, and will thus be the simplest way for the developer to integrate the automatically generated fixed-point definitions with the algorithm. The `nt` function also serves a purpose of providing an abstraction layer

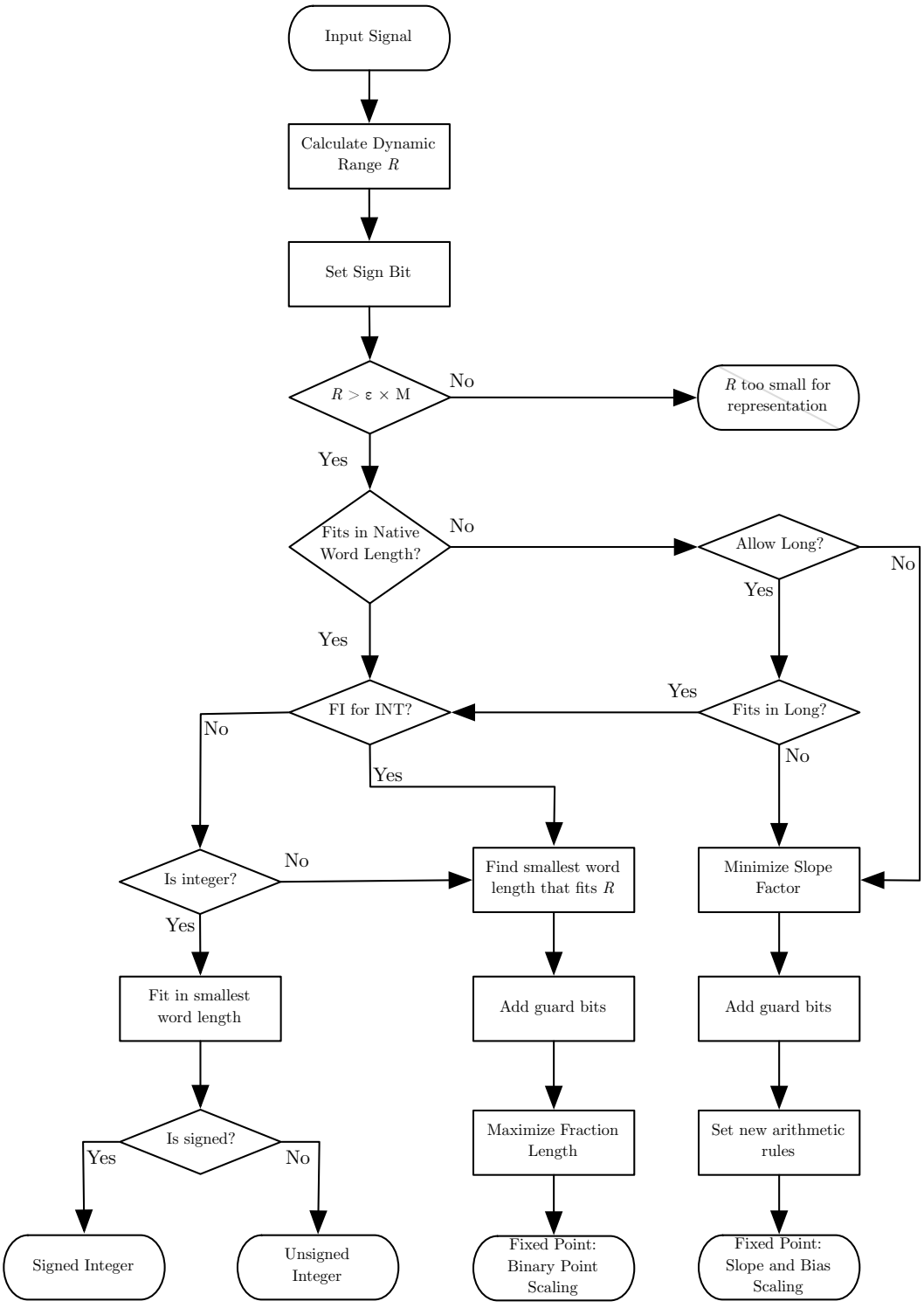


Figure 4.7: The NumericAdvisor's selection algorithm of suitable datatype for input signal

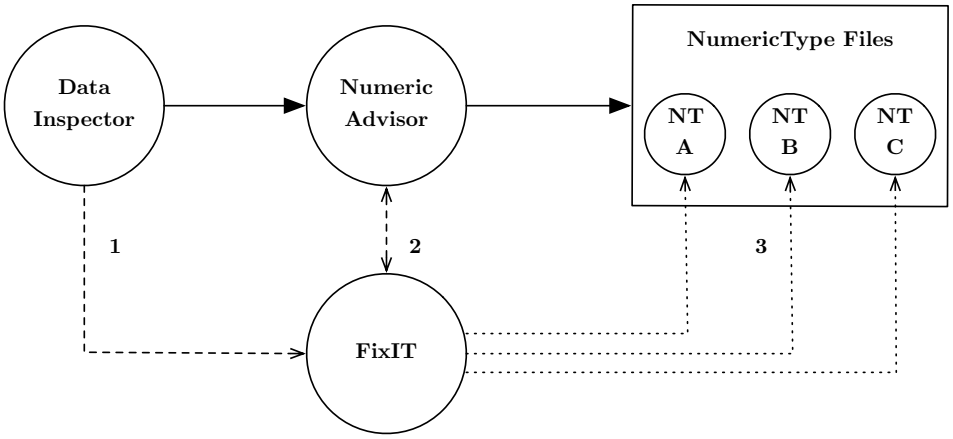


Figure 4.8: Generating `ntype` files with FixIT

between the algorithm and FixIT, thus making it easier to change the interface of the `ntype` files at a later stage, if needed. The process of integrating `ntype` files back to the algorithm with the `nt` function is illustrated in Figure 4.9.

4.3.6 Updating Data Definitions

To gather more data, the MATLAB data types must be overridden from Fixed-Point to Floating-Point using `fipref('DataTypeOverride','TrueDoubles')` and the data logging process must be rerun. If data collection is not needed, but either the hardware platform or the NumericAdvisor has changed, only the *FixIT* `ntype`-file generation process needs to rerun.

4.4 Discussion and Next Steps

We believe FixIT would be a very useful tool for developers wanting to perform rapid prototyping of algorithms using MATLAB's code conversion tools. The fixed-point conversion process in MATLAB has traditionally been problematic; to quote Vikström [42]:

> *The use of the Fixed-Point Toolbox puts too much limitations on the algorithm development team. They do not have the skills necessary to be able to define all the bit sizes of the fi-objects which would render their fixed-point MATLAB model less useful. This also means that the generation of fixed-point production code is difficult.*

However, with this new tool and workflow, fixed-point production code might become more feasible.

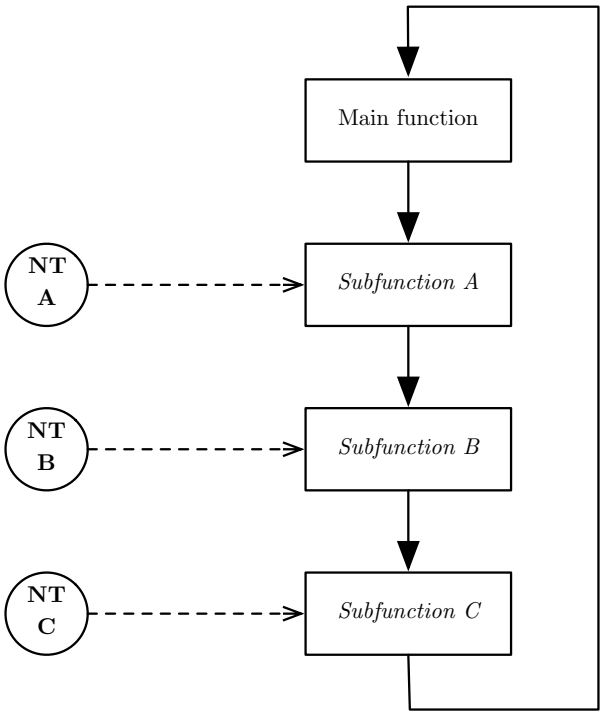


Figure 4.9: Inputting data types from `nt` files to the algorithm

4.4.1 Known issues

The FixIT tool suite has some known limitations, which are described here.

First, for certain calculations, like the accumulation part of a MAC operation, it is desirable to keep the summation word length equal or greater than the product word length. In other words, the product is not downscaled to fit in the original word length. Currently, the tool is stuck with one word length unless the desired product fixed-point definition is manually defined based on the logged data from DataInspector. The selection of different word lengths as an additional parameter to the `nt`-function, which retrieves the generated data type definition, can extend the current workflow.

Secondly, there is currently no specified behavior to handle signals with a dynamic range close to or less than $\varepsilon(x)$. During the automatic data type selection, the program will generate a warning if signal with a very small dynamic range is detected. The user must then manually solve underlying issues that have lead to this problem, by for example scaling up the signal, or rearranging calculations.

Thirdly, getting the type definitions from file introduces additional overhead to an already taxed simulation speed. Possibilities of performing data type substitutions directly in the source code can be investigated to alleviate this problem.

4.5 Issues Regarding Datatypes in MATLAB

In this section, some of the issues that haunt the integer and fixed-point data types currently used in MATLAB are discussed.

4.5.1 Issues regarding Fixed-Point numbers in MATLAB

There are currently some issues related to fixed-point numbers in MATLAB that should be discussed:

Slow simulation speed: Introducing fixed-point objects in the code will severely impede its execution speed. This can in some cases be mitigated by compiling the MATLAB code to an executable file (MEX), but then part of the convenience of interactive development and debugging is lost, which is one of MATLAB's greatest strengths.

Requires additional Toolbox: Fixed-Point numbers are not a native functionality of MATLAB and requires an additional commercial toolbox by the MathWorks.

Limited function support: Many important mathematical functions are still not supported by the Fixed-Point Toolbox. However, they may of course be manually coded in (Embedded) MATLAB.

Cannot access Fixed-Point object properties at compile time: It is not possible to access the fixed-point properties such as `WordLength`, `FractionLength` and `Signedness` at compile time. This added functionality would help to generate different C code based on the fixed-point specification.

4.5.2 Integers in MATLAB

There are two main categories of integer classes, namely unsigned and signed integers. MATLAB supports signed and unsigned 8-, 16-, 32- and 64-bit integers. Integer arithmetic is performed in the same manner as floating-point arithmetic in (Embedded) MATLAB.

There are a few serious issues with integers in MATLAB that are reason for concern when developing embedded algorithms. Here, some of the current problems with the integer classes in MATLAB[1] are summarized:

Overflow behavior in MATLAB differs from behavior in C: In MATLAB, signed and unsigned integers *saturate* at their boundary values. The ANSI-C [18] specification for unsigned integers is that it should `wrap around` upon overflow. For signed integers, ANSI-C does not specify the behavior on overflow, but the most common (and simplest) implementation is to let it wrap in the same way as unsigned integers.

No warnings available upon integer overflow: In the current version of MATLAB, there are no warning messages similar to what is available in the Fixed-Point Toolbox for fixed-point numbers that can be used to debug overflow.

[1] Evaluated for MATLAB 2010a

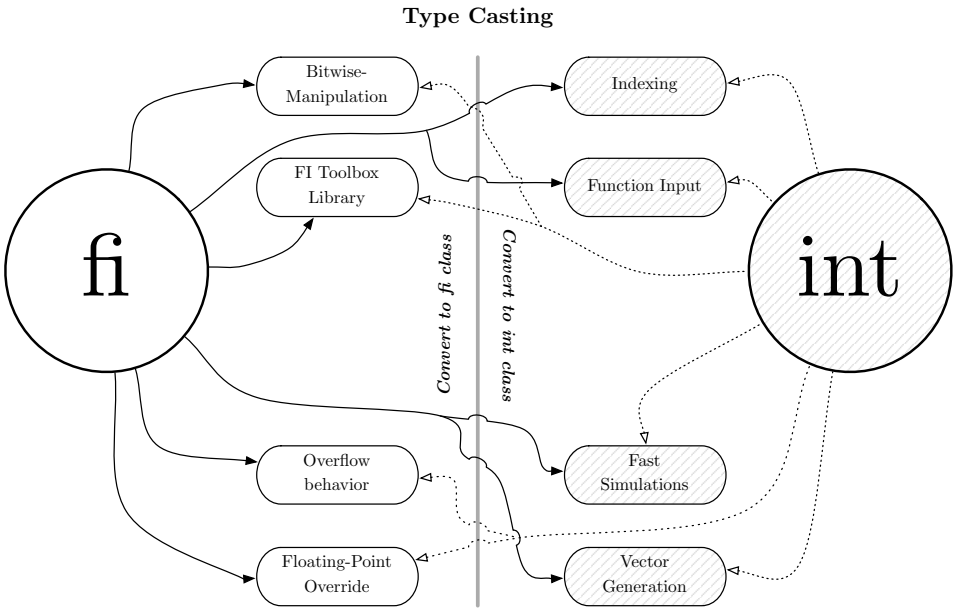


Figure 4.10: Implementation gap between fixed-point and integers in MATLAB

No bitwise arithmetic for signed integers: The bitwise functions `bitshift`, `bitand`, `bitor`, `bitxor`, `bitcmp`, `bitget` and `bitset` are not supported for signed integers in MATLAB. A workaround is to cast the signed integer to a signed fixed-point object, perform the bitwise functions, and then cast the fixed-point number back to a signed integer.

No data type override functionality: Unlike the Fixed-Point Toolbox, integers have no equivalent of the `fipref` tool. This hinders an algorithm using the integer classes to be simulated in full precision with `fipref DataTypeOverride` on.

Limited function support: Integer classes have even less support than fixed-point classes in the MATLAB function library.

Integer and Fixed-Point Class Mismatch

There are further reasons for concern when combining integers with fixed-point arithmetic in MATLAB. In C, there is no difference between a fixed-point number and an integer; the binary point is only implied, and an integer can always be regarded as a fixed-point number with no fraction length. However, in MATLAB these are two separate concepts and currently there is an implementation gap in the interaction between the two, as illustrated in Figure 4.10.

To create an efficient embedded algorithm, utilization of properties of both classes are required. For example, for correct overflow behavior with an integer, the fixed-point class must be used, but to use the result to index an array, typecasting the result back to an

appropriate integer class is needed. Although the generated C code will not be directly affected by these problems, this ends up being one of the major bottlenecks in MATLAB-to-C code generation workflow.

4.6 Summary and Discussion

In this chapter, fixed-point conversion and its challenges were discussed. Most importantly, the *FixIT* tool-suite was introduced to aid in fixed-point MATLAB-to-C conversion.

The suite consists of three parts; the *DataInspector*, which logs data from the algorithm, the *NumericAdvisor*, which determines suitable data types based on the logged data and the *FixIT* class, which channels the data through the other two classes and generates data type definition files. The data type definition files are subsequently integrated with the original algorithm and are *Embedded MATLAB compliant* so that they can be used in C code generation without any overhead in the generated code.

One of the most important properties of the FixIT suite is that the fixed-point definitions can be updated without interfering with the overlying algorithm. This saves time when the properties of the signals in the system changes, and the fixed-point types must be updated accordingly. This also enables the use of different fixed-point definitions for different hardware platforms using the same MATLAB algorithm. This can prove to be a powerful tool when combined with the automatic C generation tools.

Finally, the *DataInspector* tool can also be used individually to log and visualize and debug the system's data flow, a feature that is currently missing a standardized implementation in MATLAB.

Chapter 5

Evaluation

In this chapter, the MATLAB-to-C code generator's appropriateness for generating implementation ready code is evaluated through a case study about a real-time heartbeat detection algorithm.

5.1 Case Study: Heartbeat Detection

Heartbeat detection algorithms used for daily cardiological monitoring need to meet certain requirements in order to run efficiently in ambulatory systems. Robustness against noise, low-power consumption as well as minimized memory footprint and computing complexity are vital for operating in such conditions.

The heartbeat detection algorithm serves as the basis for most heart rhythm analysis algorithms. Because of the availability of validated test data and a manually translated implementation, the algorithm presents itself as a suitable candidate for exploring optimizations that can be performed with Embedded MATLAB. One of the main goals of this study is to explore the limits of optimization possible from this environment without resorting to manually write or modify C code.

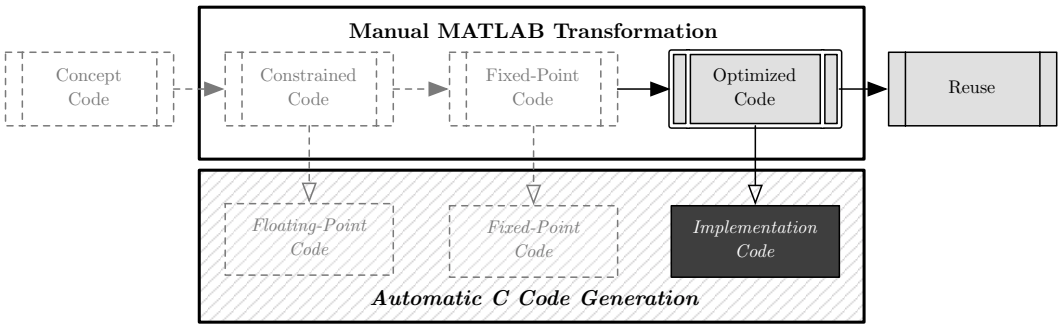


Figure 5.1: MATLAB-to-C flow — Optimization and Implementation

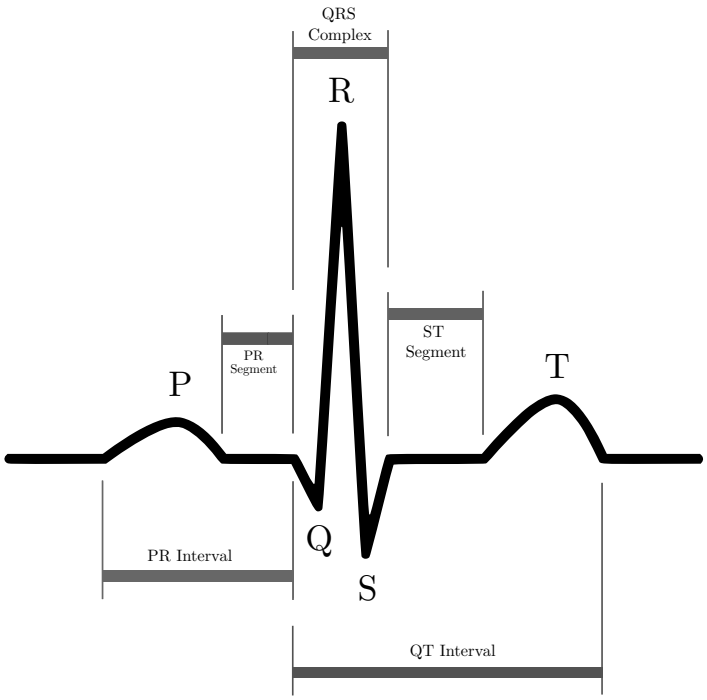


Figure 5.2: Schematic representation of a healthy ECG

5.1.1 Beat detection

The electrocardiogram

Since the invention of the String Galvanometer in 1901 by Einthoven, the electrocardiogram (ECG) has become one of the most important diagnostic tools in cardiology. The importance comes from the ability to diagnose arrhythmias and several other cardiopathies with high accuracy without resorting to inversive methods.

In a healthy ECG, usually several waves that correspond to the electrical activity of different heart chambers can be observed. These waves are named with the letters **P** - **Q** - **R** - **S** and **T**; a schematic representation can be seen in Figure 5.2. This nomenclature was introduced by Einthoven, and is still used in present-day cardiology.

The QRS complex, as seen in Figure 5.2, is according to Köhler *et. al.* the most striking waveform within the electrocardiogram [20]. The QRS complex reflects the ventricular electrical activity during ventricular contraction; this can provide the cardiologist with information about the time of its occurrence as well as its shape, giving insights on the current state of the heart. Due to its characteristic shape, it is often used in heartbeat detection algorithms. This section will describe a continuos wavelet transform (CWT) based algorithm that detects the R-peak in the QRS complex.

Heartbeat detection has been a research topic for over 40 years, and recent technological advances have enabled ultra-low power embedded implementations. The heartbeat detection algorithm studied in this thesis is based on previous work by Romero *et al.* [32, 31].

The CWT Based Heartbeat Detection Algorithm

This section will give a brief overview of the main steps of the heartbeat detection algorithm, shown in Figure 5.3.

The beat-detection algorithm is piecewise executed on 3 seconds of collected ECG data. The ECG signal is sampled at a f_s of 198 Hz, meaning that a minimum of $3 \times f_s = 594$ samples is needed in the buffer. By using a circular-buffer, as shown in Figure 5.4 (b) and described in Section 3.4.3, it is calculated that the buffer must be at least $2^{\lceil \log_2(3\times 198)\rceil} = 1024$ samples wide.

The first step of the algorithm is to (1) subtract the mean from the ECG signal. Then, (2) the CWT is calculated, which transforms the ECG signal from *time-domain* to *wavelet-domain*. In wavelet-domain (3) all values above a given threshold is found and grouped into *clusters*. Within each cluster (4) the largest coefficient is found before further processing is performed in the *time-domain* (5) in order to find the exact position the heartbeat. At this stage, any detected beats in the first and last 0.5 s must be removed to avoid the CWT border effect, thus overlapping must be performed, see Figure 5.4 (a), to process the entire signal. After finding the heartbeats, (6) the algorithm detects if there are any found heartbeats within a proximity of 0.15 s of each other, as this is most likely an erroneously detected beat. If two beats are detected within this proximity, the heartbeat with the largest wavelet-domain amplitude is kept. The final output of the algorithm is the sample numbers of the detected heartbeats within the 3 second ECG input signal.

5.2 Methods

This section will discuss the means of which the heartbeat detection algorithm will be optimized and evaluated. First, the evaluation platform for the study is described, then the means of the optimizations performed will be discussed. The MATLAB algorithm was initially constrained and converted into fixed-point using the workflow previously described in this thesis.

5.2.1 Target Platform and Evaluation Criteria

Algorithm Optimization Evaluation Criteria

To measure the effect of optimizations on this algorithm, metrics are needed from which the results can be evaluated. The ordered evaluation criteria are described in Table 5.1.

Holst Centre Universal Sensor Node

The Universal Sensor Node (USN) developed at the Holst Centre is a device designed for supporting several different BAN applications. The USN is designed with a mixture of custom- and COTS-components for ultra-low power applications. At its core lies a 16-bit Texas Instruments MSP430 micro-controller and a Nordic nRF24L01 radio to communicate with other sensor nodes and base stations. The USN has a 12-bit ADC that can record ECG

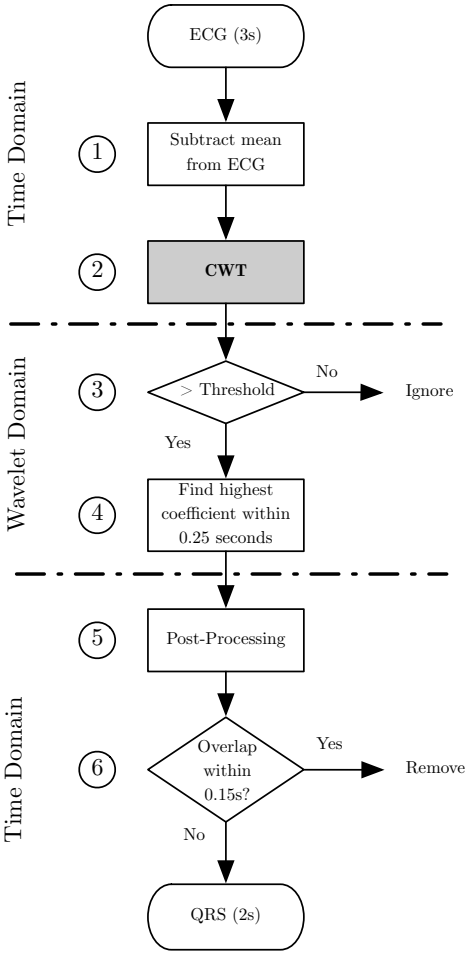
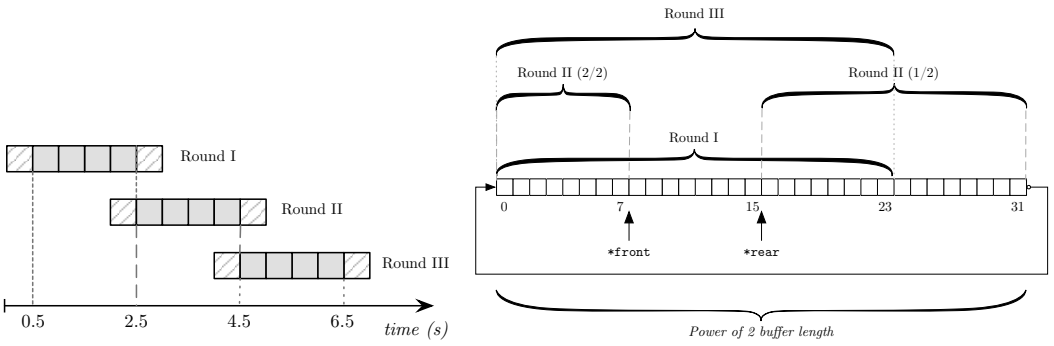


Figure 5.3: Outline of the Heartbeat Detection Algorithm



(a) Overlap in the ECG signal processing (b) Circular Overlapping FIFO buffer for ECG samples

Figure 5.4: Buffering in the Heartbeat Detection Algorithm

Critera	Description
Detection Accuracy	The accuracy of the algorithm has the highest priority.
Execution Time	Can be approximately be mapped to power consumption. Measured here in *clock cycles* on the MSP430
Noise Robustness	Robustness of the algorithm under ambulatory conditions.
RAM Usage	The available RAM on the MSP430 is much more scarce than ROM.
ROM Usage	The code and lookup-table memory usage is considered secondary to RAM usage.

Table 5.1: Evaluation criteria for heartbeat detection implementation

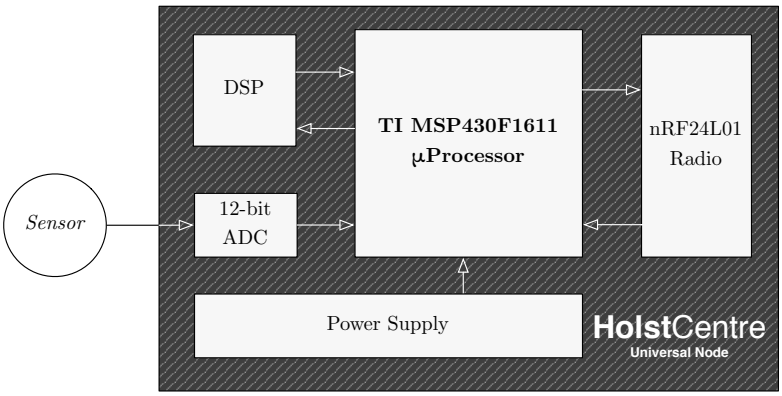


Figure 5.5: Universal Sensor Node

from attached sensors. The ADC is connected to imec's ultra-low-power biopotential ASIC readout [46]. A schematic overview of the USN can be seen in Figure 5.5.

The MSP430x1xx family [1] of CPUs by Texas Instruments has a 51 instruction 16-bit RISC architecture. Specifically, the MSP430F1611 Mixed Signal micro-controller [2] used in imec's USN has a 8 MHz CPU and has 48 kB of ROM and 10 kB of RAM. The USN also has a hardware multiplier [7], which supports any combination of 16- and 8-bit, signed or unsigned, (MAC) multiplications. Multi-word multiplications are possible through software libraries.

Benchmark Platform

The benchmarks in this case study are carried out on the platform described in Table 5.2.

5.2.2 Heartbeat Detection Accuracy

Two different databases are used to evaluate the heartbeat detection accuracy. The first database is the MIT-BIH database, and the second database consists of in-house recordings at imec/Holst Centre. This section will describe the two databases and the algorithm's evaluation protocol.

Item	Value
Model	Apple MacBook Pro 5.1 (Late 2008)
Processor	Intel Core 2 Duo, 64-bit, 2.53GHz
RAM	4 GB 1067MHz DDR3
L2 Cache	6 MB
Bus speed	1.07 GHz
Operating System	Mac OS X 10.6.4
MATLAB version	7.10.499 (2010a)
C compiler	IAR C/C++ Compiler V5.10.1.20144

Table 5.2: Platform for MATLAB benchmarks

The MIT-BIH Database

The MIT-BIH database [25] is a collection of ECG recordings gathered at Boston's Beth Israel Hospital between 1975 and 1979. Its creation was spurred by the advent of automated arrhythmia analysis algorithms in the 1960s and 1970s, which was lacking a common dataset for comparative analysis. With universal access to an extensive and quantifiable database, research on automated arrhythmia analysis and heartbeat detection flourished [26].

The MIT-BIH database is still considered one of the *de facto* standards [26, 31] for beat-detection evaluation in medical literature. The database consists of 48 half-hour recordings from 47 patients diagnosed with different forms arrhythmia. The ECG signal was sampled with a 11-bit ADC converter at 360 Hz. The database contains a wide spectrum of different rhythmic and pathological patterns that can be found in clinical practice [31]. In order to keep the evaluation consistent, a subset of the recordings was used. This subset consist of the 24 ECG signals that were previously used to evaluate the beat-detection algorithm in Romero [31, 32]. Future evaluation protocols should however be extended to include the complete database.

The imec Database

The imec database consists of 45 in-house 10-minute ECG recordings made during ambulatory conditions [31]. Ten healthy volunteers were measured during three different levels of activity: resting, biking and running. The importance of the imec database in the evaluation is that it is recorded with the same hardware that is used to implement the beat-detection algorithm. Hence, the recordings were made with a 12-bit ADC at a f_s of 198 Hz.

Evaluation Protocol

To signify the correct detection of a beat, it is specified that the algorithm must detect the *R-peak* (see Figure 5.2) within $\pm 100ms$ of the *annotated R-peak* in the database. This margin is necessary because the MIT-BIH recordings were annotated by hand and they use several leads for annotation, in which the fiducial point is not always synchronous; thus they are unreliable for comparing the exact fiducial point of the beat. To measure correctness, the

metrics *Sensitivity* (**Se**) and *Positive Predictivity* (**PP**); both these values are measured in percent, thus the highest obtainable value is *100*. The combined metric, **Se+PP**, is often used to write the results in a more compact notation; for this metric the highest obtainable value is *200*. Sensitivity is defined as shown in Equation 5.1.

$$Se = \frac{True\ Positives}{True\ Positives + False\ Negatives}, \tag{5.1}$$

where *True Positives* refers to correctly detected beats and *False Negatives* refers to missed beats. The definition of Positive Predictivity is shown in Equation 5.2.

$$PP = \frac{True\ Positives}{True\ Positives + False\ Positives}, \tag{5.2}$$

where False Positives refers to incorrectly detected beats.

The recordings in the MIT-BIH database use mV as the unit for ECG amplitude; this was converted to a 12-bit signal to fit in the same dynamic range as the ADC on the Universal Sensor Node, as described in Section 5.2.1.

5.2.3 Noise Robustness

To evaluate the beat-detection algorithm's robustness to noise, an official dataset was used consisting of a clean ECG signal combined with a noise signal, *em*. As in [31], signal 100 from the MIT-BIH database was chosen as the test signal, as it is a very clean ECG signal, and the *em* signal is taken from the MIT-BIH Noise Stress Test Database. This *em* signal represents electrode motion artifacts and is considered to be the most relevant source of noise in an ambulatory monitoring setting [31].

When s is the clean ECG-signal, and n the noise signal, and their respective signal power is denoted S and N, the signal-to-noise ratio can be calculated as:

$$SNR = 20 \times \log_{10} \frac{S}{N} \tag{5.3}$$

where

$$S = \sqrt{\overline{(s^2)}} \tag{5.4}$$

and

$$N = \sqrt{\overline{(n^2)}} \tag{5.5}$$

With this formula, a SNR range varying from -10 to +10 dB in steps of 1 dB was evaluated in terms of Sensitivity and Positive Predictivity.

Setting	Value
Language	C
Language Specification	C89/C90 (ANSI)
Saturate on Integer Overflow	Off
Enable Variable-Sizing	Off
Inline-threshold	10
Inline-threshold max	200
Inline-stack limit	4 000
Stack Usage Max	10 000
Constant Folding Timeout	10 000

Table 5.3: `emlcoder.RTWConfig('GRT')` compilation settings for benchmarking

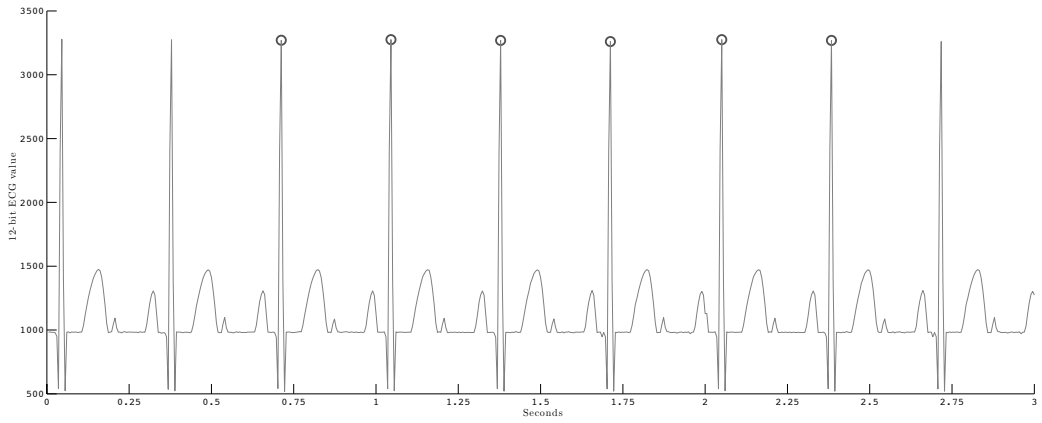


Figure 5.6: ECG Signal used for Profiling C code

Note that the algorithm discards beats found in the first and last 0.5 seconds

5.2.4 Execution Time and Memory Footprint

All benchmarks in this case study is performed with the Real-Time Workshop configuration, for the Embedded MATLAB compiler `emlc`, seen in Table 5.3.

The benchmarks in this case study are based on a static 3 second 12-bit ECG signal, which was generated by a function generator and sampled at 198 Hz, as seen in Figure 5.6. The signal was generated using an ECG signal generator connected to the sensor node, producing a 3 Hz signal, which is equivalent to 180 bpm. The signal was transmitted wirelessly from the sensor node to a host computer running MATLAB where it was logged. After logging the signal, it was put back into the ROM of the MSP430 for benchmarking. In the following results, the stored ECG signal is excluded from the memory usage. However, the given memory usage for all implementations are combined with the firmware that is required to operate the radio and low-power modes of the Universal Sensor Node, described in Section 5.2.1.

The MSP430 C compiler from IAR can optimize for speed, memory or attempt to strike

a balance between the two. In this case study, the compiler is set to optimize only for speed. The profiler in the simulator of the IAR MSP430 Workbench is used to determine the number of clock cycles spent. In the next sections, the methods of which the heartbeat detection algorithm is optimized will be discussed.

The Continuous Wavelet Transform

The Continuous Wavelet Transform is the central signal processing part of the beat detection algorithm. The algorithm [21] is defined as:

$$T(a,b) = \frac{1}{\sqrt{a}} \int_{-\infty}^{\infty} x(t) \psi^* \left(\frac{t-b}{a} \right) dt \tag{5.6}$$

where $\psi(t)^*$ is the complex conjugate of the wavelet function $\psi(t)$, a is the dilation parameter of the wavelet, and b is the location parameter in of the wavelet. At the center of the CWT is a 2D convolution transformation, which will be discussed in the next section. A Mexican hat wavelet is applied, which is the second derivative of a Gaussian function, defined as:

$$\psi(t) = (1-t^2)e^{-\frac{t^2}{2}} \tag{5.7}$$

By varying the length of $\psi(t)$, the number of coefficients used to represent the Mexican hat can be adjusted. The initial $\psi(t)$ range in the CWT algorithm is ± 8. The wavelet coefficients are calculated in MATLAB based on the current f_s and stored as a look-up table.

In the heartbeat detection algorithm, the CWT function transforms the ECG signal from *time-domain* to *wavelet-domain*, where further processing is performed to find the fiducial points of the R-peaks in the ECG signal.

Convolution Optimization

The two-dimensional convolution algorithm is the central part of the CWT algorithm, as discussed in the previous section. The convolution algorithm is both central the heartbeat detection and ubiquitous in the signal processing domain. It is by far the most computationally demanding part of the heartbeat detection algorithm, and thus is a prime candidate for optimization. The discrete convolution [23] is defined as:

$$y[n] = \sum_{k=0}^{M} h[k]\, x[n-k] \tag{5.8}$$

Convolution is a supported function for code generation in the Fixed-Point Toolbox. However, this implementation focuses on minimizing code size with rather than maximizing performance. Since the convolution is the main bottleneck in the algorithm, it can be a good trade-off to sacrifice code memory in order to obtain a justifiable decrease in execution time.

Before any further evaluation, the borders of the convoluted signal that are not used are removed. The parameters `convStart` and `convEnd` are introduced to delimit the convolution, corresponding to the lower and upper limit of n in (5.8). For the MATLAB built-in

convolution function, this optimization is not possible, and the borders must be removed *after* they have been calculated. Thus time is wasted in both initially calculating them and subsequently removing them.

To increase run-time performance, minimizing the branch overhead is attempted by writing a new Embedded MATLAB convolution function and splitting the main loop into three parts: head, body and tail. This is a similar concept to *software pipelining* [17]. The splitting of the convolution algorithm should accomplish the following: (1) removal of all *conditional* branches and (2) allow for *unrolling*, of the convolution *body*.

Comparison of Manually Translated and Automatically Translated Code

To compare manually translated and automatically translated fixed-point C code, a manual translation of the heartbeat detection algorithm is investigated, which was previously made by two researchers working at Holst Centre/imec. The largest traceable difference in the manual C implementation is that the convolution was performed with 32-bit fixed-point coefficients, thus having to resort to slow software libraries for 32×32 bit multiplications. This leads to more memory consumed both for code and data. In an attempt to provide a fair comparison, the different parts of the algorithms will be aligned to a high degree.

5.2.5 Simulation Speed

To evaluate the impact on execution speed of the transformation on the algorithm, the execution speed will be measured at different stages of the conversion process.

5.2.6 Benchmarked Implementations

For different parts of the method applied, different implementations of the heartbeat detection algorithm is used. As references, the original unmodified MATLAB algorithm and the manually translated algorithm are used.

For evaluation, three distinct Embedded MATLAB implementations of the algorithm are tested that were transformed from original MATLAB algorithm to fixed-point C using *FixIT*. The first version attempted to use the built-in Embedded MATLAB functions as much as possible, while the second and third version uses custom MATLAB libraries. The difference between the latter two is in the number of coefficients used in the CWT.

5.3 Results

In this section, quantitative and qualitative results from optimizing the heartbeat detection algorithm are presented.

5.3.1 Heartbeat Detection Accuracy

In Table 5.4, results from the best-effort automatic code generation implementation (*EMLC Optimized*) are compared with the original MATLAB algorithm, taken from Romero, Grundlehner *et. al* [32].

	EMLC Optimized			Reference MATLAB Algorithm		
FileId	**Se**	**PP**	**Se+PP**	**Se**	**PP**	**Se+PP**
100	*100.00*	*100.00*	*200.00*	*100.00*	*100.00*	*200.00*
101	99.89	99.73	199.62	*99.95*	99.73	*199.68*
102	*100.00*	*100.00*	*200.00*	*100.00*	*100.00*	*200.00*
103	*100.00*	*100.00*	*200.00*	*100.00*	*100.00*	*200.00*
104	*99.91*	98.98	198.89	99.87	*99.20*	*199.07*
105	99.30	*98.46*	*197.76*	*99.49*	98.27	*197.76*
106	99.26	99.95	199.21	*99.80*	*100.00*	*199.80*
107	*99.72*	*100.00*	*199.72*	99.44	99.95	199.39
118	*99.96*	99.87	199.83	*99.96*	*99.91*	*199.87*
119	*100.00*	99.95	199.95	*100.00*	*100.00*	*200.00*
200	*99.92*	*99.96*	*199.88*	99.88	99.92	199.80
201	99.34	*100.00*	199.34	*99.54*	99.95	*199.49*
202	*99.91*	99.72	199.63	99.77	*100.00*	*199.77*
203	*98.05*	*99.45*	*197.50*	97.78	98.95	196.73
205	99.89	*100.00*	199.89	*99.92*	*100.00*	*199.92*
208	99.29	99.83	199.12	*99.46*	*99.90*	*199.36*
209	*100.00*	*100.00*	*200.00*	*100.00*	*100.00*	*200.00*
210	*99.09*	*99.89*	*198.98*	98.00	99.85	197.85
212	*100.00*	*100.00*	*200.00*	99.96	*100.00*	199.96
213	*99.91*	*100.00*	*199.91*	*99.91*	99.97	199.88
214	99.65	99.96	199.61	*99.82*	*100.00*	*199.82*
215	*100.00*	*99.91*	*199.91*	99.91	99.88	199.79
217	99.05	99.82	198.87	*99.64*	*99.86*	*199.50*
219	99.91	*100.00*	199.91	*100.00*	*100.00*	*200.00*
Total	***99.66***	***99.81***	***199.47***	**99.65**	**99.79**	**199.44**

Table 5.4: Heartbeat Detection Benchmarks on MIT-BIH database

Implementation	**Se**	**PP**	**Se+PP**
MATLAB Reference	99.87	99.91	199.78
Manual Effort	99.99	99.95	199.94
EMLC Optimized	99.99	99.97	199.96

Table 5.5: Beat-Detection Benchmark on IMEC database

As can be seen from the table, the new algorithm slightly outperforms the reference MATLAB algorithm in terms of both Sensitivity and Positive Predictivity. This is related to algorithmic changes that were introduced in the MATLAB to Embedded MATLAB conversion process. The previously manually translated C implementation has a Sensitivity (Se) of 99.82 and a Positive Predictivity (PP) of 99.77, making a combined score of 199.59. Certain design choices that were made during the manual translation led the C implementation to be even more accurate.

For the imec database recordings, three different best-effort evaluations are presented in Table 5.5. Here, the Embedded MATLAB generated C code is the best performer in terms of both Sensitivity and Positive Predictivity.

5.3.2 Noise Robustness

In this section, the beat-detection algorithm's robustness to noise is evaluated.

CWT modification's effect on noise

The results of varying the range of the CWT's $\psi(t)$ on the *EMLC Custom* heartbeat detection algorithm's noise robustness are shown in Table 5.6. It can be seen that generally the larger the $\psi(t)$ range, the more robust the algorithm is.

	$\psi(t)\pm$							
dB	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**
10	*199.96*	*199.96*	*199.96*	*199.96*	*199.96*	*199.96*	199.87	199.82
9	*199.96*	*199.96*	*199.96*	*199.96*	*199.96*	*199.96*	199.83	199.70
8	*199.96*	*199.96*	*199.96*	*199.96*	*199.96*	*199.96*	199.56	199.39
7	199.92	199.92	199.92	199.92	199.87	*199.96*	199.13	198.87
6	199.65	199.70	199.65	199.70	199.65	*199.74*	198.65	198.35
5	*199.52*	*199.52*	*199.52*	*199.52*	199.48	*199.52*	197.89	197.72
4	198.91	*198.96*	198.87	198.91	198.78	198.91	196.96	196.59
3	197.89	*198.10*	197.84	197.93	197.72	*198.10*	195.94	195.21
2	196.09	*196.25*	196.01	196.17	195.85	*196.25*	194.37	192.70
1	194.13	194.13	194.01	194.13	193.86	*194.41*	192.36	191.02
0	*192.10*	*192.10*	191.91	192.06	191.58	191.80	189.63	189.48
-1	*189.93*	*189.93*	189.64	189.72	189.22	189.29	186.78	186.08
-2	187.56	*187.62*	187.32	187.54	186.95	187.02	182.70	182.46
-3	183.76	*183.81*	183.54	183.72	183.14	183.73	178.72	179.49
-4	179.84	*179.90*	179.50	179.78	179.22	179.75	174.67	175.51
-5	175.21	*175.29*	175.01	175.21	174.25	175.19	170.48	171.64
-6	170.37	*170.89*	169.92	170.48	168.88	169.61	163.68	164.75
-7	162.20	161.99	161.73	162.01	160.87	*162.21*	153.42	155.95
-8	153.45	*154.04*	152.75	153.41	152.27	153.23	142.66	144.02
-9	146.22	*146.47*	145.88	146.20	144.77	144.25	131.80	131.48
-10	137.98	*138.02*	137.58	137.67	136.73	135.38	120.31	119.30

Table 5.6: Effects of $\psi(t)$ ranges on noise robustness

Since it was seen in Section 5.3.1 that the most accurate beat-detection for the *EMLC* implementation was for $\psi(t) \pm 2$, this range was continued to the noise-test as well. In Figure 5.7, noise robustness results can be observed for three different implementations of the beat-detection algorithms.

EMLC Optimized outperforms the previous implementations up until extreme levels of noise (-8 dB).

5.3.3 Execution Time and Memory Footprint

In this section, the results after optimizing the algorithm for execution speed on the MSP430 are presented.

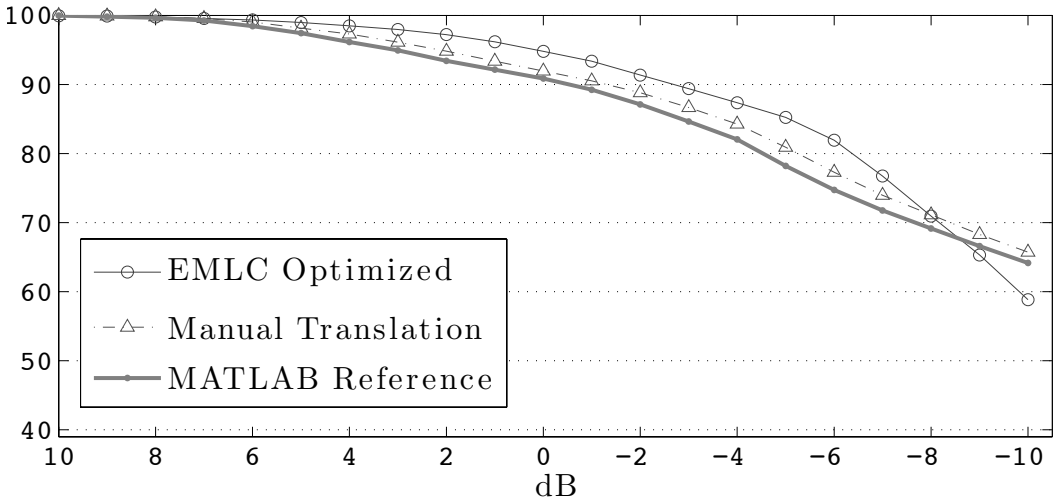


Figure 5.7: SNR Comparison of Different Best-Effort Heartbeat Detection Implementations

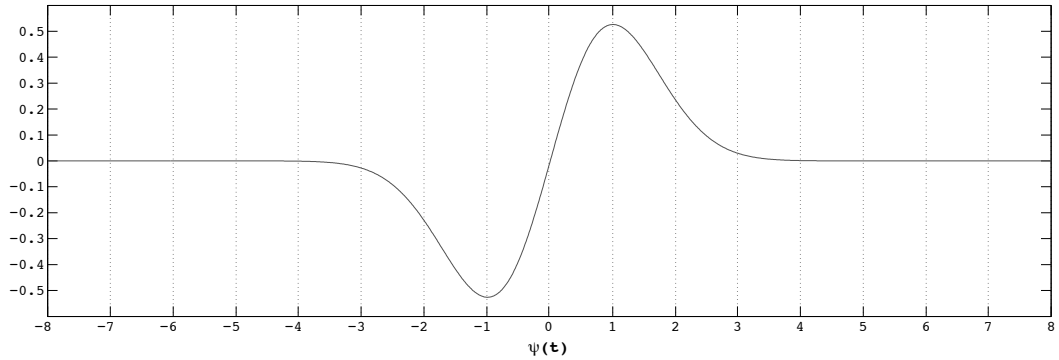


Figure 5.8: Segmentation effect of different lengths of $\psi(t)$ on the Mother wavelet

$\psi(t)\pm$	**8**	**7**	**6**	**5**	**4**	**3**	***2***	**1**
coeff	49	43	37	31	25	19	*13*	7

Table 5.7: The number of convolution coefficients for ranges of $\psi(t)$ for $f_s = 198$ Hz

Selecting CWT coefficients

Different sampling lengths of $\psi(t)$ in the CWT algorithm were evaluated in order to reduce the number of coefficients used in the convolution. Changing this length will reduce the number of coefficients used in the convolution algorithm, as seen in Table 5.7. The effect on the sampling of the Mother wavelet by varying the $\pm$ range of $\psi(t)$ can be seen in Figure 5.8. The calculated results on the MIT-BIH and imec databases for varying the length of $\psi(t)$ can be found in Table 5.8.

Optimizing the Convolution algorithm

In Table 5.9 three different implementations of the convolution function are compared in terms of clock cycles (CC), code memory (ROM, kB) and data memory (RAM, kB). All three implementations are compared against different ranges of $\psi(t)$, as discussed in the previous section. The first studied subject is MATLAB's Fixed-Point Toolbox implementation of the convolution algorithm, denoted as `conv`. The next two subjects are the unrolled and non-unrolled implementation of the partitioned convolution algorithm, denoted as `imec_conv`. It should be noted that the numerical accuracy and the RAM consumption of the three different implementations are identical.

In Figure 5.9 the ratio of clock cycles versus ROM consumption of the results found in Table 5.9 are compared. It can be seen that the unrolled version of the convolution algorithm, previously described in Section 5.2.4, provides the best trade-off between execution speed and ROM usage for all tested values of $\psi(t)$. There is a sudden jump in ROM consumption for the built-in convolution algorithm for $\psi(t)\pm$ 2; here, the MATLAB code generator automatically unrolls part of the algorithm, leading to a steep increase in ROM usage and decrease in clock cycles spent.

From the outcome of this experiment, it was decided to use a $\psi(t)$ range of ± 2 for the CWT as it gave the highest combined score for the MIT-BIH database as seen in the previous section. With this change, the number of coefficients in the convolution was reduced from 49 to 13, thus greatly reducing the number of multiplications needed. As the most computationally efficient implementation, the unrolled convolution algorithm was chosen in the final design.

In Table 5.1 it was stated that the algorithm should be optimized for RAM consumption, as the target platform had much more ROM than required, while RAM was scarce. If there were a stringent constraint on ROM usage, for example due to many more algorithms running on the same micro-controller, the built-in algorithm would be a better choice as it is much more ROM efficient than the other implementations.

	MIT-BIH			**imec database**		
$\psi(t)\pm$	Se	PP	Se+PP	Se	PP	Se+PP
8	99.21	99.84	199.05	99.98	99.96	199.94
7	99.21	*99.85*	199.06	99.98	99.97	199.95
6	99.24	99.84	199.08	99.98	99.96	199.94
5	99.21	*99.85*	199.06	99.98	99.96	199.94
4	99.26	*99.85*	199.11	99.98	99.96	199.94
3	99.28	*99.85*	199.13	99.98	99.96	199.94
2	*99.65*	99.81	*199.46*	*99.99*	99.97	199.96
1	99.64	99.67	199.31	*99.99*	*99.98*	*199.97*

Table 5.8: The effect of different ranges of $\psi(t)\pm$ on QRS benchmark

	conv		Unrolled imec_conv		imec_conv		All
$\psi(t)\pm$	CC	ROM	CC	ROM	CC	ROM	RAM
8	1 249 093	3 386	949 887	5 658	981 968	4 250	2 619
7	1 106 371	3 386	753 550	5 466	872 762	4 250	2 607
6	1 065 234	3 386	659 110	5 274	762 881	4 250	2 595
5	820 927	3 386	562 672	5 078	652 331	4 250	2 583
4	678 205	3 386	467 599	4 886	541 115	4 250	2 571
3	637 174	3 386	372 214	4 694	429 227	4 250	2 559
2	392 761	3 386	276 514	4 502	316 664	4 250	2 547
1	255 375	3 952	195 541	4 252	227 331	4 166	2 535

Table 5.9: Benchmark of three convolution algorithm implementations

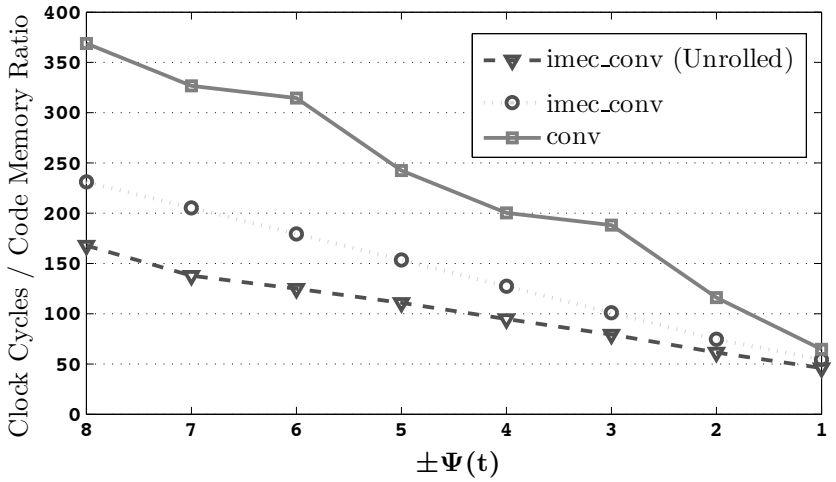


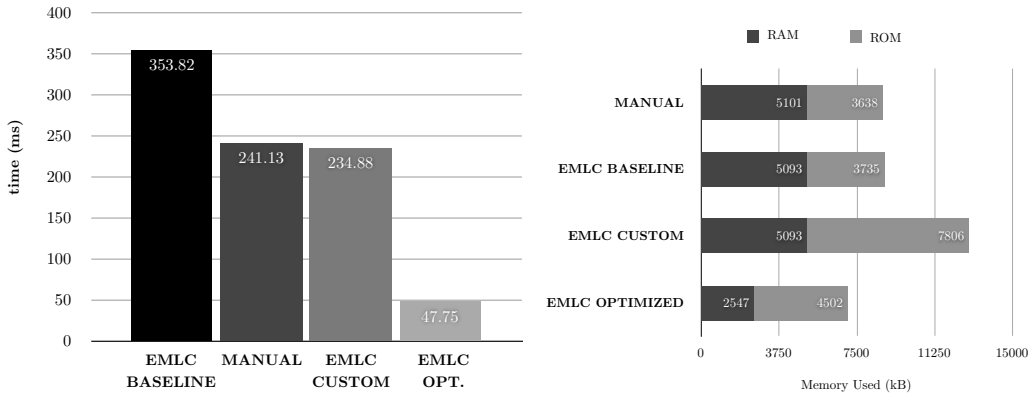
Figure 5.9: (Clock Cycle / ROM Size) ratio on convolution implementations vs. $\psi(t)$

Comparison of automatic and manual translation

In this section, a common ground is provided for comparing different implementations of the heartbeat detection algorithm. As the manual C translation uses 48 CWT coefficients, a $\psi(t) \pm 8$ was chosen in the EML implementation as the benchmark; as can be seen from Table 5.7 this is 1 less coefficient than what is used by the MATLAB algorithm, thus theoretically providing a slight computational advantage to the manual C implementation.

The manual translation uses 32-bit coefficients to calculate the CWT, thus it was necessary to update the fixed-point definitions from the initial 16-bit coefficients in the Embedded MATLAB algorithm and subsequently generate new C code in order to make a fair comparison. As data had previously been logged with the *FixIT* tool for the CWT, new 32-bit fixed-point C code could easily be generated by changing the target platform for code generation.

The manually translated algorithm used 32-bit coefficients mainly due to difficulty in



(a) Execution Time (ms)

(b) Memory Usage (bytes)

Figure 5.10: A comparison of Heartbeat Detection implementations

achieving acceptable numerical accuracy using 16-bit coefficients. This again shows some of the benefit and importance of an automated fixed-point conversion tool.

Besides the 32-bit coefficients, the manually and automatically translated algorithms differ only slightly at certain areas, but this can be considered negligible in this case as the CWT is by far the most complex and expensive part of the algorithm.

Alongside the **MANUAL** translation, three implementations of the algorithm are compared; first an implementation that uses the built-in Embedded MATLAB library as much as possible, this is denoted as **EMLC Baseline**. Secondly, an implementation using a custom developed function library, optimized for speed and using a $\psi(t)$ range of $\pm$ 8, denoted as **EMLC Custom**. Third, the **EMLC Optimized**, implementation, which is the same as EMLC Custom, except for a $\psi(t)$ range of $\pm$ 2.

In Figure 5.10 (a), the different implementations are compared in terms of execution speed. It can be seen that the manual translation and the custom version are comparable, with the custom version proving a slight edge. The EMLC Baseline version is substantially slower than the manual and custom version, showing that built-in functions cannot be relied upon for creating efficient code.

Furthermore, the results of comparing the memory consumption between the different implementations can be seen in Figure 5.10 (b).

The final implementation, *EMLC Optimized*, significantly outperforms the other implementations in execution speed, at the expense of more code memory.

5.3.4 Simulation speeds

The execution speed of different simulations of the heartbeat detection algorithms are compared in Figure 5.11.

It can be seen that the original and EML floating-point implementations are both very fast, and the compiled fixed-point code is not far behind. Some overhead due to the MAT-

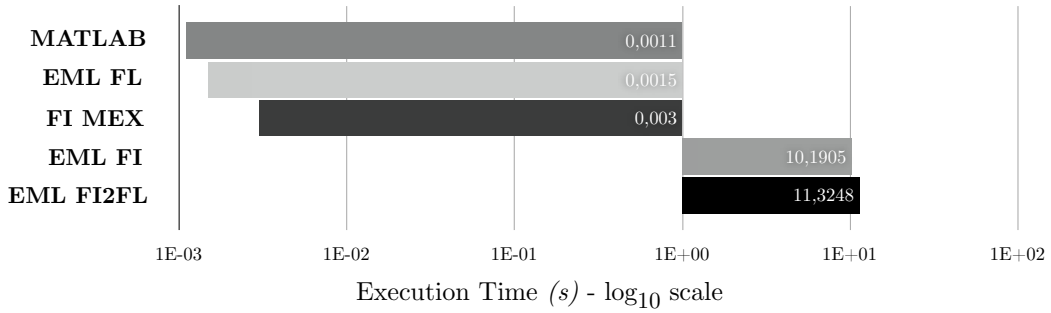


Figure 5.11: Benchmarks for different Beat Detection implementations

Legend:

MATLAB: Unmodified MATLAB algorithm

EML FL: Embedded MATLAB, Floating-Point

FI MEX: Embedded MATLAB, Compiled Fixed-Point

EML FI: Embedded MATLAB, Fixed-Point

EML FI2FL: Embedded MATLAB, Fixed-Point with Floating-Point override.

LAB interface is introduced in the compiled MEX file. The fixed-point simulations in MATLAB are several orders of magnitude slower than their floating-point equivalents. Finally, the option of running simulations with `DataTypeOverride` set to `TrueDoubles` was examined; this overrides all fixed-point calculations and makes them calculate their results in MATLAB's native `double` data type instead of in fixed-point. Unfortunately, this proved to execute even slower than the original fixed-point code.

5.4 Threats to validity

There are some limitations of this case study and workflow that should be discussed.

First, as the manual translation and the automatic code generation workflow was carried out by different people, the results are harder to compare. The workflow should be further compared by creating an optimized C implementation written by hand.

Secondly, as the algorithm described here is of limited complexity, it is still not known how the method would scale to larger scale projects.

Thirdly, the method should also be compared to a workflow where C code is written manually, and converted into MATLAB modules via the MEX compiler, as this might yet prove to be a more natural way of working, at least for the software engineers.

Fourthly, though proven effective on the beat detection algorithm, the FixIT fixed-point conversion tool has only been used in a limited context, and requires further and more

formal validation.

Fifthly, the code-generation and FixIT workflow has only been evaluated for one specific target, the results on other targets might vary to a certain extent.

Finally, the MATLAB-to-C workflow has not yet been applied by other algorithm developers or engineers to determine the acceptance or ease-of-use of the new methods.

5.5 Summary and Discussion

In this chapter, a heart-beat detection algorithm was converted from pure MATLAB code into a form which was compatible with MATLAB-to-C code generation tools following the workflow described in Chapters 2, 3 and 4.

For the algorithm, there is clearly a trade-off between code size and execution time and between accuracy and noise robustness. An implementation has been reached that weighs execution time over code size, and accuracy over noise robustness.

One advantage of the code generation process is that parameters, such as sampling frequencies, window length and filter length, can easily be changed and new implementations can be generated for different scenarios with minimal effort. It was found that the baseline EMLC code, automatically generated from the standard MATLAB libraries, was substantially slower than the manual translated algorithm. However, this could be mitigated by writing a new and optimized MATLAB library. This showed that built-in functions cannot be relied upon for creating efficient code.

It was shown that the simulation speed slows down dramatically when the algorithm is converted to fixed-point MATLAB, especially when not using MATLAB's standard function library. This has proved to be one of main problems with the proposed workflow. After the fixed-point conversion is completed, the simulation speed becomes almost intolerable unless the code is compiled. However, if the code is compiled, the algorithm more or less becomes a black box implementation, and thus part of the advantage of modeling the algorithm in MATLAB is lost. The simulation speed results show that the method currently has some issues with scalability, unless the code can be compiled. A partial solution could be to compile sub-functions of the algorithm, while leaving areas of current interest uncompiled.

In the end, the case study showed that automatically code generated implementation of the algorithm can be comparable to manually translated algorithm when using custom MATLAB code generation libraries. Converting the algorithm from its original incarnation to the code generation compatible implementation takes significant user effort, but the results can be rewarding. The Embedded MATLAB format allows a level of design-space exploration that is difficult to achieve with a C-only implementation. The question still remains if the effort spent in conversion is worth the additional effort.

Chapter 6

Discussion

It is evident that Embedded MATLAB is not a silver bullet to instantly convert MATLAB code to efficient C code. The code generator must be taken as a tool, and not a panacea to automatically resolve any implementation issues; it does not partition the algorithm or perform anything more than embarrassingly obvious optimizations.

In the early days of C, with compilers less developed as they are today, the resulting machine code often had to be optimized by hand to reach required performance. Today, usually only the most critical sections of assembly code for embedded systems and drivers are optimized to this extent. With time, perhaps advances in compiler technology will allow development of whole systems in very-high-level languages like MATLAB. This will allow engineers to be more productive by making every line of code do more. Currently, it seems that a high level of manual optimization is required to reach production level code, but even in its current state the method still has its merits.

Furthermore, the method might prove more successful if a floating-point architecture is targeted; from the evaluation, it is evident that the most severe problems appear in the fixed-point conversion phase. For any MATLAB to C project, it is no matter what useful to follow the steps in Chapter 3. This brings the MATLAB code much closer to the expected C implementation, making it easier to compare the implementation with the 'golden standard'.

Embedded MATLAB might yet serve its purpose as a bridge between software engineers and scientists, providing a common platform for research and development. In this context, Extreme Programming principles such as pair programming might be highly successful. The researchers do not need to step outside their comfort zone of the MATLAB environment, and can be a highly valuable asset in the implementation and verification of the algorithm. A decreased threshold for the researcher to participate in the embedded implementation may benefit the quality of the result greatly as the researcher has domain knowledge that can be very difficult for software engineers to obtain or apply.

On the other hand, for many cases, project teams have even found it easier to teach domain scientists how to write software than for software engineers to learn the relevant science or engineering concepts [44].

Chapter 7

Summary, Conclusion and Future Work

7.1 Summary

In this thesis, the Embedded MATLAB (EML) subset of MATLAB and the `emlc` compiler that translates EML code to C has been studied. A workflow was formalized on how to convert MATLAB to fixed-point C code by identifying the required stages to make an existing MATLAB algorithm compliant with the Embedded MATLAB subset. An automated fixed-point conversion tool, *FixIT*, was also presented that is compatible with the code generation workflow. This tool can automatically define suitable fixed-point representations based on logged data. An important feature of the tool is that it also separates the data definitions from the algorithm, thus helping to create more platform independent algorithms. The feasibility of this method was demonstrated by a case study on the implementation of a real-time heart beat detection algorithm.

In this case study, results were presented that improved upon a previously manually translated version of the algorithm. The gains were mostly due to the new insights and flexibility due to developing in a higher-level language. However, it was also found that the efforts taken to be able to reach these results are currently very high. Embedded MATLAB introduces some accidental complexities, particularly in the interaction between fixed-point and integer classes. As a consequence, much time is wasted on solving trivial implementation problems in MATLAB of trying to match the input code with expected generated C code. Finally, it was shown that the execution speed of the MATLAB algorithm can decrease by several orders of magnitude when introducing fixed-point calculations on the host computer, and by such it can reduce the practical use and reuse of the final MATLAB code.

7.2 Conclusion

While the first stage of converting MATLAB to floating-point C was well supported by the current tools, it was shown that the following stage of converting Embedded MATLAB to

Fixed-Point Embedded MATLAB code was much more complicated. However, due to the development of the *FixIT* tool, the fixed-point conversion was simplified.

The proposed workflow, at least up until the fixed-point conversion, is very useful wether or not actual code generation will be performed; in either case, it provides a much closer representation of the final intended implementation code, making the manual conversion to C code more or less straight forward. Most importantly, during the conversion process MATLAB's flexibility, function library and excellent debug functionality can be fully exploited.

This new workflow might also pave the way towards a closer collaboration between algorithm developers and implementers, as MATLAB can be used as a common platform for both research and development. However, the method still needs to gain acceptance by both parties, and problems with the code generation tools still needs to be ironed out. Hopefully, future revisions of the `emlc` code generation tool will improve upon the issues discussed in this thesis.

It is difficult to rate the impact of having access to MATLABs vast function library for algorithm implementation, but it is undeniably a welcome supplement in the algorithm developer's tool chain.

7.3 Future Work

To further validate the workflow, a larger set of use cases should be evaluated. Also, the workflow should be introduced to other researchers and implementers to identify any specific issues that are related to each group. In this way, a workflow involving both parties can be defined.

Furthermore, for programming DSPs such as imec's CoolBio [3], fractional representations (representing values in the [-1 1] range) is often preferred as it is very efficient in combination with MAC operations. As an interesting topic of future work, by extending *FixIT* automatic fixed-point specifications for future DSP can potentially be enabled.

To understand the relationship between fixed-point calculations, observing the outputted results of the system in relation to its fixed-point definitions is needed. Also, the choice of rounding mode can have a large impact on the behavior of the signal over time, especially in for example an IIR filter. So far, *FixIT* is limited to the floor rounding mode as it is the least expensive and most common method for micro-controllers.

By optimizing every calculation individually, an automatic optimization tool will produce sub-optimal results as it loses any sense of context in evaluating the current solution. In other words, it is thinking in terms of reductionism instead of a holistic approach. To obtain a better fixed-point approximation, different rounding modes and different fixed-point definitions must be evaluated in a context. *FixIT* provides the means for obtaining this context, but the framework around it needs to be extended.

Bibliography

[1] MSP430x1xx Family Users Guide. Technical Report SLAU049F, Texas Instruments, 2006.

[2] MSP430F15x, MSP430F16x, MSP430F161x Mixed Signal Microcontroller. Technical Report SLAS368F, Texas Instruments, 2009.

[3] M. Ashouei, J. Hulzink, M. Konijnenburg, Jun Zhou, F. Duarte, A. Breeschoten, J. Huisken, J. Stuyt, H. de Groot, F. Barat, J. David, and J. Van Ginderdeuren. A voltage-scalable biomedical signal processor running ecg using 13pj/cycle at 1mhz and 0.4v. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2011 IEEE International*, pages 332 –334, feb. 2011.

[4] K. Beck. *Test-driven development: by example*. Addison-Wesley Professional, 2003.

[5] K. Beck and C. Andres. *Extreme programming explained: embrace change*. Addison-Wesley Professional, 2004.

[6] A. Bertolino. Software testing research: Achievements, challenges, dreams. In *2007 Future of Software Engineering*, pages 85–103. IEEE Computer Society, 2007.

[7] L. Bierl. The MSP430 Hardware Multiplier Function and Applications. Technical report, Texas Instruments, September 1999.

[8] J.C. Carver, R.P. Kendall, S.E. Squires, and D.E. Post. Software development environments for scientific and engineering software: A series of case studies. In *Proceedings of the 29th international conference on Software Engineering*, pages 550–559. IEEE Computer Society, 2007.

[9] M. L. Chang and S. Hauck. Précis: A design-time precision analysis tool. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 229–238. Citeseer, 2002.

[10] A. G. M. Cilio and H. Corporaal. Floating Point to Fixed Point Conversion of C Code. In *Compiler construction: 8th International Conference, CC'99, held as part*

of the Joint European Conferences on Theory and Practice of Software, ETAPS'99, Amsterdam, The Netherlands, March 22-28, 1999: proceedings, page 229. Springer Verlag, 1999.

[11] A. Cockburn. Using Both Incremental and Iterative Development. *STSC CrossTalk (USAF Software Technology Support Center)*, 21(5):27–30, 2008.

[12] S. L. Eddins. Automated Software Testing for Matlab. *Computing in Science & Engineering*, 11(6):48–55, November/December 2009.

[13] E. Evans. *Domain-driven design: Tackling complexity in the heart of software*. Longman, 2004.

[14] B. Gyselinckx, C. Van Hoof, and S. Donnay. Body area networks: the ascent of autonomous wireless microsystems. *Philips Research Book Series*, Volume 5:73–83, 2006.

[15] K. Han. *Fixed-point Transformations for Low-Power Embedded Hardware and Software Design*. PhD thesis, University of Texas at Austin, August 2006.

[16] J. E. Hannay, C. MacLeod, J. Singer, H. P. Langtangen, D. Pfahl, and G. Wilson. How do scientists develop and use scientific software? In *Proceedings of the 2009 ICSE Workshop on Software Engineering for Computational Science and Engineering*, pages 1–8. IEEE Computer Society, 2009.

[17] J. L. Hennessy, D. A. Patterson, and D. Goldberg. *Computer architecture: a quantitative approach*. Morgan Kaufmann, 2003.

[18] B.W. Kernighan and D.M. Ritchie. *The C programming language*. Prentice Hall, 1988.

[19] S. Kim, K. I. Kum, and W. Sung. Fixed-Point Optimization Utility for C and C Based Digital Signal Processing Programs. *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, 45(11):1455, 1998.

[20] B. U. Köhler, C. Hennig, and R. Orglmeister. The principles of software QRS detection. *IEEE Engineering in Medicine and Biology Magazine*, 21(1):42–57, 2002.

[21] I. R. Legarreta, P. S. Addison, N. R. Grubb, G. R. Clegg, C. E. Robertson, and J. N. Watson. A Comparison of Continuous Wavelet Transform and Modulus Maxima Analysis of Characteristic ECG Features. *Computers in Cardiology*, 32:755–758, 2005.

[22] G. Martin and G. Smith. High-Level Synthesis: Past, Present, and Future. *IEEE Design & Test of Computers*, 26(4):18–25, July/August 2009.

[23] J. H. McClellan, R. W. Schafer, and M. A. Yoder. *Signal processing first*. Prentice Hall, 2003.

[24] D. Menard, D. Chillet, F. Charot, and O. Sentieys. Automatic floating-point to fixed-point conversion for DSP code generation. In *Proceedings of the 2002 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 270–276. ACM, 2002.

[25] G. B. Moody and R. G. Mark. The MIT-BIH arrhythmia database on CD-ROM and software for use with it. *Computers in Cardiology*, 17:185–188, 1990.

[26] G. B. Moody and R. G. Mark. The impact of the MIT-BIH arrhythmia database. *IEEE engineering in medicine and biology magazine: the quarterly magazine of the Engineering in Medicine & Biology Society*, 20(3):45, 2001.

[27] M. Müellegger. Evaluation of Compilers for MATLAB-to C-Code Translation. Master's thesis, Högskolan i Halmstad/Sektionen för Informationsvetenskap, Data-och Elektroteknik (IDE), 2008.

[28] J. Penders, B. Grundlehner, R. Vullers, and B. Gyselinckx. Potential and challenges of body area networks for affective human computer interaction. In *FAC '09: Proceedings of the 5th International Conference on Foundations of Augmented Cognition. Neuroergonomics and Operational Neuroscience*, pages 202–211, Berlin, Heidelberg, 2009. Springer-Verlag.

[29] J. Penders, B. Gyselinckx, R. Vullers, O. Rousseaux, M. Berekovic, M. De Nil, C. van Hoof, J. Ryckaert, R. Yazicioglu, and P. et al. Fiorini. Human++: Emerging technology for body area networks. *VLSI-SoC: Research Trends in VLSI and Systems on Chip*, 249:377–397, 2008.

[30] J. Radatz. IEEE Standard Glossary of Software Engineering Terminology, 1990.

[31] I. Romero, B. Grundlehner, and J. Penders. Robust beat detector for ambulatory cardiac monitoring. In *Annual International Conference of the IEEE Engineering in Medicine and Biology Society, 2009. EMBC 2009*, pages 950–953, 2009.

[32] I. Romero, B. Grundlehner, J. Penders, J. Huisken, and Y. H. Yassin. Low-power robust beat detection in ambulatory cardiac monitoring. In *IEEE Biomedical Circuits and Systems Conference, 2009. BioCAS 2009*, pages 249–252, 2009.

[33] J. Ronkainen and P. Abrahamsson. Software Development under Stringent Hardware Constraints: Do Agile Methods Have a Chance? In *Extreme Programming and Agile Processes in Software Engineering*, volume 2675 of *Lecture Notes in Computer Science*, pages 73–79. Springer Berlin / Heidelberg, 2003.

[34] J. Segal. When software engineers met research scientists: A case study. *Empirical Software Engineering*, 10(4):517–536, 2005.

[35] J. Segal. Some challenges facing software engineers developing software for scientists. In *Proceedings of the 2009 ICSE Workshop on Software Engineering for Computational Science and Engineering*, pages 9–14. IEEE Computer Society, 2009.

[36] J. Segal and C. Morris. Developing scientific software. *IEEE Software*, 25(4):18–20, 2008.

[37] C. Shi. Statistical method for floating-point to fixed-point conversion. Master's thesis, University of California, Berkeley, 2002.

[38] The Mathworks. *Embedded MATLAB User Guide*, 2009b edition, 2009.

[39] The Mathworks. *Fixed-Point Toolbox 3 User Guide*, 2009b edition, 2009.

[40] The Mathworks. *Real-Time Workshop 7 User Guide*, 2010a edition, 2010.

[41] EE Times. MathWorks rolls MATLAB C generator. `http://www.eetimes.com/electronics-products/fpga-pld-products/4099287/MathWorks-rolls-MATLAB-C-generator`, October 2007. Accessed July 2010.

[42] A. Vikström. A study of automatic translation of MATLAB code to C code using software from the MathWorks. Master's thesis, Luleå tekniska universitet, 2009.

[43] G. V. Wilson. Where's the real bottleneck in scientific computing? *American Scientist*, 94(1):5, 2006.

[44] G. V. Wilson, R. H. Landau, and S. McConnell. What should computer scientists teach to physical scientists and engineers? *Computational Science Engineering, IEEE*, 3(2):46 – 65, Summer 1996.

[45] J. Yang, F. Zhou, X. Liu, and L. Shi. A novel method for the float-point to fixed-point conversion from statistical perspective. *International journal of electronics*, 95(4-6):319–332, 2008.

[46] R. F. Yazicioglu, P. Merken, R. Puers, and C. Van Hoof. A 60μ W 60 nV/$\sqrt{}$ Hz Readout Front-End for Portable Biopotential Acquisition Systems. *IEEE Journal of Solid-State Circuits*, 42(5):1100–1110, 2007.

[47] H. Zarrinkoub and G. Martin. Best practices for a matlab to c workflow using real-time workshop. Matlab digest, The Mathworks, `http://www.mathworks.com/company/newsletters/digest/2009/nov/matlab-to-c.html`, November 2009. Accessed December 2009.

Appendix A

Glossary

In this appendix we give an overview of frequently used terms and abbreviations.

ADC: Analog-to-Digital-Converter

ALU: Arithmetic Logic Unit; a part of the CPU that performs calculations and logical operations.

ASIC: Application-Specific Integrated Circuit.

BAN: *Body Area Network*, a network consisting of a series of small sensor nodes that are communicating with each other and worn on the body for various applications, including health monitoring.

C: A programming language popular for embedded systems and systems programming.

CWT: Continuos Wavelet Transform; transforms a signal from time domain to wavelet domain.

DataInspector: A data logging tool for numerical analysis and visualization.

DSP: Digital Signal Processor.

ECG: *Electrocardiogram*.

Embedded MATLAB: A subset of the MATLAB language that can be used in automatic code generation.

EML: See *Embedded MATLAB*.

EMLC: The Embedded MATLAB compiler; generates C code from Embedded MATLAB code.

FFT: Fast Fourier Transform; An efficient signal processing algorithm used to transform a signal into frequency domain.

FI: See Fixed-Point.

fimath: The global or local fixed-point arithmetic rules for MATLAB's fixed-point object.

fipref: Fixed-Point Preferences. The configuration of fixed-point behavior in MATLAB.

Fixed-Point: A Fixed-Point representation is a data type for representing a real number with a fixed number of digits before or after the radix point.

FixIT: *FIXed-Point Integration Tool* A tool that gathers data from the DataInspector, funnels it through the NumericAdvisor, and generates data definition files that can be integrated into the original `.m` files.

FL: See Floating-Point.

Floating-Point: A Floating-Point representation is a data type for representing a real number using a **M**antissa, an **E**xponent and a **R**adix (usually 2 or 10). The number is represented by $M \times E^R$.

FPU: *Floating-Point Unit*, a co-processor for floating point arithmetic.

Holst Centre: An open-innovation initiative between TNO and imec in Eindhoven, the Netherlands.

Human++: A technology integration program at *imec*.

imec: Europe's leading nano-technology company, headquartered in Leuven, Belgium with offices and over 1 800 employees world-wide.

Kurtosis: A measure of how outlier-prone a distribution is. Definition: $k = \frac{E(x-\mu)^4}{\sigma^4}$

LFHF: Low-Frequency-High-Frequency ratio.

LOC: *Lines Of Code*.

MAC: Multiply-and-Accumulate. An operation heavily used in signal processing.

Mathworks, The: The company that produced MATLAB.

MATLAB: ***Ma**trix **Lab**oratory*, a programming environment by The MathWorks for writing scientific software.

MCS: MATLAB-to-C Synthesis. A MATLAB-to-C code generation product by Catalyst (later: Agility Design Solutions).

MEX: *MATLAB **Ex**ecutable*. Compiled C code with a MATLAB API.

MIT-BIH: A database of manually annotated ECG recordings which is considered a *de facto* standard in medical research.

NumericAdvisor: A tool developed in this thesis for automatically determining a suitable fixed-point data type based on input data and target hardware.

numerictype MATLAB's definition object for fixed-point data type properties.

Overflow: Overflows occur when a number is greater than the maximum representable number within the finite range of the used data type.

pdf: The *Probability Density Function* describes the relative likelihood for a random variable to occur at a given point in the observation space.

QRS: The QRS complex is a recording of a single heartbeat on an ECG that corresponds to the depolarization of the right and left ventricles.

R-peak: See Figure 5.2.

RV: Random-Variable

Skewness: A measure of the asymmetry of the data around the sample mean. Definition: $s = \frac{E(x-\mu)^3}{\sigma^3}$

Underflow: Underflows occur when a number is less than the minimum representable number within the finite range of the used data type.

WSN: Wireless Sensor Network

Ålesund: A beautiful city on the west coast of Norway, and the author's hometown.

Appendix B

Related Work

B.1 Algorithm Development

Carver *et al.* [8] states that an unique characteristic of scientific and engineering software, that separates them from their commercial counterparts, is that the requirements are sprung from a discovery and gathering process. While most projects are based on the underlying, fixed, physical laws—the application of the laws to a given problem is often unknown at the beginning of the project. A second distinguishing characteristic is that the main force behind these projects is to ensure that the theory is accurately replicated, rather than focusing on sound software engineering practices.

B.1.1 How algorithm designers develop software

Advances in science and engineering rely increasingly on the results produced by scientific software. According to a recent study [16] scientists and engineers acquire the knowledge required to develop and use scientific software primarily from peers and through self-study rather than from formal education and training. Most algorithms start out as *proof-of-concept* prototypes developed by scientists. These algorithms might require substantial engineering efforts to reach production quality level code [35].

Many traditional software methodologies impose constraints that are contradictory to the way research scientists work [34]. Because many of the projects are performing new science and cannot in a detailed manner be known in advance, operating in a rigid plan-driven style is neither feasible nor productive [8]. Actually, many scientists tend to think they will have greater flexibility by not following rigid software development processes [8]. A persistent development process for researchers developing algorithms and scientific software is shown in Figure B.1. In most cases, scientists develop software to produce *immediate* results for their research [35], which yields trial-and-error development and thus they are working in experimental increments. Hence, the common scientific development method is akin to agile development processes, defined amongst others by Beck [5].

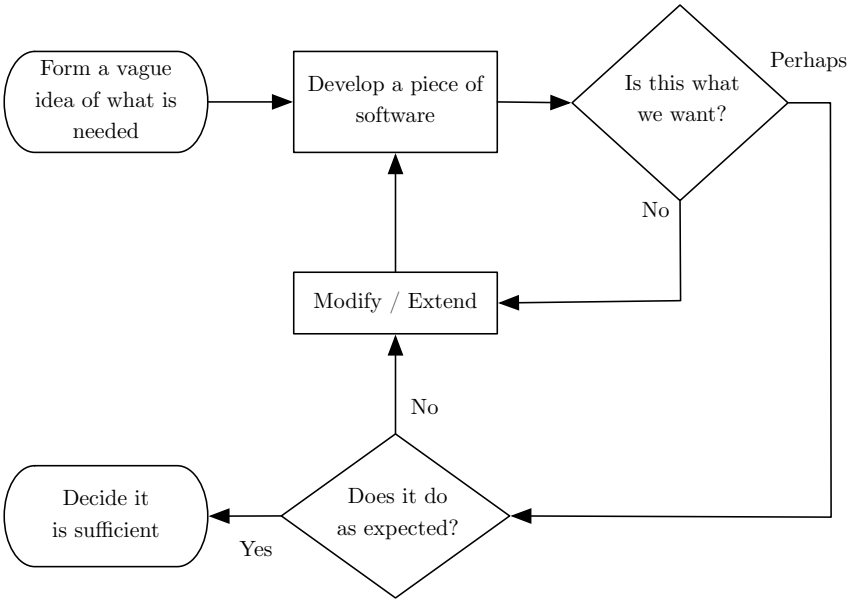


Figure B.1: A model of software development performed by scientists

Adapted from Segal [36]

B.2 Automatic Code Generation

Currently, code generation is gaining popularity and is supported by an increasing amount of numerical computing languages. In this section, an overview of code generation software and its potential benefits and its challenges are given.

B.2.1 An Overview of Code Generation Software

An overview of the code generation options for a selection of popular proprietary numerical computing languages, found in the algorithmic development community, can be seen in Table B.1.

From the overview, it can be seen that National Instruments' *LabView*, Maplesoft's *Maple*, The Mathworks' *MATLAB* and Wolfram Research's *Mathematica* all offer code generation functionality to different degrees. C and C++ enjoys the most widespread support for code generation, which is natural considering their prevalent use and acceptance in the industry. *Maple* stands out from its competitors by having the ability to generate code for six different target languages, including MATLAB. On the other hand, simply having the ability to generate code for different languages does not necessarily say anything about the ease of development in that environment, nor about the quality of the generated code.

	Target Language					
Source Language	C	C++	FORTRAN	Java	MATLAB	Visual Basic
LabVIEW	✓	✓				
Maple	✓	✓	✓	✓	✓	✓
MATLAB	✓	✓	✓*		—	
Mathematica		✓*	✓*			

Table B.1: Code Generation options for several popular algorithm development languages

* *Possible through third-party code generators*

B.2.2 MATLAB-to-C Code Generation

There is not much research done yet on the new code generation abilities of MATLAB, except for The MathWork's own documentation, which is in fact also severely limited (in quality, not quantity). However, there are two other master theses of interest; Vikström [42] presents a qualitative investigation of the code generation capabilities of `emlc`. He performs a small-scale study of the readability of the generated C code as compared to the original MATLAB code amongst the engineers at AutoLiv. He also notes that

> "*Somewhere there is a limit when it would be more practical to implement an algorithm directly in C instead of trying to write C-like Embedded MATLAB code.*"

Müellegger, [27], evaluates MathWork's *Embedded MATLAB C* (`emlc`,2007) compiler and the MATLAB-to-C Synthesis (`MCS`, 2006) compiler. He covers three aspects of automatic code generation; (1) generation of reference code, (2) target code generation and (3) floating-to-fixed-point conversion.

His conclusions are that `emlc` is more suitable for efficient embedded C code generation, while `MCS` is more suitable for creating code to be run on the development platform due to extended support of MATLAB functions. `emlc` allocates memory statically, thus increasing the real-time reliability of the system. Müellegger also states that the functional correctness in both compilers is generally achieved with both compilers. He also notes that these compilers are still in their infancy, and it is expected that the results of this evaluation will not be valid for long as development progresses. In fact, since Müellegger's evaluation, MCS is no longer available and has been acquired by Mentor Graphics [22].

B.3 Automatic Fixed-Point Conversion

In [19], Kim *et al.* describes the creation of a fixed-point optimization utility for C/C++ based on statistics derived from input data. The method of logging and selecting the optimal fixed-point definition does not significantly differ from the method described in this thesis. The difference lies in the method of calculating the range *R(x)*, which is based on the mean, variance, skewness and kurtosis. Depending on the distribution of the signal which is

characterized as uni-/multi-modal, (non-)symmetric or (non-)zero mean, different statistical formulas are applied to calculate *R(x)*.

In his dissertation, Han [15] introduces automatic code generation for fixed-point in MATLAB, however the code generated violates Embedded MATLAB syntax, which was introduced the following year. Furthermore, the method embeds the data type directly in the algorithm, which breaks the desire of decoupling the data type from the algorithm. The dissertation focuses on word length optimization for hardware implementation, and word length reduction for lowering power consumption using simulation based search and genetic algorithms. These methods seem the most promising for the intended purpose in this thesis, and will be investigated further in the future.