Holistic Framework for Human-in-the-Loop Cyber-Physical Systems using Body/Brain-Computer Interfaces

A Dissertation Presented

by

Shen Feng

to

The Department of Electrical and Computer Engineering

in partial fulfillment of the requirements
for the degree of

Doctor of Philosophy

in

Computer Engineering

Northeastern University
Boston, Massachusetts

April 2017
This thesis is dedicated to my wife, Naili, who has been encouraging and supportive throughout my Ph.D. journey. I also dedicate this thesis to my parents and sister who always love me. And this thesis is also dedicated to you as a reader.
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>List of Figures</td>
<td>vi</td>
</tr>
<tr>
<td>List of Tables</td>
<td>viii</td>
</tr>
<tr>
<td>List of Acronyms</td>
<td>ix</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>x</td>
</tr>
<tr>
<td>Abstract of the Dissertation</td>
<td>xi</td>
</tr>
<tr>
<td><strong>1 Introduction</strong></td>
<td>1</td>
</tr>
<tr>
<td>1.1 Multi-disciplinary Challenges for Assistive BBCI Development</td>
<td>2</td>
</tr>
<tr>
<td>1.1.1 Reliable Intent Estimation from Physiological Data</td>
<td>3</td>
</tr>
<tr>
<td>1.1.2 Interfacing with Various BBCI Hardware</td>
<td>3</td>
</tr>
<tr>
<td>1.1.3 Deployment to Portable Devices</td>
<td>4</td>
</tr>
<tr>
<td>1.1.4 Availability of Affordable Hardware</td>
<td>4</td>
</tr>
<tr>
<td>1.2 Problem Definition</td>
<td>5</td>
</tr>
<tr>
<td>1.3 Dissertation Overview</td>
<td>6</td>
</tr>
<tr>
<td>1.4 Contributions</td>
<td>8</td>
</tr>
<tr>
<td><strong>2 Background</strong></td>
<td>11</td>
</tr>
<tr>
<td>2.1 Cyber-Physical Systems</td>
<td>11</td>
</tr>
<tr>
<td>2.1.1 Human-in-the-Loop Cyber-Physical Systems</td>
<td>12</td>
</tr>
<tr>
<td>2.2 Model Based Design</td>
<td>14</td>
</tr>
<tr>
<td>2.3 Related Work</td>
<td>16</td>
</tr>
<tr>
<td>2.3.1 Software Development of Assistive BBCI</td>
<td>16</td>
</tr>
<tr>
<td>2.3.2 Human Interface Devices</td>
<td>17</td>
</tr>
<tr>
<td><strong>3 Holistic BAT Framework</strong></td>
<td>20</td>
</tr>
<tr>
<td>3.1 Assistive BBCI Application Abstraction</td>
<td>20</td>
</tr>
<tr>
<td>3.2 Framework Systems</td>
<td>22</td>
</tr>
<tr>
<td>3.2.1 Human Interface</td>
<td>22</td>
</tr>
<tr>
<td>3.2.2 Computer Interface</td>
<td>22</td>
</tr>
<tr>
<td>3.2.3 Action System</td>
<td>23</td>
</tr>
</tbody>
</table>
List of Figures

1.1 BBCI-based Assistive Technology Overview ........................................... 2
1.2 BAT Framework using Model-Based Design ............................................. 6

2.1 Cyber-Physical System (CPS) ................................................................. 11
2.2 Human-in-the-Loop Cyber-Physical System (HiLCPS) ............................. 12
2.3 HiLCPS Classification ........................................................................... 13
2.4 Industrial Design Process using Model-Based Design [1] ......................... 15

3.1 Assistive BBCI Applications ................................................................. 21
3.2 BAT Framework .................................................................................. 22
3.3 Framework Publisher-Subscriber Communication Paradigm ................... 26
3.4 Application Communication Layer Using DDS ....................................... 27

4.1 Embedded Assistive Application Design Flow using MBD ....................... 30
4.2 Hardware Device Categorization (Class, Type, Instance) ....................... 31
4.3 DevClass ........................................................................................... 32
4.4 Location Transparent Access to Distributed HW ........................................ 34
4.5 Packet Frame of DAQ TCP Proxy ......................................................... 35
4.6 Software Stack of Prototyped Assistive APP ........................................... 36

5.1 Hardware Diversity in Assistive Applications ......................................... 39
5.2 BODE Suite Overview ........................................................................... 40
5.3 EEGu2 .................................................................................................. 41
5.4 Architecture Overview ......................................................................... 42
5.5 EEGu2 DAQ Cape ............................................................................... 43
5.6 Power Board ....................................................................................... 44
5.7 Power and Signal Isolation ..................................................................... 45
5.8 EEGu2 3D Printed Enclosure .................................................................. 46
5.9 ............................................................................................................... 47
5.10 Input Referred Noise (IRN) .................................................................. 50
5.11 AFE Communication [2] ....................................................................... 51
5.12 EEGu2 AFE Real-time Analysis ........................................................... 52
5.13 Acquisition Delay due to Buffering ....................................................... 54
5.14 Kernel-PRU Driver End-to-end Delay .................................................... 55
5.15 Kernel-PRU Driver Delay vs System Utilization .................................................. 56
5.16 *StimTron: Visual and Tactile Stimulus* ............................................................... 57
5.17 *StimTron Firmware* ............................................................................................... 58
5.18 *StimTron* vs Monitor Signal Quality @8Hz .......................................................... 59
5.19 *StimTron* vs Monitor Stimulus SNR ..................................................................... 60
5.20 *PstVis: Visual Presentation* ................................................................................... 61
5.21 ................................................................................................................................. 62
5.22 Auto-scaling ............................................................................................................. 62
5.23 Rapid Embedded assistive APP Development (deployment on *EEGu2* as an example) ................................................................. 65

6.1 Brain-Controlled Wheelchair Overview .................................................................. 67
6.2 Brain-Controlled Wheelchair Field Test ................................................................... 68
6.3 System Realization ................................................................................................... 70
6.4 *DAQ: PAL and EEGu2* ........................................................................................... 71
6.5 Stimulus and Presentation ......................................................................................... 72
6.6 Example EEG trace for a single user and trial ......................................................... 73
6.7 Average accuracy and information transfer rates as a function of trial length for 2 potential classifier (maximum CCA scores and bayesian classifier with CCA features) ........................................ 74
6.8 Demo APP Execution Flow ....................................................................................... 76
6.9 Brain-Controlled Robotic Arm Application ............................................................ 78
6.10 Using *BODE Suite* for Brain-Controlled Robotic Arm ....................................... 78
6.11 BCI speller processing flow .................................................................................... 80
6.12 Calibration and testing sessions of spelling application. During calibration, the user is asked to focus on the corresponding target LED array. To spell, letters are assigned to different icon boxes before the LEDs flash. By attending one of the LED stimuli, the probability mass of the chosen letters will increase according to the classifier’s confidence. Once the probability of a letter exceeds a predefined threshold, that letter will be chosen as the user’s intent. .................................................................................... 81
6.13 Average Accuracy and Information Transfer Rate (ITR) ....................................... 82

7.1 SimSH Flow Overview ............................................................................................... 85
7.2 SimSH Flow ............................................................................................................... 86
7.3 Communication Optimization over Underutilized Bus ............................................ 88
7.4 *pack and unpack* for Concatenating N Transfers .................................................. 89
7.5 Model Split for Each PE ........................................................................................... 90
7.6 Proxy OSI Model ...................................................................................................... 91
7.7 HW/SW Synchronization .......................................................................................... 92
7.8 *Proxy Addressing* at Network Layer ...................................................................... 93
7.9 Sobel Edge Detection Algorithm ............................................................................. 95
7.10 Sobel Edge Detect Performance Estimation ........................................................... 96
7.11 Sobel-Edge-Detect Communication Optimization over Underutilized Bus .......... 96
7.12 *Proxy Network Layer Addressing* ......................................................................... 97
7.13 Sobel Edge Detect Performance ............................................................................. 98
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Other BCI DAQ Systems</td>
<td>18</td>
</tr>
<tr>
<td>3.1</td>
<td>Application Examples using BAT Framework</td>
<td>24</td>
</tr>
<tr>
<td>5.1</td>
<td>Voltage Regulation Noise and Efficiency</td>
<td>45</td>
</tr>
<tr>
<td>5.2</td>
<td>EEGu2 Configuration and Control Interface via VFS</td>
<td>48</td>
</tr>
<tr>
<td>5.3</td>
<td>EEGu2 Power Consumption</td>
<td>49</td>
</tr>
<tr>
<td>5.4</td>
<td>EEGu2 AFE Real-time Analysis</td>
<td>52</td>
</tr>
<tr>
<td>5.5</td>
<td>EEGu2 End-to-end Delay Due to Buffering</td>
<td>54</td>
</tr>
<tr>
<td>6.1</td>
<td>Demo APP Hardware Types and Connectivity</td>
<td>69</td>
</tr>
<tr>
<td>6.2</td>
<td>Application Footprint</td>
<td>75</td>
</tr>
<tr>
<td>6.3</td>
<td>Productivity Gain</td>
<td>76</td>
</tr>
<tr>
<td>6.4</td>
<td>System Utilization ($f_{sampling} = 250Hz$)</td>
<td>77</td>
</tr>
<tr>
<td>7.1</td>
<td>Timing and Dependency of Proxy</td>
<td>94</td>
</tr>
<tr>
<td>7.2</td>
<td>FPGA Utilization of HW/SW Optimized Solution</td>
<td>99</td>
</tr>
</tbody>
</table>
List of Acronyms

ATLM  Arbitrated Transaction Level Model.
BCI  Brain-Computer Interface
BBCI  Body/Brain-Computer Interface
CRC  Cyclic Redundancy Check
CPS  Cyber-Physical Systems
DAQ  Data Acquisition
DDS  Data Distribution Service.
EEG  Electroencephalography
FPGA  Field-Programmable Gate Array
assistive  Human-in-the-Loop Cyber-Physical System
IDL  Interface Definition Language
MBD  Model-Based Design
OMG  Object Management Group
RP  Rapid Prototyping
TLM  Transaction Level Model. A model of a system in which communication is described as transactions, abstract of pins and wires.
Acknowledgments

Here I wish to thank those who have supported me during the process of the thesis work. First and foremost, I want to thank my advisor, Prof. Gunar Schirner, for his patient guidance and generous support throughout my Ph.D. journey. I am grateful for our countless constructive discussions that inspires me and guides me during the research. And on top of all, I have learned from him to be a person who does things right and does the right things.

I also thank Prof. Deniz Erdogmus and Prof. Taskin Padir for serving on my committee. Their valuable discussions and comments enrich the research and improve this thesis. It’s their support that leads this multidisciplinary research to a success.

I would like to thank Fernando Quivira for his support in running experiments and his valuable feedback as a BCI researcher. I would thank Mian Tang, Xiao Zhao, Hang Zhang, Tianyu Xia, Yanfeng Wang, Guanye Luo and Yun Yao for their help in developing and improving BCI suite hardware and drivers. I also thank Tim Dyson and his Capstone15 group members for developing EEGu2, BBB-LED and BBB-LCD hardware. I thank Capstone14 and Capstone13 group for initially prototyping our DAQ. I would also thank Hao jiang for improving the hw-sw co-design toolchain.

I also thank Matt Higger, Hooman Nezamfar and Dimitry Sinyukov for their discussion in composing the holistic framework to tackle the multidisciplinary challenge.

This thesis work was influenced by the member of ESL group, through meetings and discussions. I thank those ESL members for their encouragement and their help for the work in this dissertation.

This work is supported by the National Science Foundation under grant 1136027.
Abstract of the Dissertation

Holistic Framework for Human-in-the-Loop Cyber-Physical Systems using Body/Brain-Computer Interfaces

by

Shen Feng

Doctor of Philosophy in Computer Engineering

Northeastern University, April 2017

Dr. Gunar Schirner, Adviser

Body/Brain-Computer Interface (BBCI) provides an additional communication channel between the human and the machine. Researchers have been studying BBCI-based human-in-the-loop cyber-physical systems (HiLCPs) to augment human communication and mobility, e.g. enabling 3.7 million locked-in people to communicate with their families, and restoring the independence of amputees using prostheses. However, developing and deploying these societal impacting assistive applications poses immense multidisciplinary challenges. Concerted efforts are required in the horizontal integration of components in multiple disciplines, including robotics, user interface design, biomedical signal processing, and embedded system design. Challenges also stem from the vertical integration from algorithm prototyping in a laboratory setting to embedded deployment of final products. Furthermore, utilizing BBCI technology is also challenging due to needing specialized devices for interfacing with the human. A holistic approach is needed to provide a common view of the development of assistive BBCI applications as well as coordinated human interface devices for rapid prototyping and portable deployment.

This dissertation addresses the challenges in the design, development, and deployment of various assistive BBCI applications from three aspects: horizontal integration of multidisciplinary efforts, vertical integration across various design phases, and human interface devices for BBCIs. To capture various applications, we formalize the applications and identify their common functionality. Utilizing the application abstraction, we propose a holistic BBCI-based Assistive Technology (BAT) framework providing generic modules for multiples disciplines with well-define interfaces to work together. The framework allows a modular, distribution composition and can be configured to support a wide range of applications. In addition, our framework adopts a model-based design (MBD) approach to accelerate the development of embedded assistive applications from computational modeling to
rapid prototyping and finally automated embedded deployment. Our framework unifies the development across various hardware types and connection mechanisms. Our domain-specific synthesis tool, \textit{HSyn}, empowers algorithm designers to prototype portable, hardware-agnostic applications in MATLAB while offering an automatic path to embedded deployment.

In addition to the horizontal and vertical integration challenges, specialized devices necessary to interface with the human also pose challenges in assistive BBCI development. BBCI requires high-precision acquisition of bio-signals often time-correlated with an accurate stimulus (to invoke a physiological response). Additionally, real-time embedded processing is required for portable deployment of BBCI algorithms. While individual stand-alone human interface devices are available, they are often too expensive (e.g. $15,000 \text{ g.USBamp}$) and incomplete due to the lack of embedded processing capability. Moreover, integrating these stand-alone devices into an application is tedious and error prone, unnecessarily burdening BBCI developers. To address the challenge of high device cost and integration effort requiring embedded systems knowledge, we introduce an affordable, open device (\textit{BODE}) suite tailored for BBCI and supported by hardware abstraction for rapid prototyping and development. We identified three key categories of devices interfacing with the human. Our suite provides these three interface devices in an open, affordable, portable implementation, including physiological data acquisition (\textit{EEGu2}), simulation (\textit{StimTron}) and presentation (\textit{PstVis}). Our suite is fully integrated into our framework enabling a distributed composition of devices, rapid prototyping in MATLAB, and automated deployment on any device in our suite. \textit{EEGu2} provides 16-channel, 24-bit resolution acquisition with 25dB signal-to-noise ratio and battery-powered embedded processing for mobile operation. \textit{StimTron} generates 12-channel visual and tactile stimulation with configurable bit pattern and frequency from 0.1Hz to 409.6Hz at 0.1Hz resolution. \textit{PstVis} presents visual feedback and additionally incorporates a rail car system for flexible stimulus placement.

We demonstrate the flexibility and usability of our \textit{BAT} framework and \textit{BODE Suite} with three assistive BBCI applications: a brain-controlled wheelchair, a brain-controlled robotic arm, and a BCI speller. Using our rapid design workflow, designers first prototype applications in MATLAB and test them on human subjects utilizing our plug-and-play device suite. By using \textit{HSyn}, MATLAB application prototypes are automatically synthesized to embedded implementation which can then be deployed to any device in our suite. \textit{HSyn} dramatically reduces the development time of embedded deployment by six orders of magnitude as compared to manual conversion. Our flexible framework (software), affordable suite (hardware), and rapid design workflow compose a holistic tool set for rapid assistive BBCI development.
Chapter 1

Introduction

Assistive technology enhances independence by enabling people to perform tasks that their physiologies may struggle to accomplish. Besides traditional human-machine interaction using a keyboard or a mouse, current assistive applications take advantage of Body/Brain-Computer Interfaces (BBCI) as an additional communicate channel, such as operating prosthetic devices [3], a brain-controlled robot [4], typing through virtual keyboards [5, 6, 7] and playing games [8]. In particular, BBCI can establish a communication bridge where traditional approaches fail.

An example that motivates the work in this dissertation is to assist individuals with verbal and motion impairments due to their physical paralysis, commonly classified as Locked-in-Syndrome (LIS), to restore fundamental autonomy such as mobility and communication. People affected by various levels of LIS suffer from degraded quality of life stemming from their limited ability to interact with their surroundings, communicate with their loved ones, and/or autonomously move to places. This general condition affects more than 3.7 million people worldwide covering individuals with amyotrophic lateral sclerosis (ALS), severe traumatic brain injuries, among others [9]. Moreover, some diseases are degenerative in nature, thus reducing the physical capabilities of what people can do as the disease progresses. Many assistive applications exist in this area and demonstrate a great societal impact.

Assistive BBCI applications interfaces with both the human and the physical environment. Figure 1.1 overviews assistive BBCI applications that typically contain three domains: human, cyber and physical. Users are presented with a set of stimuli, each associated with a system action. Users select the desired action by focusing on the associated stimulus. The cyber domain infers human intent based on the physiological evidence collected via a data acquisition system (DAQ). The physical domain enacts inferred action on human’s behalf through sensors and actuators. Human also observes
the physical action result as an input to make new decisions. This type of assistive applications is a subclass of Human-in-the-loop Cyber-Physical Systems (HiLCPS) \[10\] (see Section 2.1.1). In this dissertation, we focus on the BBCI-based assistive technology in the context of HiLCPS.

1.1 Multi-disciplinary Challenges for Assistive BBCI Development

Developing BBCI-based assistive applications poses immense multi-disciplinary challenges. Application components requiring specialized domain knowledge, including robotics, user interface design, biomedical signal processing, and embedded system design. Successfully designing, developing and deploying assistive solutions requires integration across disciplines: 1) BBCI with signal processing intensive algorithm, 2) embedded system design for interfacing with the human and application deployment, and 3) robotics (Cyber-Physical) that acts onto the physical environment \[11\]. However, these disciplines have very different views/experience even when looking at the same problem. For example, while BBCI experts develop algorithms focusing on functional behavior (non-real-time), embedded experts address real-time problems in the interaction with the human and physical environment. The divergence between the developers’ attitude, e.g. to real-time systems, poses a design challenge. Currently, there is no holistic view of assistive applications, which leaves developers in different fields working in isolation and potentially operating under wrong assumptions.

In addition to the multidisciplinary challenges going across all disciplines, challenges also
CHAPTER 1. INTRODUCTION

stem from developing reliable intent inference, interfacing with various hardware, the availability of affordable BBCI hardware, and embedded deployment.

1.1.1 Reliable Intent Estimation from Physiological Data

Designing and implementing reliable intent inference algorithms that extract users’ intent from noisy neurophysiological signals have been the focus of BBCI research over the past several years [12]. A problem in EEG-based BCI is that the non-invasive EEG measurement produces extremely low signal levels (µV) usually overshadowed by noise [13]. On one hand, noise can stem from ubiquitous external sources. For instance, locked-in patients are usually surrounded by a wide variety of machines such as respirators, appliances, among others, which introduce signal disturbances like 60 Hz noise from the power supply. On the other hand, noise can appear from within the body itself. Muscular (EMG) activity, for example, tends to appear over a wide frequency range, thus corrupting many of the EEG patterns needed for intent inference. With so much noise present in the systems, many trials (repetitions of the stimulus) are needed to consistently make correct decisions [14]. To detect users’ intent from a noisy physiological source, active research is ongoing to improve feature extraction and classification algorithms using the machine learning technique. These algorithms reduce the decision time as well as increase the accuracy of the system. However, to enable this research, high-quality data acquisition devices are needed to reject noise and attenuate the effects of artifacts over the signal, making inference systems more robust. While commercial devices are so expensive (thousands of dollars) that hinders researchers from participating in assistive BBCI research, low cost biomedical signal acquisition devices are required.

1.1.2 Interfacing with Various BBCI Hardware

To interface with the human, BBCI utilizes various hardware to provide stimulus to the human as well as to collect human physiological data. For example, event-related potential (ERP) based BBCIs [15] [16] presents auditory, visual, or tactile stimulus to elicit a P300 response [17], which is a positive deflection in centro-parietal scalp voltage (measured via EEG) with a typical latency just over 300ms. Researchers also utilize electromyography (EMG) signals from muscles to control prosthetic limbs and restore mobility of amputees [18]. Another example is visually evoked potential (VEP) based BBCIs such as Steady State VEP (SSVEP) [19] and code-modulated VEP (c-VEP) [20] [21] that display flickering light (stimulus) and measure invoked VEP (EEG) from
CHAPTER 1. INTRODUCTION

visual cortex. Using BBCI to extract human intent is challenging due to needing interfacing with specialized hardware.

Interfacing with hardware devices through manufacturer-specific APIs is challenging due to requiring huge integration efforts and necessary embedded systems knowledge. Additional challenges stem from interfacing with various hardware components with different specifications. Hardware evolves with technology and varies in realization to meet distinct optimization goals, such as small size, low power, and high performance; however, developing an application using manufacturer-specific APIs hinders the application’s evolution and growth with the state-of-the-art hardware. Overall, interfacing with various BBCI devices poses challenges to the assistive application development.

1.1.3 Deployment to Portable Devices

Deploying assistive applications to a portable, embedded setting is also challenging. Algorithm designers prototype applications in high-level development environment, such as Octave [22], MATLAB [23], and Scilab [24]. However, most assistive products require an embedded deployment to portable, low-power platforms to assist users in their daily lives. The gap between high-level algorithm development and low-level embedded realization challenges algorithm designers requiring a significant amount of manpower and embedded knowledge for the manual conversion. Traditionally, algorithm designers first develop high-level algorithms and then embedded developers realize the embedded implementation of the algorithms. However, this sequential approach isolates the development in one discipline (e.g. the algorithm design) from another (e.g. embedded realization). This isolation not only prolongs the time-to-market, but also leads designers to make simplifying assumptions about other disciplines until the implementation of all disciplines is integrated.

1.1.4 Availability of Affordable Hardware

Assistive BBCI applications require many devices of different types to work together. However, challenges stem from the availability of affordable, accurate BBCI Hardware. We identify three class of hardware required to interface the human, including a DAQ device for bio-signal acquisition, a Stimulus device generating stimuli to invoke physiological responses, and a Presentation device to provide feedback and instructions.

Developing BBCI applications using existing interface devices sets high barriers of entry, including requiring embedded system knowledge, high integration effort, and device limitations and
CHAPTER 1. INTRODUCTION

cost. Integrating individual stand-alone devices into assistive applications is tedious and error prone, unnecessarily burdening BBCI developers. Off-the-shelf BBCI devices are often too expensive and incomplete due to the lack of embedded processing capability. While the accuracy of intent inference from the bio-signal is essentially dependent on the quality of the acquisition, a high-resolution commercial DAQ such as like a $15,000 g.USBamp (g.tec Medical Engineering GmbH) is cost prohibitive and hinders many researchers from contributing to this interesting field. To allow rapid evolution of the BBCI research, as well as the proliferation of societal impacting assistive applications, a coordinated device suite tailored for BBCI and supported by manufacturer-agnostic API to simplify application prototyping and development is needed.

Overall, assistive BBCI applications improve the quality of life, but it does not come without a cost.

1.2 Problem Definition

Designing, developing and deploying BBCI-based assistive applications is extremely challenging and time-consuming. This dissertation addresses the following problems:

1. **Concerted multi-disciplinary efforts required for application integration.**
   A holistic approach is needed that provides a common view and bridge disciplines to integrate efforts across multiple disciplines

2. **Vertical integration from high-level algorithm prototype to low-level embedded deployment.**
   While algorithm designers prototype assistive applications in high-level development environment, assistive products requires portable deployment beyond a laboratory setting. A holistic design flow is needed that bridges the gap from high-level algorithm development to low-level embedded realization.

3. **Interfacing with various human interface devices.**
   Dealing with manufacturer-specific APIs of various human interface devices is tedious and unnecessarily burdening developers. Hardware abstraction is required to simplify interfacing with various human interface devices and integrating new devices.

4. **Communication in distributed system.**
To allow for distributed composition in assistive BBCI applications, communication design support for flexible connectivity among distributed systems is needed.

5. **Availability of affordable human interface devices that are easily integrable.**

Developing applications using existing stand-alone human interface devices is expensive and requires huge integration effort. To reduce the hardware barrier of entry, affordable, coordinated device suite tailored for BBCI with processing integrated is needed.

A holistic approach is needed to address these challenges in the design, development, and deployment of assistive BBCI applications.

### 1.3 Dissertation Overview

This dissertation addresses the challenges in developing assistive applications from multiple perspectives. We propose a BBCI-based Assistive Technology (BAT) framework providing a holistic approach to design, develop and deploy complex assistive BBCI applications. Figure 1.2 overviews our framework bridging multiple disciplines and a holistic design flow for vertical integration using the model-based design (MBD) concept. To facilitate the BBCI realization, we provide an open embedded device suite for BBCI, called **BODE Suite**.
CHAPTER 1. INTRODUCTION

To capture various assistive BBCI applications, Chapter 3 formalizes these applications across human, cyber and physical domains, each with a distinct responsibility. Chapter 3 then introduces the holistic BAT framework to capture a variety of applications. Our BAT framework abstracts these three domains into three essential systems: a) human interface through physiological signal extraction, stimulation, and feedback to the user, b) computer interface fusing physiological and non-physiological evidence for intent inference, and c) action system interacting with the physical environment. These systems communicate through well-defined interfaces. Our framework is configurable in what is presented to the user (icons) and the supported actions. The modular design, the well-defined interfaces and the application-specific configuration allow constructing a wide range of assistive applications. To accelerate the development process of complex assistive applications, our framework integrates a design flow using a MBD approach. Chapter 4 discusses the design flow, from system modeling and algorithm design in high-level development environment, to rapid application prototyping testing on human subjects, to automated embedded deployment. Our domain-specific synthesis toolchain, HSyn, automatically generates the embedded implementation of applications for the portable deployment.

To reduce the hardware barriers of entry, this article introduces an open, affordable embedded BBCI device suite, called BODE Suite. Chapter 5 analyzes the design and the quality of the suite, including a bio-signal DAQ device (EEGu2), a multisensory Stimulus device (StimTron) and a visual Presentation device (PstVis). EEGu2 (in Section 5.3.1) provides 16-channel 24-bit acquisition at 25dB signal-to-noise ratio (SNR), powerful processing capability (Cortex-A8), and intelligent power management for up to 12 hour mobile operation. The StimTron (in Section 5.4) generates high-precision visual and tactile stimulation with run-time configuration of bit pattern and frequency from 0.1Hz to 409.6Hz at 0.1Hz resolution. The PstVis (in Section 5.5) offers portable display and integrates a rail car system for flexible stimuli placement. The suite is low-power/battery-powered, capable of embedded processing and fully integrated into our framework with hardware and location transparent interfaces in MATLAB. The affordable, plug-and-play suite devices enable researchers to easily develop a wide range of BBCI applications.

To show the flexibility and expandability of the framework, we demonstrate three assistive application examples in Section 6 using modular systems with convenient application-specific configuration. The applications are a brain-controlled robotic wheelchair (Section 6.1), a brain-controlled robotic arm (Section 6.2), and a BCI Speller (Section 6.3). In particular, the wheelchair application is discussed in detail to show the applicability and usability of our framework and the suite. This application augments the user’s mobility by driving a semi-autonomous wheelchair.
CHAPTER 1. INTRODUCTION

via BBCI. We demonstrate the hardware abstraction with the transparent access to various BBCI hardware devices with different connectivity. Following the MBD approach, we rapidly prototyped the application in MATLAB, tested it on human subjects, and deployed it on our EEGu2. HSyn enabled six orders of magnitude of productivity gain compared to manual embedded deployment.

In short, our flexible framework (software), affordable device suite (hardware), and the design flow compose a comprehensive starter kit for rapid design, development, and deployment of assistive BBCI applications. Section 2 describes the background of CPS and classification of HiLCPS. Section 2.3 discusses the related work. Section 8 concludes the dissertation.

1.4 Contributions

This dissertation addresses challenges in designing, developing, and deploying assistive BBCI applications in three aspects: our BAT framework for horizontal integration across multiple disciplines, a rapid design workflow for vertical integration from algorithm prototyping to embedded deployment, and our BODE Suite providing affordable, ready-made, coordinated human interface devices. The contributions of this study are the following:

1. **Holistic BAT Framework for Horizontal Integration**

   The framework provides a holistic overview of assistive BBCI development and integrate efforts from multiple disciplines.

   - **Application Formalization**: We formalize assistive BBCI applications and identify the partition of common functionality.

   - **Generic Modules with Flexible Configuration**: Utilizing the application formalization, our framework provides a set of generic modules with well-defined interfaces, each defining the responsibility of one discipline and its interface across discipline boundaries. By externalizing configurations, the modules are application-agnostic and can be configured to support a wider range of assistive applications.

   - **Modular, Distributed Composition**: With standard module interfaces, our framework adopts a publish-subscribe communication paradigm, using Data Distribution Service (DDS), which enables a modular, distribution composition in the communication application development.
CHAPTER 1. INTRODUCTION

2. Rapid Design Workflow for Vertical Integration
We provide a rapid design flow using a model-based design (MBD) approach to accelerate the application design in three steps, from the system modeling in MATLAB (model-in-the-loop), to rapid application prototyping (tested on human subjects), to automated embedded deployment via HSyn’s domain-specific synthesis.

- **BBCI Hardware Abstraction:** To simplify interfacing with various BBCI devices during rapid prototyping, we group similar device types to classes, called DevClass, with abstract interfaces offering hardware and location transparent access from MATLAB. Using DevClass simplifies the application prototyping and saves designers from dealing with hardware-specific details and customized communication design.

- **HSyn’s Domain-specific Synthesis:** HSyn automatically synthesizes the MATLAB application prototype to its embedded implementation, which empowers algorithm designers to focus on assistive algorithm development without worrying its embedded deployment.

3. Open, Affordable BBCI Device Suite
To interface with the human, our BODE Suite provides a package of affordable embedded BBCI devices with processing integrated.

- **EEGu2:** It provides 16-channel, 24-bit resolution acquisition with 25dB signal-to-noise ratio and battery-powered embedded processing for mobile operation.

- **BODE Suite:** It generates 12-channel visual and tactile stimulation with configurable bit pattern and frequency from 0.1Hz to 409.6Hz at 0.1Hz resolution.

- **PstVis:** It presents visual feedback and additionally incorporates a rail car system for flexible stimulus placement.

The suite enables a rapid, cost-effective BBCI application prototyping using plug-in-play, ready-made human interface devices supported by BBCI hardware abstraction.

4. HW/SW Co-design from Simulink
To utilizing video feed in assistive applications, we target heterogeneous platform for specialized computation for feature extraction. Instead of developing customized hardware

---

1In this framework, we use MATLAB as the algorithm development environment due to its popularity. The concept and principle of hardware transparency, however, are transferable to other algorithm development environments, such as Octave and Scilab.
CHAPTER 1. INTRODUCTION

accelerator for video processing, we propose a HW/SW Co-design toolchain, SimSH, exploring the domain-specific synthesis to target heterogeneous platforms and leveraging their low power and high performance benefits. SimSH provides an automatic path from an algorithm in Simulink to a heterogeneous implementation. Given an allocation and a mapping decision, the SimSH automatically synthesizes the Simulink model onto the heterogeneous target with optimizing the synchronization and communication between processing elements.
Chapter 2

Background

This dissertation addresses the challenges in designing, developing and deploying assistive BBCI applications. This type of applications is a subclass of Human-in-the-Loop Cyber-Physical Systems (HiLCPS). This chapter first overviews the background of Cyber-Physical Systems (CPS) and then analyzes the classification of HiLCPS. Section 2.2 introduces the fundamentals of model-based design (MBD), a powerful design technique for CPS. Finally, Section 2.3 discusses the relevant research in the area of designing complex assistive BBCI applications.

2.1 Cyber-Physical Systems

Assistive applications augment users’ interaction with the physical world, which are naturally Cyber-Physical Systems (CPS). CPS are networked systems with computation and physical processes integrated. Figure 2.1 overviews CPS. The joint of “cyber” and “physical” requires sensing

Figure 2.1: Cyber-Physical System (CPS)
CHAPTER 2. BACKGROUND

Figure 2.2: Human-in-the-Loop Cyber-Physical System (HiLCPS)

the environment, control processing and actuators acting onto the physical world [28]. The computing platforms communicate through network for coordination or to achieve an overall goal.

The term “Cyber-Physical System” emerged around 2006, when it was coined in a workshop held by National Science Foundation (NSF), a U.S. government research institute [29]. The CPS technology involves multi-disciplines including embedded systems, communication, control theory, physical modeling, etc. Examples are advanced automotive systems, medical devices, smart buildings, transportation systems and so on [30].

One key property of CPS is predictable timing [29]. Physical systems operate in real-time, which can be mathematically modeled using ordinary differential equations. To interact with physical systems, real-time computation for predictable timing is needed. While the low-level processing hardware has deterministic timing of execution, the high-level software program does not capture computation timing, which is intrinsically hidden by the abstraction of instruction set architecture and programming language. Research in [31, 32, 33] investigates the timing precision at the software level of abstraction.

2.1.1 Human-in-the-Loop Cyber-Physical Systems

Assistive applications are in a subclass of Human-in-the-Loop Cyber-Physical-Systems (HiLCPS) that interact with both human and physical world [10]. Figure 2.2 depicts the human involvement in CPS for control as well as collaboration in the physical environment. Examples are a user driving a robotic wheelchair which restores user’s mobility, a self-driving car.
sharing the road with a human driver, and an augmented hand holding a cup of coffee with the help of the other normal hand. Few researchers [34] consider the human performing tasks as an extension of actuators; they propose a human service capability description model which describes human capability to perform tasks. However, in the scope of assistive applications, we focus on the HiLCPS interacting with the physical environment on behalf of the human.

To identify the characteristics of our target applications, we classify HiLCPS in three axes: level of autonomy, tightness of collaboration, and the extraction of human intent. The level of autonomy centers on who makes decision, human or computer. The human can participate in the control process at various levels, which results in different levels of autonomy. As illustrated in Figure 2.3, we classify HiLCPS in three degrees of automation: manual control, semi-autonomous and autonomous systems. A fully autonomous HiLCPS operates autonomously and accepts user configuration if needed. An example is a smart thermostat that monitors the occupancy of room and turns off HVAC to save power. The user can change the thermostat setting, but does not directly control system actions (turning off HVAC). While a fully autonomous system does not require human intervention, a manual system asks the user to tele-operate all actions. In the middle of autonomous and manual systems, a semi-autonomous HiLCPS takes top-level decision from the user and executes it on the user’s behalf. An example of semi-autonomous HiLCPS is a robotic wheelchair. While the user navigates to move toward a direction or turn around, the robotic wheelchair drives low-level motors to follow a trajectory avoid obstacles and danger. For illustration purposes, we only use a coarse grain subdivision in the levels of autonomy. More accurate, finer grained distinctions are possible, as for example proposed by Sheridan [35].

As an orthogonal concept to the autonomy level, the tightness of collaboration evaluates the temporal and spacial correlation between system actions and human actions. In HiLCPS with
CHAPTER 2. BACKGROUND

no direct collaboration, the user only controls CPS to perform tasks; however, the user does not directly perform the action in the physical space. For example, a health wristband application tracks users’ sleep, steps and heart rate, and in turn provides health recommendations based on these information. The application only monitors users and does not require human-machine collaboration. A coarse-grain collaboration is required when HiLCPS interact with each other in the same physical space and time. For example, a self-driving car must adapt to the surrounding traffic and collaborate with other drivers to share the road. Furthermore, the HiLCPS that manipulates the object together with the user demands a tighter collaboration between the human and the system. For instance, holding a cup of coffee using both an robotic hand and a normal hand requires the coordinated efforts of both hands. To achieve a concerted manipulation, the augmented hand system must seamlessly adapts to human gesture and collaborate with the other normal hand. When HiLCPS collaborates more tightly with the human, the system takes more human factors into consideration and therefore behaves more human-alike. Increasing tightness of collaboration between HiLCPS and the human makes HiLCPS design increasing complex and challenging.

Another axis to categorize HiLCPS is how the system extracts human intent. Users can provide direct input to the system through traditional I/O devices such as keyboard, touchscreen, joystick, etc. An additional communication channel to obtain human intent is available through BBCI. BBCI infers user’s intent from physiological evidence and interprets it as the control over the system. Some other HiLCPS can also “guess” human intent purely based on statistic history of human behavior. For example, the smart thermostat can learn users’ preference of the temperature during different time of a day from the configuration history and intelligently controls HVAC.

This dissertation focuses on assistive BBCI applications in the context of HiLCPS which extracts user’s intent via BBCI and augment user’s interaction with physical environment. These assistive BBCI applications cover all levels of autonomy and tightness of collaboration.

2.2 Model Based Design

Model-based design (MBD) has evolved to be a powerful design technique for cyber-physical systems by emphasizing mathematical modeling to design, analyze, verify and validate complex dynamic systems in which physical processes and computation are tightly integrated [27]. Figure 2.4 depicts the V-diagram of industrial design process using MBD [1]. The left stroke of “V” is top-down design process and the right stroke is bottom-up test and verification process. The design requirement is transformed into system specification which is further decomposed into subsystems,
CHAPTER 2. BACKGROUND

each modeled with well-defined interface specification and internal behaviors. A complete model of CPS encapsulates the coupling of physical processes (plant) and embedded computation (control software). Physical processes describes the dynamic behavior versus time and are usually modeled in the form of differential equations and Laplace transfer functions. Models of computation describes system behavior in a abstract, conceptual form, which results in a greater analyzability of features and complexity. The models can be simulated and tested, which enables designers to verify the algorithm functionality, performance, and assumptions before the actual implementation. The control model can be fine tuned via rapid prototyping. In rapid prototyping, the control model is simulated on a PC taking real-time input and output from the physical plant (sensors and actuators). In on-target rapid prototyping, the control software is executed on the target processor (the same as controller hardware) to validate the resource constraint (e.g. ROM/RAM). Once the control model has been designed and verified, a code generator synthesizes code that faithfully executes the semantics of the model of computation.

From the bottom to up right in V-diagram, the implemented components are put together to realize the subsystems which are tested at various composition level. Software-in-the-Loop (SiL) incorporates the generated software code back to the control model which is simulated with the plant model on a PC. SiL validates the generated code of control model has the same functional behavior as the model itself. Processor-in-the-loop (PiL) tests the control software on target target processor

Figure 2.4: Industrial Design Process using Model-Based Design [1]
and validates the cross-compiler, linker and loader. In PiL, the target processor communicates with the plant model simulated on a PC through standard communication such as Ethernet. As one step further from PiL, hardware-in-the-loop (HiL) simulates the plant model on a PC in real-time that communicates with the controller through actual input and output device. Controller outputs physical electrical signals to the PC (as if the input to actuator). The PC converts the signals into physical variables which are fed into plant model. The model calculates output variables and PC converts them to electrical signal (as if the output of sensor) sent to controller.

Overall, MBD is a powerful design technique for cyber-physical systems. MBD empowers developers to verify and correct the design with little efforts at an early design phase. Furthermore, MDB also accelerates development process and reduce time-to-market by generating artifacts such as code and test. In the scope of this dissertation, we focus on the computational modeling (BBCI algorithms) in state machines that may also include modeling physical dynamics. In our MBD flow for HiLCPS, designers first model algorithms and then rapidly prototype the application tested on human subjects (Section 4.1). Finally our domain-specific synthesis tool, HSyn, automatically generates the embedded implementation of the application for portable deployment.

2.3 Related Work

This section briefly discusses relevant research. In general, we categorize the related work into two major categories: software development of assistive BBCI applications and specialized devices to interface with the human.

2.3.1 Software Development of Assistive BBCI

A variety of research efforts have been made to ease the development of applications interacting with humans through a cyber-physical system. Researchers in [36, 37] tackle the challenge of interfacing hardware. A middleware, called ANGELAH [36], is developed for monitoring elders and emergency detection at home. ANGELAH integrates sensors and actuators using Open Service Gateway initiative [38] for component abstraction. Mueller-Putz et al. [37] propose a hybrid Brain-Computer Interface (hBCI) framework with data fusion and classification for multiple data sources. However, hBCI primarily focuses on the hardware transparent access to data acquisition device. Both ANGELAH and hBCI offer hardware abstraction, but neither address the challenge of embedded deployment in the application development. Unlike ANGELAH and hBCI, BAT frame-
work incorporates HSyn and offers domain-specific synthesis for rapid development of embedded assistive applications.

Researchers have also investigated the communication and processing in distributed systems. Plourde et al. [39] develop an open Integrated Clinical Environment (OpenICE) facilitating the interoperability of distributed medical devices. ICE provides a set of device adapters that represent hardware and communicate through Data Distribution Service (DDS) [40] in a location-anonymous publish-subscribe pattern. A healthcare application architecture, CPeSC3 [41], provides ubiquitous sensing, communication and cloud computing. CPeSC3 builds a local secured wireless sensor network which communicates with the cloud with real-time scheduling and resource management. BCI2000 [42] is a software framework for prototyping Brain-Computer Interface (BCI) applications. BCI2000 provides a user-friendly graphical user interface to configure the data acquisition using g.Tec USBamp and data analysis. Unlike [39, 41, 42], BAT framework explicitly introduces an additional layer of hardware abstraction, DevClass that represents the taxonomy of hardware devices with similar specification.

While the above approaches make advances in individual areas, they do not focus holistically on developing embedded assistive applications. In contrast, BAT framework offers a complete design flow: 1) algorithm model design, 2) rapid assistive application prototypes in MATLAB with hardware and location transparent access to hardware, 3) automatic embedded deployment.

2.3.2 Human Interface Devices

To interface with the human, BBCI requires three essential devices: DAQ, Stimulus and Presentation. BBCI requires a high-resolution, low-noise DAQ for extremely low-level bio-signals such as EEG from 1µV to 100µV. Moreover, DAQ needs sufficient dynamic range, e.g. to measure EEG and EMG (up to 10mV) at the same time. Commercial DAQ products like a $15,000 g.USBamp (g.tec Medical Engineering GmbH) provides high quality acquisition (24-bit), yet they can be cost prohibitive and hindering some researchers from contributing to BBCI research. While the g.USBamp must connect to a PC in a laboratory setting, a deployed assistive BBCI application requires embedded processing capability for mobile operations. Products like EMOTIV EPOC+ (16-bit) and Neurosky (12-bit) provide affordable bio-signal acquisition but at lower signal resolution mostly for gaming. Furthermore, these commercial DAQs don’t provide embedded processing for the application deployment. In contrast, our EEGtu2 provides 16-channel, 24-bit acquisition (equivalent to high-end g.USBamp) with processing integrated at only $600 for prototype and $400 for 1K unit

17
CHAPTER 2. BACKGROUND

production.

<table>
<thead>
<tr>
<th>Product</th>
<th>Resolution</th>
<th>Channel</th>
<th>Portable (battery)</th>
<th>Cost</th>
<th>Embedded Processing</th>
</tr>
</thead>
<tbody>
<tr>
<td>EEGu2</td>
<td>24 bits</td>
<td>16</td>
<td>Yes</td>
<td>$400</td>
<td>1GHz ARM + 2xPRU</td>
</tr>
<tr>
<td>g.USBamp</td>
<td>24 bits</td>
<td>16</td>
<td>No</td>
<td>$15,000</td>
<td>None</td>
</tr>
<tr>
<td>OpenBCI</td>
<td>24 bits</td>
<td>16</td>
<td>Yes</td>
<td>$900</td>
<td>None</td>
</tr>
<tr>
<td>EPOC+</td>
<td>16 bits</td>
<td>14</td>
<td>Yes</td>
<td>$3,000</td>
<td>None</td>
</tr>
<tr>
<td>NeuroSky</td>
<td>12 bits</td>
<td>1</td>
<td>No</td>
<td>$99</td>
<td>None</td>
</tr>
</tbody>
</table>

Table 2.1: Other BCI DAQ Systems

OpenBCI \cite{43} has introduced open-source DAQ. OpenBCI supports 16-channel 24-bit acquisition, however, due to hardware limitations, is limited to an effective sample rate of 125 Hz when all 16 channels are used. The system samples at 250 Hz and averages data with the previous time step (decimation by 2), resulting in limited usability for other types of physiological signals like EMG. Furthermore, the on-board microcontroller (50MHz PIC32MX250F128B) is used only for data transmission, not processing. In comparison, our EEGu2 supports up to 16KHz sampling rate and battery powered processing capability (BBB ARM Cortex-A8) for portable deployment of assistive BBCI applications.

Stimulus can be carried out in various means. For example, steady state visually evoked potential (SSVEP) based BCI requires visual stimulus \cite{19}. A tactile stimulus and visual stimulus can be used event-related potential (ERP) based BCI \cite{15, 16}. Designers often uses desktop monitors for visual stimulation due to the general availability of hardware and software library. But available stimulus frequencies are natively limited to the submultiples of the monitor refresh rate. The study in \cite{44, 45} proposes approximating frequency with a mixture of two or more frequencies. However, our experiment shows that frequency approximation on monitor produces inaccurate frequency with noisy spectrum (see Section 5.4.2). Unlike monitor stimulus, our StimTron generates clean accurate stimulus signal from 0.1Hz to 409.6Hz at 0.1Hz resolution with the signal quality of at minimum 20dB SNR.

Instead of using monitors, some research groups develop customized visual stimulus device. Mouli et al. \cite{46} uses LEDs controlled by a 32-bit Teensy microcontroller. To guarantee the stimulus timing, the microcontroller exclusively runs a bare-metal program bit-banging digital I/Os to control LEDs. In result, this stimulus device as an island system does not support communication for runtime
configuration of stimulus frequency, bit pattern. In comparison, our StimTron decouples the real-time stimulus generation using field-programmable gate array (FPGA) from the loosely-timed software running on BBB.

Integrating existing island solutions and developing custom-made hardware is time-consuming and error-prone. To facilitate the assistive BBCI development, our BODE Suite provides a package of synergetic BBCI devices. The suite is further integrated into the framework, which enables designers to easily use devices from unified MATLAB interfaces.
Chapter 3

Holistic BAT Framework

We provide a holistic, flexible framework that facilitates development of assistive BBCI applications. We first formalize the class of assistive BBCI applications that span human, cyber and physical domains, each with a distinct responsibility. Our BAT framework abstracts these three domains into three essential systems without losing generality and flexibility. The systems communicate through well-defined interfaces. Our framework is configurable in what is presented to the user (icons) and the supported actions. The modular design, the well-defined interfaces and the application-specific configuration allow constructing a wide range of assistive applications. This chapter describe the target application abstraction, the framework subsystems, the application configuration and the communication scheme.

3.1 Assistive BBCI Application Abstraction

To provide a holistic framework, it is critical to identify common functionality among various applications. In Figure 3.1 we formalize the class of assistive BBCI applications. Each application defines a set of possible actions that it provides to assist the user. For example, an action for a speller application is to spell a specific character. An application informs users of available actions by presenting icons, each associated with one or a few actions. An icon can be presented as a visual display or an audio cue. Each presented icon has a 1:1 correlation with a stimulus presented to users in various forms, including visual [19], tactile [15,16] and auditory [47] forms. Users select an icon (and corresponding action) by focusing on the associated stimulus. Focusing on the stimulus invokes a physiological response. Responses can be measured electrically, such as through electroencephalography (EEG), electromyography (EMG), electrocardiography (ECG), and
electrooculography (EOG), which measure electrical signals emitted by the brain, skeleton muscles, the heart, and eyes respectively. The application then infers human intent from the physiological evidence collected by the DAQ and enacts inferred actions through low-level actuators and sensors. Then, the cycle repeats; the application continues and updates icons for users to make a new selection.

If more actions are available than simultaneously displayable icons, presenting all available actions requires their partition into subsets, each mapped to an icon and the corresponding stimulus. To select the desired action, users may need to perform several selections to navigate from an icon that represents a group of actions, including the intended action, eventually to the icon specific to that action. For example, the BCI Speller in [48] allows users to type 26 characters and a space (actions) by presenting only six icons with grouped characters at a time. Mapping an icon to a set of actions enable applications to support a large number of actions.

In this class of assistive BBCI applications, users make discrete selections at a low frequency, due to the limited bandwidth of BBCI. While the inferred action is loosely-timed (human-cyber), the low-level realization of an action is in real-time (cyber-physical). Integrating components with diverse sense of time poses a challenge in developing such applications.

To capture this class of applications, our framework provides modular, generic systems that abstract common functionality with configurable actions and icons.
CHAPTER 3. HOLISTIC BAT FRAMEWORK

3.2 Framework Systems

Utilizing the assistive application formalization, our BAT framework abstracts applications into three systems depicted in Figure 3.2: a) human interface to extract physiological evidence, b) computer interface for intent inference from physiological and non-physiological evidence, and c) action system interacting with the physical environment.

3.2.1 Human Interface

The human interface collects all physiological data (input) and presents stimulus and feedback (output) to the user. The data acquisition subsystem collects raw physiological signals. The stimulus subsystem generates physiological stimuli to evoke the necessary response for the user intent inference. The presentation subsystem presents available system actions in the form of icons as well as feedback information such as the system status.

3.2.2 Computer Interface

Computer interface estimates the desired action (intent) using physiological (input from DAQ) and non-physiological (from context) evidence. The inference engine develops an action probability distribution given all evidence (action posterior). Due to the noisy nature of brain and body signals, context-aware inference is necessary to make reliable decisions. The context engine complements the inference process by adding non-physiological information relevant to the task at
CHAPTER 3. HOLISTIC BAT FRAMEWORK

hand. For example, in a speller application, a language model with character and word probabilities given previously typed characters can significantly improve communication rates [14]. The context engine also stores action history and generating action probabilities given past behavior (action prior).

3.2.3 Action System

The action system is the physical interface of our framework that enacts the user desired action received from the inference engine. While action engine enacts the action through actuators, perception engine detects the environment via sensors to facilitate the actuator control. The action system translates high-level system action into a series of lower-level tasks executed in fast local loops. However, challenges stem from the shared control between BBCI (on behalf of the human) and action system. While the human makes top-level decisions through a slower BBCI loop (nonrealtime), the local realization is left to the discretion of the semi-autonomous action systems that runs a faster control loop (realtime) to interact with the changing environment. Although the human owns the system control, safety overrides might be required to avoid implausible actions, depending on the overall physical state, in that the action system may detect the danger of environment that the human is unaware of.

3.3 Framework Configurability

Various applications can have a wide range of possible semantics of actions. Instead of limiting to a predefined set of actions, our framework provides generic human interface and computer interface with configurable action vocabulary and icon set. The action vocabulary is the collection of all action identifiers. The icon set defines all icons and their mapping to actions. Applications present a fixed number of stimuli, each paired with an icon. The human interface and computer interface work abstractly on the action and icon identifiers. It is agnostic of the actual semantic meaning of an action. Conversely, the action system cannot be generalized due to the distinct functionality and structure of action realization. Instead, our framework provides a standard interface to action system and allows a parallel development of action system and BBCI. The abstraction of actions and icons empowers designers to develop application-agnostic BBCI which can be reused and configured for a wide range of applications. Ideally, to prototype a new assistive application, designers simply need to instantiate the necessary systems (and subsystems) with proper icon and action configurations. We demonstrate the flexibility of our framework using four applications: BCI Speller, Brain-controlled IoT, Prosthetic
### Table 3.1: Application Examples using BAT Framework

<table>
<thead>
<tr>
<th>Application</th>
<th>BCI Speller (Voice Synthesizer)</th>
<th>Brain-controlled IoT (Smart Dimmer)</th>
<th>Brain-controlled Wheelchair</th>
<th>Prosthetic Robot Hand</th>
</tr>
</thead>
<tbody>
<tr>
<td>Action Vocabulary</td>
<td>Character A</td>
<td>Character B</td>
<td>Character 100%</td>
<td>Character Pinch</td>
</tr>
<tr>
<td>Icon Set</td>
<td><img src="image" alt="A" /></td>
<td><img src="image" alt="B" /></td>
<td><img src="image" alt="↑" /></td>
<td><img src="image" alt="ok" /></td>
</tr>
<tr>
<td>Action Vocabulary</td>
<td>Character A</td>
<td>Character B</td>
<td>Character 500%</td>
<td>Character Sphere</td>
</tr>
<tr>
<td>Icon Set</td>
<td><img src="image" alt="A" /></td>
<td><img src="image" alt="B" /></td>
<td><img src="image" alt="↑" /></td>
<td><img src="image" alt=" Palm" /></td>
</tr>
<tr>
<td>Action Vocabulary</td>
<td>Character A</td>
<td>Character B</td>
<td>Character 250%</td>
<td>Character Finger</td>
</tr>
<tr>
<td>Icon Set</td>
<td><img src="image" alt="A" /></td>
<td><img src="image" alt="B" /></td>
<td><img src="image" alt="↑" /></td>
<td><img src="image" alt="Grip" /></td>
</tr>
<tr>
<td>Action Vocabulary</td>
<td>Character A</td>
<td>Character B</td>
<td>Character OFF</td>
<td>Character Grip</td>
</tr>
<tr>
<td>Icon Set</td>
<td><img src="image" alt="A" /></td>
<td><img src="image" alt="B" /></td>
<td><img src="image" alt="↑" /></td>
<td><img src="image" alt="Grip" /></td>
</tr>
</tbody>
</table>

- **BCI Speller (Voice Synthesizer)**
  - **Character A**: A
  - **Character B**: B
  - **Character Z**: Z

- **Brain-controlled IoT (Smart Dimmer)**
  - **Brightness 100%**: 100%
  - **Brightness 500%**: 50%
  - **Brightness 250%**: 25%
  - **Brightness OFF**: OFF

- **Brain-controlled Wheelchair**
  - **Move Forward**: ![Up](image)
  - **Turn Right**: ![Right](image)
  - **Turn Left**: ![Left](image)
  - **Move backward**: ![Down](image)

- **Prosthetic Robot Hand**
  - **Pinch**: ![Pinch](image)
  - **Sphere**: ![Sphere](image)
  - **Finger**: ![Finger](image)
  - **Grip**: ![Grip](image)
CHAPTER 3. HOLISTIC BAT FRAMEWORK

Robot Hand, and Brain-controlled Wheelchair. Table 3.1 illustrates the action vocabulary and icon set for each application.

The BCI Speller application, listed in the first row of Table 3.1 augments LIS users’ communication and enables them to type using the steady state visually evoked potential (SSVEP) paradigm [49, 19]. SSVEP is the brain response to a flickering pattern measurable via EEG. Flickering stimulus patterns are paired with system actions (characters) presented as visual icons. The user types one character by visually focusing on the stimulus associated with the character. The BCI infers user desired character from the user’s EEG response. Note that typing one character may need a few selections to navigate from a character grouping (e.g. “H-L”) to the final character (e.g. “K”). Once a sentence is composed, the voice synthesizer can speak it out on behalf of the user.

By simply changing the action and icon configurations, the same SSVEP paradigm and algorithm (inference and context engines) are employed to create a Brain-controlled Internet-of-Things (IoT) application controlling a smart dimmer illustrated in the second row of Table 3.1. Instead of character icons, the application presents the various brightness options (icons) to set the light to 100%, 50%, and 25% brightness or OFF (actions). The action system translates the action into low-level commands controlling the physical dimmer. Other devices, such as curtain opener and smart outlet, can be incorporated by expanding the action vocabulary and icon set.

The Brain-controlled Wheelchair application helps LIS users to regain their independence by controlling a robotic wheelchair. Using the same SSVEP paradigm and algorithm to extract user intent, designers simply reconfigures the action vocabulary and icon set (Table 3.1, the third row). The application presents four icons associated with four actions to move the wheelchair (forward, backward, left turn, and right turn). While the robotic wheelchair requires additional development effort, the abstraction of action system allows independent wheelchair development and verification. Section 6.1 discusses this application in more detail.

The Prosthetic Robot Hand application restores the hand operation for amputees. The application converts EMG measured from the upper limb to desired robot poses [50]. In this application, the Stimulus and Presentation subsystems are not needed. A trained user can control the robot hand to perform a posture from the action vocabulary by attempting to do the same (or different mapping) with the amputated upper limb. The last row of Table 3.1 list four common postures - palmar pinch, precision sphere, prismatic finger and large diameter grip from the action vocabulary of the robot hand.

The modularity and flexibility of our BAT framework simplifies the development of assistive applications. Designers can rapidly prototype new applications by adding and configuring...
CHAPTER 3. HOLISTIC BAT FRAMEWORK

Figure 3.3: Framework Publisher-Subscriber Communication Paradigm

plug-and-play systems of the framework based on the application requirement.

3.4 System Interface and Communication

To enable a flexible modular composition, each subsystem in the BAT framework has predefined interfaces. The interface is captured in Interface Definition Language (IDL) provided by Object Management Group (OMG) [40]. IDL defines interfaces that both client and server objects understand regardless of programming language. From the subsystem IDL interfaces, the BAT framework can generate the programming language implementation that communicates the topic structure defined in IDL. This allows implementing systems in different programming languages (as supported by IDL). Our framework uses a publish-subscribe communication paradigm shown in Figure 3.3. Each distributed subsystem can publish or subscribe one or multiple topics. A subsystem can hear from another only if they have the same topic (same interface definition). The subsystem interface definition and publish-subscribe paradigm allow a modular design and simplifies the communication development.

Our framework uses a communication middleware - Data Distribution Service (DDS) [40] that integrates IDL and provides publish-subscribe paradigm. DDS is an open middleware for real-time publish-subscribe communication and is already used in medical applications [39]. DDS

\[1\] Our BAT framework uses OpenDDS [51] which is an open source implementation of DDS. However, each distributed system has a free choice of DDS implementation (such as RTI Connext [52]) due to the interoperability of DDS.

26
provides a rich set of Quality of Service (QoS) to tailor the communication behavior, such as reliability, durability, the persistence of data, and the amount of system resource used.

With application-defined topics, DDS allows application developers to implement the communication at application level without handling chores such as data marshalling and demarshalling, message addressing, etc. Figure 3.4 illustrates an interoperable layered communication design using DDS. Given a predefined topic (e.g., InferredAction), DDS generates application APIs. On the left side of Figure 3.4, a C++ application writes an instance (action) of a topic (InferredAction), whereas a MATLAB application reads it on the other side. Since OpenDDS does not generate MATLAB DDS interface, we develop a MATLAB DDS driver that wraps the C++ DDS API in MATLAB Executable (MEX) and thus expose it to MATLAB. DDS library transparently handles the message delivery across different operating systems and network protocols such as TCP and UDP. DDS uses a standard wire and wireless protocol that enables interoperability on the wire or wireless among different implementation of DDS such as OpenSplice, RTI DDS, etc.

We use the *Stimulus* as an example to demonstrate system interfaces. The *Stimulus* generates a pattern perceived by the human whose physiological response can be measured and analyzed to correlate with that stimulus pattern. In Listing 3.1 the *Stimulus* interface includes properties: a) *bitPatt* - an array of 0s and 1s, b) *bitPattLen* - effective number of bits in the *bitPatt*, c) *bitRate* - rate of output bits. The *Stimulus* interface is agnostic to the physical realization of stimulation and hides the hardware-specific details from other systems. The public *Stimulus* control interface is specified by *StimulusCmd* and *ChNum* to start, stop and configure the given channel stimulus. The *Stimulus* interface also includes optional configurations specific to certain stimulus media, such as visual pattern format (checkerboard or one group in checkerboard) and brightness for
CHAPTER 3. HOLISTIC BAT FRAMEWORK

Listing 3.1: Stimulus Subsystem Interface

```c
#define MAX_BITPATT_LEN 32
typedef sequence<char, MAX_BITPATT_LEN> BitPatt_t;

module StimulusSystem {
  enum StimulusCmd {Start, Stop, Config}; // stimulus function
  enum LedArrayMode {CheckerBoard, pChannel, nChannel, Solid};
  #pragma DCPS_DATA_TYPE "StimulusSystem::Stimulus"
  struct Stimulus { // Stimulus system interface
    StimulusCmd cmd;
    LedArrayMode ledMode;
    char ChNum;
    unsigned short brightness; // visual brightness
    float bitRate; // stimulus bit rate
    BitPatt_t bitPatt; // stimulus bit pattern
    unsigned short bitPattLen; // effective bits
  };
};
```

visual stimulus.

3.5 Summary

This chapter addresses the challenges in the horizontal integration of components from multiple disciplines in the development of assistive BBCI applications. To address these challenges, we formalize the applications and identify a common partition of functionality. From the application abstraction, we introduce a BAT framework to provide a holistic view in the application design and bridge disciplines. Our framework contains a set of abstract systems, each encapsulating one discipline. Systems communicate through well-defined interfaces using the publish-subscribe paradigm, which enables a modular, distribution composition. Our framework is application-agnostic and can be configured to support a wide range of applications. Overall, our framework tackles the multi-disciplinary challenge and facilitates the development of assistive BBCI applications.
Chapter 4

Assistive BBCI Design Flow

While the development of intensive signal processing algorithms can be dramatically simplified by using high-level algorithm development environment (e.g. MATLAB), the gap between high-level BBCI algorithm development and low level embedded implementation traditionally requires a significant amount of manpower for manual conversion.

To address the above challenge, we propose a design flow using a MBD approach that simplifies and expedites the development. MBD has evolved to be a powerful design technique for cyber-physical systems by emphasizing mathematical modeling to design, analyze, verify and validate complex dynamic systems in which physical processes and computation are tightly integrated [27]. Within our framework, we adopt an iterative MBD approach to model context-aware BBCI algorithms, rapidly prototype applications with the domain-specific synthesis. Our design flow in Figure 4.1 allows algorithm designers to rapidly develop embedded assistive applications in 3 phases as follows.

1. **BBCI Algorithm Design**: designers first model the algorithm in a high-level algorithm development environment (MATLAB). Then designers refines the algorithm through iterations of model-in-the-loop (MiL) testing using synthetic or pre-recorded physiological and non-physiological data.

2. **Rapid Application Prototyping**: in rapid prototyping, the application is validated with human subjects using actual assistive hardware. While the algorithm is executed in MATLAB, it communicates with physical sensors and actuators to interface with the human and the environment. The application instantiates DevClass modules from BAT framework to transparently read sensor data (algorithm model input) and control physical actuators (algorithm model output).
Instead of worrying about hardware details (i.e. manufacturer-specific driver), designers can focus directly on refining the algorithm.

3. **Embedded Deployment**: BAT framework enables an automatic path to embedded implementation using HSyn’s domain-specific synthesis. It automatically synthesizes the MATLAB application prototype down to a portable embedded implementation in C/C++ interfacing with hardware libraries.

   Our workflow enables rapid embedded assistive BBCI development, from algorithm design to rapid prototyping to embedded deployment. Next sections discuss hardware abstraction for rapid prototyping and domain-specific synthesis for automated embedded deployment.

### 4.1 DevClass Hardware Abstraction

To address the challenge of interfacing with various hardware and connectivity in the application design phase, our BAT framework groups hardware types into classes as shown in Figure 4.2 based on the following definitions.

*DevClass* is a class of all hardware types that implement the same semantics of input and output.

*DevType* is a target related realization of a device class.
Chapter 4. Assistive BBCI Design Flow

Figure 4.2: Hardware Device Categorization (Class, Type, Instance)

**DevInstance** is hardware instance of a **DevType**.

Matching the subsystems within the human interface, our framework realizes the **Dev-Classes**: DAQ, Stimulus, Presentation. The DAQ **DevClass** represents DAQ devices that acquire bio-signals. They share similar configuration parameters such as sampling rate and channel gain. The Stimulus **DevClass** generalizes stimulus devices with configurable frequency and bit pattern. The visual Presentation **DevClass** displays rendered figures and texts.

Figure 4.3 depicts the structural composition of a **DevClass**. The **DevClass** implements a unified API in MATLAB that hides the underlying **DevType**-specific (i.e. manufacturer-specific) driver API and its communication semantics from algorithm designers. The hardware and location transparency of **DevClasses** simplifies the rapid prototyping by hiding hardware-specific details from designers and avoiding customized communication design.

4.1.1 Hardware Transparent Access

To capture various **DevTypes**, an expandable design of **DevClasses** is needed. Our **DevClass** contains three levels of hardware abstraction: **DevClass** API, canonical HW API, and manufacturer driver API. A **DevClass** API is a unified user-level API in MATLAB. A canonical HW API unifies the low-level access to all **DevTypes**, each supported by a backend. A backend interfaces with manufacturer driver and translates between generic **DevClass** data and hardware-specific data. With hardware abstraction, a **DevClass** provides a unified access to the class of **DevTypes**, which enables a portable application design.

We use the DAQ **DevClass** API as an example to demonstrate the hardware transparent access from user level. In Listing 4.1, the API captures the fundamental functionality of DAQ, such as configuring channels, setting the sampling frequency, reading samples, etc. The `getSamples()`
method internally calls DevType-specific backend to read raw samples from DAQ hardware and returns samples in generic data type (double precision floating point) independent of the DAQ resolution. The `addChannel()` method enables a DAQ channel given an integer channel number, an channel name, and a filter. The `setSamplingFreq()` method sets the sampling frequency. `Start()` and `Stop()` turns on and off acquisition process. Our DevClass API generalizes the access to all DevTypes, which enables a portable, hardware-agnostic application design.

To show the benefit of convenient DevClass interfaces, Listing 4.2 demonstrates a usage example of DAQ DevClass API. To control a DAQ device, the application simply instantiates a DAQ DevClass object and configures the device connection (Ethernet in this example). After setting DAQ sampling frequency and desired channels, data acquisition starts. The DevClass object buffers samples and return them to the application upon the request via `getSamples()` call. The `getSamples()` returns immediately if the enough samples are available in the buffer, otherwise the call blocks until the requested amount of samples are acquired.

DevClass API abstracts the functionality of a class of hardware and generalizes a transparent access to all DAQ DevTypes in the class. Utilizing DevClass abstraction, designers can focus on the application algorithm without worrying dealing with various hardware drivers. The hardware abstraction also enables a portable, hardware-agnostic application design that simplifies switching between various types of devices.
CHAPTER 4. ASSISTIVE BBCI DESIGN FLOW

Listing 4.1: DAQ DevClass API Definition (MATLAB)

```matlab
1  % class definition
2  classdef DaqBase < handle
3    %#codegen
4  % class member elements
5  properties
6    ...
7  end
8  % class member functions
9  methods
10    % DAQ module initialization
11    Init(self)
12    % clean up the DAQ module
13    Close(self)
14    % enable channel number
15    addChannel(self,ChNum,ChName,ChFilter)
16    % return number of available samples
17    dataAvailable = SamplesAvailable(self)
18    % samplesPerCh*ChNum matrix of samples
19    data = getSamples(self,samplesPerCh)
20    % configure the sampling frequency
21    setSamplingFreq(self,SampleFreq)
22    % stop DAQ sampling
23    Stop(self)
24    % start DAQ sampling
25    Start(self)
26    % clear the current buffered samples
27    clearBuffer(self)
28    ...
29  end
```

Listing 4.2: DAQ DevClass API Example (MATLAB)

```matlab
1  % Create DAQ DevClass object
2  DaqTCPObj = DaqTCP;
3  % Connect DAQ device via TCP
4  DAQServerIp = ‘192.168.10.102’;
5  DAQPortNum = uint16(9999);
6  DaqTCPObj.init(DAQServerIp,DAQPortNum);
7  % set sampling freq
8  DaqTCPObj.setSamplingFreq(uint16(250));
9  % enable channel 1 and 2.
10  DaqTCPObj.addChannel(1);
11  DaqTCPObj.addChannel(2);
12  % clear internal buffer
13  DaqTCPObj.clearBuffer();
14  DaqTCPObj.Start(); % start acquisition
15  % main loop of acquistion and processing
16  for i = 1:10
17    % number of samples per channel requested
18    nSamplesToGetPerChannel = 250;
19    rawData = DaqTCPObj.getSamples(nSamplesToGetPerChannel);
20    for i = 1:10
21      % omit signal proc detail for simplicity
22      processing(rawData);
23    end
24  end
25  DaqTCPObj.Stop(); % stop acquisition
26  DaqTCPObj.Close(); % disconnect DAQ
```

4.1.2 Location Transparent Access

Assistive applications often control distributed hardware, e.g. remote sensors or actuators, to interface with the human and the physical world. Challenges stem from the complex communication design of distributed systems, including customizing payload format, data marshaling/demarshaling, and the transmission following a specific protocol. Ad-hoc implementation of communication schemes is prohibitively error-prone and dramatically decreasing development productivity. To address this challenge, our DevClass offers location transparent access to the remote hardware from the MATLAB interfaces.

To remotely control a hardware, the DevClass integrates a DevClass-Remote module which has direct access to the hardware. A DevClass communicates with the corresponding DevClass-Remote through a pair of proxy-master and proxy-slave. Figure 4.4 illustrates a DevClass accessing a remote HW1 as an example. Compared to the locally connected HW2, the control of HW1 is relayed via proxies to DevClass-Remote that calls the actual HW1. The proxy-slave runs dedicated thread that listens to proxy-master and notifies DevClass-Remote to execute the control. The DevClass-Remote reads HW1 data and asynchronously streams it back to DevClass through the proxies. The
Figure 4.4: Location Transparent Access to Distributed HW

proxy-master receives and buffers data for read calls from the application. The data streaming and buffering hide the communication latency by overlapping the communication with the processing. The application benefits from avoiding the idle time of waiting for sensory data from distributed systems. The streaming data packet size is configurable depending on the network quality. Without network congestion, the sensory data can be streamed at finest packet granularity (but with the highest packet overhead) in favor of the lowest response time.

The proxies are agnostic of HW types but specific to the communication types including network protocols and middlewares. The communication data type between proxies is tailored based on the generic data type of DevClass. One supported network type is DDS. DDS allows user-defined topics as the communication payload format that all participants agree upon. Applications can simply read and write data objects using the application-defined name (Topic) and a key. Each participant, namely proxy-master and proxy-slave (DDS), can publish and subscribe the topic without specific knowledge of the location or even existence of other participants. The proxies simply need to instantiate and send a message of a topic while the DDS library handles the data marshaling/demarshaling and reliable transmission over the underlying communication protocol. Listing 4.3 shows DAQ control topic as an example. A message of this topic relays the control (function call) from DevClass to DevClass-Remote. In DaqSystem, the topic DaqCtrl contains a cmd field as the function identifier and function arguments, such as the channel number (chNum) and the sampling frequency (freq). For example, setting sampling frequency 250Hz is interpreted as a control message of \{SetSampFreq,DoNotCare,250\}.
CHAPTER 4. ASSISTIVE BBCI DESIGN FLOW

Listing 4.3: Control Topic of DAQ Backend-Remote using DDS

```c
module DaqSystem {
  enum DaqCmd { // control command enum
    Start, // start DAQ
    Stop,
    AddChannel, // enable channel
    SetSampFreq // sampling frequency
  };
  # pragma DCPS_DATA_TYPE "DaqSystem::DaqCtrl"
  struct DaqCtrl { // control topic
    DaqCmd cmd; // func identifer
    unsigned short chNum; // AddChannel
    double freq; // SetSampFreq
  };
};
```

To have a small footprint and lightweight deployment, our DevClass also provides dedicated proxy realization for TCP. The proxies define a TCP frame shown in Figure 4.5 that integrates the DevClass interface in IDL. The proxy-master encodes the function call from canonical HW API into the control frame. The function is identified by a one-byte cmd field followed by the length of the entire frame. The function arguments are encoded in the data field (e.g. the value of sampling frequency to set). Since the largest length value is 255 (stored in 1 byte), the maximal data size is 252 bytes after excluding cmd, length and crc, which suffices the DAQ control. The last byte of the frame is for cyclic redundancy check (CRC). The control frame is forwarded to proxy-slave which notifies DevClass-Remote for execution.

<table>
<thead>
<tr>
<th>cmd</th>
<th>length</th>
<th>data</th>
<th>crc</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Byte</td>
<td>1 Byte</td>
<td>n Bytes</td>
<td>1 Byte</td>
</tr>
</tbody>
</table>

Figure 4.5: Packet Frame of DAQ TCP Proxy

In short, our DevClass provides hardware and location transparent access through backends and proxies. This abstracts enables portable application design and simplifies using hardware in rapid prototyping. Furthermore, a DevClass is scalable to support other hardware and communication types by adding the corresponding backends and proxies respectively. The DevClass flexibility dramatically increases with the combination of supported hardware types and network types.

4.2 HSyn: Domain-Specific Synthesis

Thanks to the hardware and location transparency of DevClass, designers can rapidly prototype and validate the MATLAB application using real sensors and actuators (phase 2 in
Figure 4.6: Software Stack of Prototyped Assistive APP

Figure 4.1. However, the application requires an embedded deployment to portable platforms to assist users in daily lives. Instead of traditional manual deployment, our BAT framework integrates a domain-specific synthesis tool, HSyn, to automatically generate the embedded implementation (phase 3 in Figure 4.1). HSyn utilizes Embedded Coder [23] as the core of code generation for the MATLAB application. Moreover, HSyn additionally generates C/C++ code to interface with external DevClass libraries, including backend (hardware library) and proxies (communication library). As HSyn uses the Embedded Coder for the underlying synthesis, it also inherits the restrictions of that product. Embedded Coder supports a subset of the MATLAB language. Features beyond MATLAB synthesizable subset include dynamic datatype, recursion, etc.

HSyn executes in two modes: execution within MATLAB accessing fixed C++ hardware/network driver libraries and domain-specific synthesis to generate C++ code seamlessly interfacing with those fixed libraries. HSyn identifies MATLAB runtime execution and synthesis process using coder.target() [23]. In MATLAB runtime, DevClass object interfaces with backends and proxies through MATLAB Executable Layer (MEX) [23]. MEX allows dynamic linking to external C/C++ libraries. Figure 4.6 depicts that the DevClass implementation (in green) spans MATLAB domain (DevClass API realization) and C/C++ domain (backend and proxy libraries). While DevClass is implemented as a MATLAB class using object-oriented design, the backends and proxies are natively implemented in C/C++ to interface with hardware and network drivers. In domain-specific synthesis process, the application and DevClass objects in MATLAB are automatically synthesized to C/C++ code, while backends and proxies initially developed in C/C++ are instantiated. HSyn specifies canonical HW API calls using coder.ceval() [23], which allows the synthesized DevClass object to directly interface with backends and proxies (skipping MEX).

Since the HSyn domain-specific synthesis does not change functionality and hardware connectivity, the application deployed on the embedded target collaborates with other distributed systems.
CHAPTER 4. ASSISTIVE BBCI DESIGN FLOW

just as if the application still runs in the host MATLAB. This fully automated embedded deployment bridges the gap from high-level BBCI algorithm design to low level embedded implementation and dramatically expedites the application development.

4.3 Summary

This chapter addresses the challenge of the vertical integration in the development and deployment of assistive BBCI applications. Our framework adopts a rapid design workflow using a MBD approach, from algorithm modeling, to rapid prototyping with simplified hardware access (via hardware abstraction), finally to automated embedded deployment (via HSyn’s domain-specific synthesis). To simply interfacing with the human in the rapid prototyping, our framework provides hardware and location transparent access to human interface devices. Furthermore, our framework offers an automatic path from application prototype to its embedded implementation using HSyn’s domain-specific synthesis. Overall, our design workflow enables a rapid development of embedded assistive BBCI applications. In result, designers can simply focus on the algorithm design without worrying about interfacing with hardware and embedded deployment.
Chapter 5

*BODE Suite: Open Hardware Suite for BBCI*

BBCI provides an additional communication channel in assistive technologies, such as operating a prosthetic device [3] and a brain-controlled robot [4], typing through virtual keyboards [5, 6, 7] and playing games [8]. Challenges stem from interfacing with the human in assistive applications. This chapter first surveys various BBCI applications and identifies commonly used interface devices and their challenges. To lower the barrier of entry in BBCI for the hardware equipment, we introduce an open device suite and discuss its design, quality, and usability with framework integration.

5.1 BBCI Device Categorization

To capture various BBCI applications, we formalize applications (in Section 3.1) as well as identify and categorize the required BBCI devices. While the application formalization provides a common view of applications, BBCI device categorization overviews the necessary devices to interface with the human. Some BBCIs utilize stimulus to elicit necessary physiological response for intent inference. For example, visually evoked potential (VEP) based BBCIs such as steady state VEP (SSVEP) [19] and code-modulated VEP (c-VEP) [20, 21] display flickering light (following a bit pattern for c-VEP) and measure VEP (EEG) from visual cortex. Steady-state somatosensory evoked potential (SSSEP) based BBCI applies vibratory stimulation to fingers and palms and measures steady-state EEG response from somatosensory cortex. Event-related potential (ERP) based BBCIs [15, 16] presents auditory, visual, or tactile stimulus to elicit P300 response [17], a
positive deflection in centro-parietal scalp voltage (measured via EEG) with a typical latency just over 300ms. We categorize stimulus-based BCIs utilizing three forms of stimulus: visual, tactile, and auditory.

Compared to some BCIs needing stimulus, all BCIs utilize considerable variety of physiological data. We categorize physiological data into three groups: electrophysiological data, motor physiological data, and miscellaneous physiological data. Examples of electrophysiological data are: electrical scalp recording of brain activity (EEG), muscle activation (EMG), electrical activity in the heart (ECG). Motor physiological data includes eye movement (via eye tracker), human pose (captured using vision technology), muscular movement (via accelerometer), etc. Other miscellaneous physiological data are perspiration, respiration, heart rate, pulse oximeter and so on. To acquire various physiological data, specialized acquisition devices are needed. To realize above mentioned mainstream BCIs, devices to acquire physiological data and generate stimulus are needed.

After surveying the device requirement of various BCI applications, we identify three groups of human interface devices in Figure 5.1 including high-precision physiological data acquisition (DAQ), stimulus, and presentation. Applications present auditory, visual or tactile stimulus to invoke users’ physiological response. BCI acquires physiological information via sensing mechanisms about the functions of living organisms and their parts. Applications can also present system information and instructions to users in the forms of auditory cues or visual display. Applications infer user’s intent based on the physiological evidence, and enact the physical interaction on the human’s behalf, such as controlling a prosthetic limb or a robotic wheelchair. BCI applications requires concerted efforts of these essential devices to interface with the human.
For wide range of BBCI applications, three types of devices are needed to interface with the human, including *DAQ*, *stimulus*, and *presentation*. All BBCIs acquire physiological data for intent inference; some BBCIs utilize stimulus to invoke physiological response and presentation to provide feedback.

5.2 **BODE Suite Overview**

To simplify interfacing with the human in BBCI applications, we introduce an affordable, open device suite, *BODE Suite* as depicted in Figure 5.2, for physiological *DAQ*, *stimulus*, and *presentation*.

Physiological acquisition is essential for all BBCIs. Electrophysiological signals, such as EEG, EMG, ECG, are widely used in BBCIs, but off-the-shelf high-precision acquisition devices can be cost prohibitive. Other physiological data such as eye movement can be captured using affordable off-the-shelf devices like EyeTribe, TobiiEyeX, and Pupil. Heart rate, respiration and perspiration can also be obtained using cheap bio-sensors. To cover a wide range of BBCI applications, we propose *EEGu2* (Section 5.3.1) for affordable, accurate electrophysiological signal acquisition that otherwise is cost prohibitive using commercial *DAQ*. Our *EEGu2* provides high-resolution acquisition with wide dynamic range, which supports all types of electrophysiological signals used in prevailing BBCIs. Nevertheless, other physiological data, such as eye tracking data (after feature extraction) as four channels of X-Y position of left and right eye gaze, can be integrated into the *DAQ* firmware and supported by an abstract *DAQ* API that returns samples corresponding to a particular channel and time index.

Some BBCIs utilize visual, tactile, or auditory stimulus to elicit the physiological response. Visual and tactile stimulus is the most commonly used stimulus for mainstream stimulus-based BBCIs,
such as VEP and ERP based BBCIs. These BBCIs require accurate stimulation with configurable
frequency (for SSVEP and SSSEP) and bit pattern (for c-VEP). To support these stimulus-based
BBCIs, our *BODE Suite* incorporates *StimTron* (Section 5.4) that generates high-precision visual and
tactile stimuli with run-time configuration of bit pattern and frequency. That being said, an auditory
stimulus can be obtained by easily extending our *StimTron*.

For BBCI applications that present feedback and instructions to users, a visual display is
most commonly used. To benefit most application requiring presentation, our suite provides *PstVis*
(Section 5.5) with portable display for visual presentation. To allow visual stimuli tightly coupled in
space with the display (e.g. for VEP-based BBCI), our *PstVis* integrates a rail car system for flexible
visual stimulus positioning around the screen.

Section 5.6 discusses the rapid BBCI development using our suite. Utilizing our manufacturer-
agnostic programming interfaces in MATLAB, designers can prototype BBCI algorithms in MAT-
LAB while conveniently interfacing with the human using our suite. The suite is further integrated
into our framework, which enables a modular, distributed composition as well as automated applica-
tion deployment on our suite devices.

### 5.3 EEGu2: DAQ and Embedded Processing

To interface with human, BBCI relies
on a variety of physiological signals, such as
through EEG, ECG, EMG, EOG, etc. For this,
BBCI requires a very high resolution and low
noise *DAQ* for very small bio-signals such as
EEG from 1 to 100µV. In addition, to mea-
sure EEG and EMG (up to 10mV) at the same
time, a wide dynamic range is needed. Further-
more, battery powered operation with embedded
processing capability is required for portable de-
ployment of intent inference algorithms.

To address the above challenges, we propose *EEGu2*, an affordable, portable (battery
powered) *DAQ* with 16-channel 24-bit bio-signal acquisition, embedded processing and intelligent
power management for mobile operation (up to 12hours). Figure 5.3 depicts the *EEGu2* with the
upper cover removed. Section 5.3.1 first discusses the *EEGu2* architecture design for embedded
processing, acquisition, power management, and isolation. Then Section 5.3.2 analyzes the firmware design from low-level acquisition to user interfaces. Section 5.3.3 discusses the quality of real-time acquisition.

### 5.3.1 EEGu2 Architecture

EEGu2 requires an embedded processing component for reading samples from the analog front-end (AFE) of bio-signal acquisition as well as the deployment of BBCI algorithms. Compared with Arduino (ARM Cortex-M3), BeagleBone Black (BBB) Rev.C provides more powerful processing capability (ARM Cortex-A8 AM3358 1GHz) for running complex algorithms. While Raspberry-Pi 2 provides similar processing capability, BBB additionally incorporates two Programmable Real-time Units (PRU) for real-time processing and I/O operations, which makes BBB particularly suitable for the base platform of a real-time DAQ. Since we want to use the same processing platform for different devices in our suite, BBB is also a preferred platform due to its high-performance processing, fast storage (on-board eMMC flash), and generous I/O availability. Therefore, our EEGu2 integrates a BBB for data acquisition and streaming to PC as well as application deployment.

To acquire electrophysiological signals, EEGu2 incorporates a custom-made cape stacking on top of BBB. Figure 5.4 overviews the architecture of the cape, composed of a power board and an acquisition board. The power board (in the upper dash box) provides digital power 5V to BBB (bottom left) and analog power to the DAQ board (in the bottom right dash box). The vertical double
line marks the signal and power isolation between the digital and analog circuit islands for user protection. The acquisition board collects 16-channel bio-signals from the human, digitizes them and sent them to BBB over SPI through proper isolation. The next subsections describe the components in more detail.

### 5.3.1.1 DAQ Board

Figure 5.5a and Figure 5.5b depict the physical board and conceptual signal flow between components. Human bio-signals sensed from passive electrodes travel through protection circuit to the central analog front end (AFE). The protection circuit (IEC60601 standards [53]) limits the current flow to the patient. The AFE provides low-power high-performance signal acquisition using two state-of-the-art ADS1299 chips. Each ADS1299 contains a 24-bit delta-sigma analog-to-digital converters (ADC) and programmable gain amplifiers (up to 24x) for all channels. This high resolution ADC is necessary for sensing bio-signals with extremely low voltage level, such as EEG (around 10\(\mu\)V). The dynamic range is \(\pm 4.5V\) at the gain of 1. Even with the finest gain of 24, a dynamic range of \(\pm 187.5mV\) suffices for concurrent mixed bio-signals measurement (such as simultaneous recording of EEG and EMG) and tolerates high noise level without ADC saturation. The sampling frequency of ADC can be configured from 250 samples per second (SPS) to 16KSPS to target different bio-signals with their own frequency range. The available high sampling frequency allows researchers to utilize bio-signals with high frequency (like EMG up to 500Hz). Also, ADS1299 is optimized for bio-signal sensing offering lead-off detection, bias sensing, and bias drive amplifier as
CHAPTER 5. BODE SUITE: OPEN HARDWARE SUITE FOR BBCI

5.3.1.2 Power Board

To enable portable, battery-powered operations, EEGu2 integrates a custom-made power board. Figure 5.6 illustrates the power board for battery and wall charging. The power board contains a BQ24172 power management IC (PMIC) that takes 12V input from an adapter connected to the wall outlet. The PMIC outputs the 7.6V rail and charges two lithium batteries with LED status indicator. The power board is orthogonally attached to the battery slots so that batteries can be slid in and switched just like the battery in a cellphone. Standard cellphone connectors are installed on the board to support affordable commercial cellphone batteries. EEGu2 incorporates two Samsung Galaxy Note 4 battery (3200mAh, 3.8V) to provide up to 12-hour mobile operation (see power measurement in Section 5.3.3.1). Those two batteries in series provide 7.6V to power BBB (needing 5V) and DAQ board. The system buck regulates 7.6V to 5V for BBB and several low-dropouts (LDO) further regulate the 5V to 3.3V and ±2.5V to power the DAQ board.

To protect users, the AFE (analog domain) directly connecting to the users is isolated from the processing platform (digital domain). Figure 5.7 illustrates the power and signal isolation between the analog and digital domain of EEGu2. SPI and GPIOs signals are isolated using ADuM6401 isolators (each with 4 channels) at the analog/digital border. Also, the 5V power rail from the system buck (digital island) to the analog island is isolated by a transformer rectifier.

All power rails contain several levels of voltage regulation with the trade-off between noise and efficiency (listed in Table 5.1). A high efficient power buck drives both BBB and DAQ...
board. However, the buck must be isolated (via transformer rectifier) due to its high switching noise (65KHz) and ripples that could pollute very small sensed signals (e.g. EEG). LDOs further regulate the power with much less noise and therefore reside in analog island. Although LDOs have under 75% regulation efficiency, the power consumption of LDOs driving ADS1299 is trivial (7.6mA, ±2.5V). Therefore, we directly use LDOs in the analog island. Rather than isolating each LDO output, only one isolator is needed to isolate output of the buck.

<table>
<thead>
<tr>
<th>Component</th>
<th>Noise</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>System Buck TPS562209</td>
<td>high</td>
<td>93% @5V, 350mA</td>
</tr>
<tr>
<td>Transformer Rectifier SN6501</td>
<td>n/a</td>
<td>53% @10mA</td>
</tr>
<tr>
<td>LDO TPS76333</td>
<td>low</td>
<td>65.14% @7.6mA</td>
</tr>
<tr>
<td>LDO TPS76325</td>
<td>low</td>
<td>74.77% @7.6mA</td>
</tr>
<tr>
<td>Charge Pump TPS60403</td>
<td>n/a</td>
<td>95% @10mA</td>
</tr>
</tbody>
</table>

Table 5.1: Voltage Regulation Noise and Efficiency

Overall, the power board powers both the acquisition board and the BBB with the consideration of low-noise acquisition and analog-digital isolation for user protection. The power board can charge the battery while EEGu2 is in use. And a fully charged EEGu2 can operate up to 12 hours on battery when fully loaded.
CHAPTER 5. BODE SUITE: OPEN HARDWARE SUITE FOR BBCI

![Image](EEGu2.png)

(a) Front View  (b) Perspective  (c) Battery Cover Unlatched  (d) Side View

Figure 5.8: EEGu2 3D Printed Enclosure

5.3.1.3 Mechanical Enclosure

To protect the circuitry and allow for portable operation, EEGu2 incorporates a 3D printed acrylic enclosure as shown in Figure 5.8. Figure 5.8d shows the DAQ board horizontally stacking over BBB and the power board (front) vertically connected to the DAQ board and batteries. To provide stable mechanical support to BBB and the cape, the BBB is installed in the enclosure through four mounting holes. Using a detachable battery cover shown in Figure 5.8c, users can easily slide batteries in and out for replacement. The enclosure selectively exposes a translucent power button, an Ethernet port, a power jack and a USB port for WiFi dongle. For user’s convenience, the enclosure has a belt clip on the side for wearable BBCI applications.

5.3.2 EEGu2 DAQ Firmware

The EEGu2 firmware interfaces with the ADS1299 chips (for control and sample read) and provides an abstracted interface to the user program. To read samples from the ADC, the firmware waits for sample ready signal (DRDY) and then reads samples from ADS1299 via SPI to the memory. However, handling DRDY event and reading samples in a timely manner is challenging. While improving the performance of real-time acquisition, we developed three versions of DAQ firmwares: a user-level driver, a kernel driver and a driver using the co-processor (PRU).

An initial approach is based on a user-level driver that polls DRDY signal and then reads samples via SPI in the user space. However, the driver is limited by the OS scheduling. The driver has to compete (for being scheduled for execution) with other user-level processes and furthermore forcibly yield to all kernel processes. When the delay due to waiting for scheduling is larger than a sampling period, the driver loses the sample in that period. Details of real-time analysis can be found in Section 5.3.3.3.
To improve the performance and avoid scheduling competition with user tasks, a more robust kernel-level driver was developed. Figure 5.9a depicts the kernel driver that waits for DRDY events via interrupts in kernel space. The kernel driver is based on Industrial I/O Subsystem (IIO) [54], a standard kernel driver structure (proposed by Analog Devices, Inc) for a class devices involved with ADCs or DACs. The DRDY signal of each sample triggers a two-level interrupt handling. The lower level GPIO interrupt service routine (ISR) invokes the higher level IIO ISR to read samples in a separate kernel thread. All channel samples are interleaved and pushed into the tail of the IIO circular buffer. While kernel driver handles each sampling event as an interrupt, an increasing sampling rate results in an increasing interrupt rate burdening the processor. Our experiment (in Section 5.3.3.3) shows the kernel driver can only support up to 1KHz sampling rate.

To fully explore the maximum sampling rate of ADS1299 (16KHz), we utilize an additional co-processor PRU to offload the low-level acquisition from CPU and reduce the interrupt rate. Figure 5.9b depicts the our kernel-PRU driver. The PRU runs independently from the CPU and has direct SPI and GPIO pin access. PRU polls the low-level DRDY event (via GPIO). When data is ready, it uses SPI to receive the samples and transfers the data into a buffer (we use a double buffer scheme).

When a buffer is full, PRU generates an interrupt to notify the CPU to read data from the PRU buffer to IIO buffer (shared memory between CPU and PRU). In the meantime, PRU
CHAPTER 5. BODE SUITE: OPEN HARDWARE SUITE FOR BBCI

switches to the other buffer for data acquisition in parallel with CPU reading data. Buffering samples and reducing the granularity of interacting with the CPU reduces the interrupt rate of the CPU for acquisition. Results show a consistent delay of $8 \mu s$ from DRDY event to data read via SPI. This constant very low delay is achieved due to the pulling loop implementation in PRU assembly. It always executes the same execution path. In addition, it does not use any external memory (DRAM) which would incur a variable access latency. Instead, fast dual-port scratchpad memory (shared with the CPU) is utilized. Any operations that can alter the timing (e.g. writing into DRAM) are performed in the kernel running on the CPU. It picks up the data from the scratchpad and interfaces with the user program.

<table>
<thead>
<tr>
<th>File</th>
<th>Description</th>
<th>Read / Write</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>trigger0/name</td>
<td>Device name</td>
<td>R</td>
<td>eegu-dev0</td>
</tr>
<tr>
<td>current_trigger</td>
<td>Device triggering the recording. If eegu, then self triggered.</td>
<td>W</td>
<td>eegu-dev0</td>
</tr>
<tr>
<td>in_voltage[n].en</td>
<td>Enables sampling of channel N</td>
<td>R/W</td>
<td>1</td>
</tr>
<tr>
<td>sampling_frequency</td>
<td>EEGu2 sampling frequency for all channels</td>
<td>R/W</td>
<td>250</td>
</tr>
<tr>
<td>buffer/length</td>
<td>Number of samples to store in IIO ring buffer</td>
<td>R/W</td>
<td>1024</td>
</tr>
<tr>
<td>buffer/enable</td>
<td>Enable buffering in ring buffer</td>
<td>W</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 5.2: EEGu2 Configuration and Control Interface via VFS

To interface with user programs, both kernel driver and kernel-PRU driver provide user interfaces via virtual file systems (VFS). Table 5.2 overviews user interfaces through sysfs character device files accessing driver configurations, including trigger registration, channel gain, sampling frequency and buffer size and enable. Some configurations like circular buffer enable and size are driver software configuration; other configurations such as channel gain and sampling frequency take effect in the acquisition hardware.

We discussed three DAQ drivers with increasing performance of real-time acquisition. Our kernel-PRU driver has the best real-time performance and can support up to 16KSPS sampling frequency. Detailed real-time analysis of the all three firmware implementations is reported in Section 5.3.3.3.
5.3.3 Experimental Results

This section evaluates both the HW quality (power consumption and accuracy) and firmware quality (real-time and end-to-end delay) of EE Gu2.

5.3.3.1 Power Consumption

As a portable, battery powered embedded system, the power consumption of EE Gu2 is critical. Table 5.3 lists the measured power consumption of EE Gu2 components. When system is idle (DAQ board idle, BBB idle), EE Gu2 consumes 1219.5 mW in which the cape consumes only 69.5 mW. When running the DAQ driver (DAQ board busy) and DevClass-Remote for data streaming to PC (BBB busy), EE Gu2 consumes 1951.2 mW in total, that being 731.7 mW more in comparison to the idle state. While BBB contributes 1850 mW the majority of the overall power consumption due to embedded acquisition and WiFi communication, the cape consumes only 101.2 mW operating signal acquisition.

EE Gu2 is equipped with two Samsung Galaxy Note 3 lithium ion battery, each with 3200 mAh at 3.8 V. The capacity of the batteries translates to \((2 \times 3200 \text{ mAh} \times 3.8 \text{ V}) / 1951.2 \text{ mW} = 12.46 \text{ hour}\) battery operation with assistive applications running on EE Gu2, which suffices for most applications.

<table>
<thead>
<tr>
<th>Component</th>
<th>Idle Power (mW)</th>
<th>Busy Power (mW)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cape</td>
<td>69.5</td>
<td>101.2</td>
</tr>
<tr>
<td>BeagleBone Black</td>
<td>1150</td>
<td>1850</td>
</tr>
<tr>
<td>Total</td>
<td>1219.5</td>
<td>1951.2</td>
</tr>
</tbody>
</table>

Table 5.3: EE Gu2 Power Consumption

Future optimization on the power consumption can look at more aggressive power management to reduce power consumption of the BBB. This might enable operating with a single battery, reducing both weight and size of EE Gu2. Also, for mass deployment of a particular application, more customized lower power compute platforms can be explored.

5.3.3.2 Accuracy

The accurate bio-signal acquisition is crucial for the human intent extraction from the acquired signals. We evaluate accuracy as signal-to-noise ratio (SNR) and input-referred noise (noise...
flow). Our experiments show an average 25dB SNR over 10 trials, each of which contains 250x10 samples with the gain of 24, sampled at 250Hz. In comparison, the commercial DAQ product, g.USBamp ($15k) shows 21.35dB SNR with hardware filter off and 24.61dB SNR with hardware filter on (1-30Hz).

To quantify the internal noise in AFE analog circuit, we analyze the input-referred noise (IRN) with shorting positive and negative analog inputs. Our experiment shows 0.785μV Vpp (peak to peak) and 0.126μV Vrms (root mean square) IRN. We use the same test configuration as reported in ADS1299 manual, 1000 samples with gain of 24, sampled at 250Hz sampling frequency with a 0.01Hz-70Hz bandpass filter applied. Figure 5.10 illustrates the time signal and amplitude histogram of IRN. The Gaussian distribution in amplitude histogram indicates that IRN is a white noise. This measured IRN of EEGu2 is slightly smaller than 0.98μV Vpp and 0.14μV Vrms reported in ADS1299 manual. Our low IRN may result from chip to chip variation. It also indicates a valid routing in our PCB design and effective digital-analog isolation.

Figure 5.10a depicts the trend of decreasing IRN with larger channel gain setting (equivalently reducing internal noise). This indicates the analog noise is injected after the amplification so that the noise is effectively scaled down by the gain.

Our EEGu2 with high SNR and low IRN acquires high-quality bio-signals, which enables accuracy intent inference in BBCI applications.
5.3.3.3 Real-time Analysis of AFE Communication

Since the AFE chip signals the availability of every single sample and requires timely retrieving the sample to avoid data loss, strict real-time behavior is needed to correctly realize the protocol. Figure 5.11 depicts the AFE communication protocol reading samples from AFE. When DRDY signals data is available on the AFE chip, the processor reads data via SPI. AFE periodically generates new data overwriting old data, which imposes a hard real-time constraint, i.e. a deadline of reading samples within one sampling period. Since the SPI transaction takes fixed time (at a clock rate), the response time $t_{\text{delay}}$ from DRDY to SPI read (CS), is critical. We measure the delay between two signals using the logic analyzer and take average delay over one minute trial (recorded and post-processed in MATLAB).

Figure 5.12 illustrates the cumulative distribution (CDF) of DRDY to CS delay for our three driver implementation - user driver (the initially developed driver in user space only), kernel driver and kernel-PRU driver. Each driver is evaluated with and without additional system load (4 dummy applications of busy loops) to examine the driver’s susceptibility of OS scheduling (for user-level applications). The sampling frequency is set to 250Hz resulting in a 4ms deadline. In Figure 5.12 the x-axis is the delay and y-axis shows the cumulative percentage of samples with a smaller delay. For example, a coordinate (x, y) in the graph indicates y percent of total samples have a delay equal to or smaller than x (accumulated from 0 to x). Table 5.3 lists the statistics for the same experiment.

Figure 5.12 shows that the user driver performance deteriorates rapidly with system load. Without system load, the delay of 90% samples ranges from 20µs to 40µs. However, the maximal delay of 4488µs indicates sample loss. When system is loaded, more than 51% of the samples fail to meet 4ms deadline because the driver has to compete with other user processes for OS scheduling. Hence, this user driver is not reliable.

In contrast, kernel driver shows an average delay of 189.40µs latency with system load and
CHAPTER 5. BODE SUITE: OPEN HARDWARE SUITE FOR BBCI

287.22µs otherwise (see Table 5.4). The delay results from the kernel context switch and interrupt handling (reading samples from AFE and storing them into IIO buffer). The two tight curves indicate that the kernel driver is much less independent from the system load due to its higher priority as kernel processes against user processes. Note that kernel driver even exhibits a better performance with system load because the BBB frequency scales up when system is loaded (lower frequency when idle). Interestingly enough, the kernel driver average delay is much larger than the user driver. This additional delay of kernel driver results from the overhead of IIO subsystems for interrupt handling and sample buffering. Since the kernel driver reacts to DRDY event in kernel space, the high volume of the interrupt request for each sampling limits sampling rates up to 1KHz.

<table>
<thead>
<tr>
<th>Latency (Unit: µs)</th>
<th>Mean</th>
<th>STD</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>User Driver w/o Load</td>
<td>74.44</td>
<td>353.92</td>
<td>4488.00</td>
</tr>
<tr>
<td>User Driver w/ Load</td>
<td>4229.05</td>
<td>3805.45</td>
<td>20756.00</td>
</tr>
<tr>
<td>Kernel Driver w/o Load</td>
<td>287.22</td>
<td>37.84</td>
<td>626.00</td>
</tr>
<tr>
<td>Kernel Driver w/ Load</td>
<td>189.40</td>
<td>22.51</td>
<td>520.00</td>
</tr>
<tr>
<td>Kernel-PRU Driver w/(o) Load</td>
<td>8</td>
<td>0</td>
<td>8</td>
</tr>
</tbody>
</table>

Table 5.4: EEGu2 AFE Real-time Analysis
CHAPTER 5. BODE SUITE: OPEN HARDWARE SUITE FOR BBCI

In comparison to kernel driver, kernel-PRU driver has only $8\mu s$ delay consistently for all sample read. The PRU real-time response to sampling is independent from the system load on the main processor. Furthermore, as the kernel-driver aggregates multiple samples before interrupting the CPU, it allows the driver to support much higher sampling rate without overloading CPU. Our kernel-PRU driver enables EEGu2 to fully utilize maximum 16KHz sampling rate of ADS1299. Beyond that, the SPI clock frequency becomes next bottleneck. Given the same setup, our PRU driver can support up to \( \frac{1}{((1/48MHz) \times 216 + 8\mu s)} = 80KHz \) sampling rate (BBB SPI master max speed 48MHz). The polling of the PRU is not limiting the performance.

In short, the kernel-PRU driver demonstrates the best real-time performance of acquisition among three driver implementations. The kernel-PRU driver provides reliable acquisition for the full range of ADS1299 (16KHz), which enables BBCI researchers to utilize physiological signals in a wide frequency range.

5.3.3.4 Firmware End-to-End Delay

Since BBCI applications interacts with the human in real-time, it’s desired to have the change of human status (physiological signals) visible to applications as soon as possible. End-to-end delay measures the delay from the time an analog signal appears at the input to the time the sample can be read at the DAQ HW API. This response time of EEGu2 evaluates how fast the EEGu2 responds to the environment (input change). To get an overview, we first evaluate the end-to-end delay of all three firmware implementations. The experiment writes a GPIO connected to an analog input of EEGu2 sampled at 250Hz and measures the delay until the event being observed at the user level. The user-level program is running at the highest priority (nice -20) without any other workload. The result shows $10.7ms$, $12.3ms$ and $3.5ms$ average end-to-end delay for user driver (no system load), kernel driver and kernel-PRU driver (PRU buffer size 1) respectively. The kernel driver has slightly longer delay than user driver due to the IIO overhead, including the two-level interrupt handling. In comparison, the PRU-kernel driver largely reduces the delay by running low-level acquisition in PRU. While BBCI application are loosely timed and infer user’s intent at seconds level, the delay of all three drivers is at millisecond granularity, which can be negligible for most of intent inference algorithms.

To better understand the impact factors, we look into the kernel-PRU driver as it is the best performing interface. Three contributors to the delay can be identified: a) data buffering at various levels, b) code execution, and c) system scheduling. The scheduler decides the execution sequence.
of the firmware (kernel-level ISR and user-level library) and other kernel/user processes. While the code execution time (b) is static, the scheduling delay (c) differs with various priority and policy settings, as well as the load. Since we use an unmodified BBB Linux kernel 3.8.13-bone72 (not real-time OS), bounded scheduling delay is not guaranteed. Instead of the code execution and scheduling, we focus on the delay due to buffering introduced by our firmware design as it directly corresponds to design choices of users and driver developers.

Figure 5.13 depicts the delay due to buffering at both PRU and user level. When an external event is captured at sample 3, PRU still need to acquire another 4 samples before transmitting the complete PRU buffer (8 samples as an example) to kernel IIO. Similarly, when user-level program calls to read $N$ samples, e.g., $num = \text{read}(fd, buffer, N \times sizeof(int))$, IIO needs to buffer the requested $N$ samples and then return them at once. Buffering the following samples acquired after the event introduces a delay depending on location in buffer and sampling frequency for that event to surface to the user program. While a larger buffer size introduces a larger average end-to-end delay, buffering more samples can reduce the interrupt rate of the CPU and hence decrease the CPU usage. Conversely, a smaller buffer size causes smaller delay but higher interrupt rate and CPU usage.

<table>
<thead>
<tr>
<th>User Read</th>
<th>No Buffering</th>
<th>Size 1</th>
<th>Size 32</th>
<th>Size 128</th>
<th>Size 512</th>
</tr>
</thead>
<tbody>
<tr>
<td>Delay</td>
<td>3.9ms</td>
<td>7.17ms</td>
<td>6.85ms</td>
<td>15.96ms</td>
<td>68.3ms</td>
</tr>
</tbody>
</table>

Table 5.5: EEGu2 End-to-end Delay Due to Buffering

We evaluate the buffering impact on the end-to-end delay of kernel-PRU driver at sampling
CHAPTER 5. BODE SUITE: OPEN HARDWARE SUITE FOR BBCI

Figure 5.14: Kernel-PRU Driver End-to-end Delay

rate 4096Hz which requires PRU buffering (e.g. size 32) to reduce the interrupt rate. Figure 5.14 demonstrates the CDF of end-to-end delay when user read 1, 32, 128, and 512 samples at a time. Table 5.5 reports the average delay of $7.17\text{ms}$, $6.85\text{ms}$, $15.96\text{ms}$, and $68.3\text{ms}$ for various user read size respectively. Considering the location of the event in the buffer(s), the delay due to acquiring the remaining samples to completely fill the buffer can be computed. We also compute the delay excluding the buffering effect, $3.9\text{ms}$ which is obtained by subtracting the sampling time of all samples after that event signal in the buffer. The delay excluding buffering effect reflects the delay due to code execution and OS scheduling, which sets a lower bound of all other delay measurements in Figure 5.14. The user could observe this delay if the event would always occur in the last sample of a completely filled buffer. In contrast, the event signal takes $68.3\text{ms}$ on average to appear at user-level when the user reads 512 samples at a time. Since all event signals are issued at random time, the event sample is uniformly distributed in the user requested 512 samples. Statistically, the additional delay due to reading 512 samples is the sampling time of half of those samples, i.e. $64\text{ms}$ which is very close to the measured delay difference ($68.3\text{ms} - 3.9\text{ms}$). The delay linearly decreases from read size of 512 to 128 and 32. However, the delay no longer decreases when the user reads only one sample at a time. At this point, the delay due to PRU buffering is exposed when user read less samples than PRU buffer size. Therefore, user gets the fastest system response (to analog input) when each time reading the same number of samples as PRU buffer size. Matching user read size
with a given PRU buffer size allows applications to observe the physiological change at the analog input as soon as possible.

While decreasing PRU buffer size (and user read size accordingly) can reduce the end-to-end delay, it will reach to a limit of interrupt rate that the CPU can handle. Figure 5.15 illustrates the trade-off between CPU utilization and delay due to PRU buffering. The result is measured at 4096Hz sampling rate and counteracts the user read buffering effect. While decreasing PRU buffer size reduces the delay, it increases the CPU utilization of interrupt handling due to the increasing rate of CPU interrupts. The delay decreases proportionally to PRU buffer size from 64 to 8, but stops decreasing from 8 to 1. We observe sample loss with PRU buffer size 4 and it gets worse with buffer size 1. This indicates that CPU interrupt rate 1024Hz starts to overload the CPU so that CPU cannot respond to PRU interrupt request to read buffer in a timely manner. Therefore, for any sampling frequency, the firmware can pick the smallest PRU buffer size that incurs interrupts at less than 1024Hz rate for the fastest real-time response without overloading the CPU.

Overall, our EEGu2 provides 16-channel 24-bit high-precision acquisition with battery powered embedded processing (up to 12hours) integrated. Our EEGu2 supports up to 16KSPS real-time acquisition, which can benefit all BBCI applications utilizing electrophysiological data.
5.4 \textit{StimTron}: Multisensory Stimulus

Some BBCIs, such as VEP and ERP based BBCIs, utilize visual, tactile, or auditory stimulus to elicit the physiological response. These BBCIs require accurate stimulation with configurable frequency (for SSVEP and SSSEP) and bit pattern (for c-VEP). For example, in SSVEP-based BBCIs, users are presented with a set of flickering stimuli associated with system actions. User can select an action by focusing on the associated stimulus. The application infers the user intent of selection from the invoked EEG response. This type of BBCIs requires multiple channels of individually configurable stimulation. To support these stimulus-based BBCIs, we propose a multisensory \textit{Stimulus} device - \textit{StimTron} that generates 12-channel visual and tactile stimuli with runtime configuration of stimulation frequency, bit pattern and optional brightness.

Figure 5.16 overviews the \textit{StimTron} hardware design. Similar to \textit{EEGu2}, \textit{StimTron} also uses BBB for embedded processing. Generating synchronized stimulus signals of 12 channels is beyond the capability of the CPU and also the PRU in BBB Implementing the stimulus on PRU would require some common sample rate across all channels. To eliminate restrictions of a common sample rate, and to allow for fine grained PWM control, we opted for a hardware-based implementation. Instead, our \textit{StimTron} incorporates a custom-made cape with a FPGA (Xilinx Spartan3E XC3S500E) on board for parallel execution and stimulus signal output. To generate stimuli with accurate timing, our \textit{StimTron} decouples the real-time stimulus output generated by FPGA from the loosely-timed software running on BBB. The software on BBB configures FPGA to produce 12-channel stimuli in the range from 0.1Hz to 409.6Hz with 0.1Hz span which is sufficient for most BBCI. Since the FPGA can achieve much higher I/O frequency (100MHz), our \textit{StimTron} can support larger frequency range and finer resolution if needed. Utilizing the high-frequency parallel execution of FPGA, \textit{StimTron} modulate each channel of stimulus signal with 20KHz pulse width modulation signal (PWM) to

![Figure 5.16: StimTron: Visual and Tactile Stimulus](image)
control the brightness level. Each channel outputs two signal that can be configured identical or alternating to form a checker box. To allow flexible stimulus positioning, StimTron integrates 2.5mm interfaces providing two signals to custom-made LED Arrays (two addressable groups of surface mount LEDs) for visual stimulus and C-3 tactors (Engineering Acoustics, Inc.) for vibrotactile stimulus. Since the required current for driving a LED array is beyond the FPGA I/O capacity, the stimulus signal controls dedicated power rails to LED arrays.

Overall, StimTron provides 12 channel high-precision visual and tactile stimulus (generated by FPGA) with run-time configuration of frequency and bit pattern (user interface in BBB).

### 5.4.1 StimTron Firmware

Figure 5.17 illustrates the StimTron firmware. The firmware comprises the stimulus backend on BBB implementing a canonical HW API (see Section 4.1.1) and stimulus controller in FPGA. The backend provides standard Stimulus API to configure stimulation frequency, bit pattern and brightness for individual channels. The stimulus configuration is written to the dedicated register files in stimulus controller. The controllers runs a set of finite state machines to output the stimulus signal following the bit pattern at a reduced clock frequency (bit rate). Each channel has its an own clock divider to obtain a given frequency. The firmware computes the number of FPGA cycles to count for one period of target frequency. Then that counter value is written to the frequency register and used to downsample the FPGA clock to stimulus frequency. Then, each channel clocks out a bit...
pattern of up to 1024 bits. If only a constant frequency is desired, a bit pattern of 0,1 is repeated given the target frequency. Each channel has two signal outputs which can be configured either to alternate 1 and 0 (checker box) or to output the same signal synchronously. The stimulus signal modulated with PWM with configurable duty cycle controls the average output voltage and consequently brightness level. For instance, 50% duty cycle of PWM reduces the LED brightness by half. The high frequency noise introduced by the PWM is irrelevant and beyond the interested frequency range of BBCI. The firmware also provides flexible mask for each channel so that the application does not lose the stimulus status of temporarily disabled channels. This mask is particularly useful in controlling bit stream stimuli (c-VEP). Since BBCI applications usually need to know the time frame of valid physiological signal during stimulation, StimTron additionally outputs a trigger signal - a TTL that stays high during stimulation (level sensitive).

5.4.2 Experimental Results

The accurate stimulus frequency is critical in the intent inference of some BBCIs such as SSVEP-based BBCI. To demonstrate the quality of our StimTron, we compare the measured StimTron signal quality against the stimulus presented on a traditional desktop monitor (Dell E2416HM 60Hz). A photodiode was placed close to the LED array and flashing area on the monitor to measure the

Figure 5.18: StimTron vs Monitor Signal Quality @8Hz
actual stimulus signal. The photodiode output was recorded at 1200Hz sampling rate. While available frequencies of monitor stimulus are natively limited to the submultiples of the monitor refresh rate, other frequencies can be approximated by mixing multiple frequencies using Psychtoolbox [55]. For example, 6Hz stimulus can be realized with 10 bits “1111100000”, one bit per frame. But 8Hz stimulus has a period of 7.5 frames which can be approximated by concatenating 7 bits “1110000” and 8 bits “11110000” as shown in Figure 5.18. However, this frequency approximation introduces lots of noise, visible in the spectrum. Furthermore, the measured monitor stimulus frequency is 7.865Hz, deviated from 8Hz. In contrast, StimTron generates periodic signal with clean 8Hz and third harmonic 24Hz components in the spectrum. Note that ideal rectangle signal only contains odd harmonics, which are observable in our measurements. This indicates that the StimTron generates a close to ideal rectangle signal.

To quantify the stimulus signal quality, we measured the SNR of stimulus from 1Hz to 20Hz with 1Hz span, each frequency measured for 30 seconds. Figure 5.19 depicts the signal-to-noise ratio (SNR) of measured stimulus signal of StimTron and monitor. StimTron produces consistent accurate stimuli over all frequencies with an average SNR of 22dB. In contrast, monitor signal only has around 5dB SNR at 1Hz and 2Hz and quickly decreases to −20dB or lower for higher frequencies. The low SNR of monitor stimulus results from the inaccurate stimulus frequency which is in turn regarded as noise in the SNR calculation.

In summary, StimTron provides 12 channel high-precision visual and tactile stimulus with large frequency range from 0.1Hz to 409.6Hz, fine-grained resolution at 0.1Hz, and high signal quality (SNR above 20dB). Our StimTron is flexible to control each individual channel with run-
CHAPTER 5. BODE SUITE: OPEN HARDWARE SUITE FOR BBCI

Figure 5.20: PstVis: Visual Presentation

time configuration of frequency, bit pattern, and optional brightness. Our StimTron enables BBCI developers to prototype stimulus-based BBCIs using ready-made devices with high quality stimulus signal generation and flexible configuration.

5.5 PstVis: Visual Presentation

The BBCI Presentation subsystem provides user feedback regarding system status and instructions. The Presentation can be carried in various means such as a visual display or an audio cue. In a laboratory setup, a desktop monitor is commonly used to present visual feedback to users. However, a cumbersome monitor with AC power supply from the wall outlet does not qualify for an embedded, portable BBCI application. Our BODE Suite provides an embedded visual presentation device - PstVis. Figure 5.20 depicts PstVis that integrates a 10.1-inch 1280x800 LCD (12V/2A DC) controlled by a BBB mounted behind the screen. The DC power supply of our PstVis allows it to be integrated in a portable setting. For example, our brain-controlled wheelchair application described in Section 3.3 incorporates a portable battery (DC) installed in the base of the wheelchair which can power our PstVis for portable display. All electronic components are protected in a laser cut acrylic enclosure with VESA compatible mounting holes. Our PstVis can be easily mounted on standard monitor stands or arms.

The PstVis additionally incorporates a rail car system around the enclosure to allow flexible visual stimulus positioning. Users can easily rearrange the stimulus around the screen for different application requirement and adjust the angle of the stimulus to point to users’ eye for effective stimulation. Figure 5.21 depicts the Solidworks model of the rail car system. 3D printed cars can slide on the L-shape rail and be fixated at any position around the enclosure using a locking thumbscrew. A ball joint connects the car to the claw holing the LED board, which allows researchers
Figures 5.21

to easily adjust the LED array pointing directly to the user’s eyes for best stimulation performance.

5.5.1 Firmware

To capture various BBCI applications, designing a portable presentation firmware for various screen size and processing platform is needed but challenging. To interface various low-level display hardware, we utilize an open-source Simple DirectMedia Layer (SDL) library providing hardware transparent interfaces to draw primitives, such as lines and texts. On top of SDL, firmware implements high-level APIs to display figures and texts wrapped in text boxes. However, like many display libraries, SDL display text using traditional point size (1pt = \( \frac{1}{72} \) inch) in absolute dimensions. For example, the text in font 12pt displayed on a small screen has the same absolute size as that displayed on a large screen. As a result, the text of the same absolute size on a larger screen (placed further from users) is harder for the user to see. Figure 5.22 depicts the scenarios of the visual presentation on a small screen and a large screen. The large screen has to be placed further from the user so that user can see it within the same vision range (without moving head) as seeing a small screen at a closer position. To allow for the same visual effect of text on screens of various sizes, the relative font size and the relative coordinate agnostic to the physical screen size is needed.
Relative Font Size = \frac{\text{display width in inch}}{\frac{1}{72} \times \text{font in point}} \tag{5.1}

To address the challenge of keeping the same visual effects of display on various screen sizes, our \textit{PstVis} provides both relative coordinate and relative font size. The position of text box are specified by two relative coordinates (range from 0.0 to 1.0) of upper left corner and lower right corner. To enable consistent visual effect of texts, the firmware scales texts using a relative font size defined as the number of monospaced character per screen width. Similarly, a figure is positioned on the screen using the relative coordinate of upper left corner and scaled to a the given relative size (keeping aspect ratio). While the top-level application calls to display the text in relative font size, the firmware internally translates the size into points using Equation \ref{eq:5.1} and then calls SDL for low-level text display. In result, the relative font size and coordinate enable a consistent vision effect for various screen dimensions.

\section{5.6 Framework Integration}

Section \ref{sec:3} has discussed our \textit{BAT} framework that expedites the development of embedded assistive applications that augment user interaction with the physical world via BBCI. Our framework provides unified access to similar hardware types independent of their location and domain-specific synthesis for automated embedded deployments.

To interface with various hardware and connectivity, \textit{BAT} framework groups similar HW types into classes, called \textit{DevClass}. Each \textit{DevClass} is a class of HW types that implement the same semantics of input and output. For example, \textit{DAQ} is a \textit{DevClass} sensing bio-signal sand \textit{EEGu2} is one supported HW types of \textit{DAQ DevClass}. Figure \ref{fig:4.4} depicts the structural composition of a \textit{DevClass}. Each \textit{DevClass} (i.e. \textit{DAQ}, \textit{Stimulus} and \textit{Presentation}) implements a unified MATLAB API which provides hardware type and location transparent access from MATLAB. To support various HW types, a \textit{DevClass} contains a set of HW-specific backends that implement the canonical HW API by interfacing with actual HW drivers. The \textit{DevClass} also provides a pair of \texttt{proxy-master} and \texttt{proxy-slave} to allow the application transparently communicate with hardware regardless of its connectivity. The proxies support dedicated TCP connection as well as a communication middleware - Data Distribution Service (DDS). Using \textit{DevClass}, algorithm designers can prototype application interfacing with distributed HW from MATLAB without worrying hardware specific details and
customized communication. The DevClass API also enables a portable assistive application design in that the application benefits from a consistent access to different hardware types within the class.

**BODE Suite** is fully integrated into the framework so that each device - EEGu2, StimTron and PstVis are abstracted by DAQ, Stimulus and Presentation DevClass. Listing 5.1 lists a snippet of DAQ HW API as an example. The `init()` creates a DAQ object (struct instance) and returns its pointer value as the DAQ instance ID. Returning the object pointer value instead of the pointer itself allows the function being called without requiring the caller knowing the complex object definition. This design simplifies interfacing DAQ HW API with MATLAB through MEX (see Figure 4.6) and domain-specific synthesis (code generation for embedded deployment) during the framework integration. Other functions can refer to the DAQ object by casting the DAQ instance ID back. Given the amount of the samples requested per channel, `getSamples()` function returns a sample matrix, each column of which contains one channel samples. Therefore, all channel samples are interleaved in the data array. `samplesAvailable()` refer to the DAQ instance by casting the DAQ instance ID to the actual object and returns the number of sample available samples in the buffer. The `configCh()` function turns on or off a DAQ channel given the channel number. The `setSamplingFreq()` method sets the sampling frequency of the DAQ device. The `getActiveChNum` returns the number of current active channels. The `clearBuffer` empties the buffer and `end()` cleans up the DAQ object. EEGu2 is integrated into the framework as one supported DAQ DevType in that EEGu2 firmware implements the DAQ HW API and serves as a DAQ DevClasses Backend.

With the framework integration, designers using BODE Suite benefit not only from the hardware and location transparent access in MATLAB but also automated embedded deployment.
enabled by domain-specific synthesis, as shown in Figure 5.23. While we use EEGu2 as an example for automated embedded deployment, applications can also be automatically deployed onto StimTron and PstVis. Algorithm designers first design and validate the algorithm model in MATLAB and rapidly expand it to the application with hardware-in-the-loop (using real sensors). The MATLAB application simply instantiates a DevClass (e.g. DAQ) to access a class of hardware (e.g. EEGu2) regardless of the hardware type and connectivity (e.g. TCP). Instead of worrying about hardware details (i.e. HW driver API), designers can focus directly on developing the algorithm. Finally, the domain-specific synthesis tool HSyn automatically synthesizes the prototyped MATLAB application down to an embedded implementation in C/C++ seamlessly interfacing with DAQ Backend library. More details regarding the hardware abstraction of DevClass and automated process of embedded deployment can be found in Section 4.2.

5.7 Summary

This chapter addresses the challenge of needing human interface devices in assistive BBCI applications. To lower the barrier of entry in BBCI for hardware equipment, we propose a package of
open, affordable embedded devices for BBCI, our BODE Suite. The suite comprises a bio-signal DAQ device (EEGu2), a multisensory Stimulus device (StimTron) producing visual and tactile stimulation and a visual Presentation device (PstVis).

EEGu2 comprises a customized cape on top of BBB, offering the 16-channel, 24-bit acquisition up to 16KHz sampling frequency, powerful processing capability (Cortex-A8), and intelligent power management for up to 12 hour mobile operation. The StimTron offers high-precision visual and tactile stimulation with run-time configuration of bit pattern and frequency from 0.1Hz to 409.6Hz at 0.1Hz resolution. Measured visual stimuli show a high signal quality with above 20dB SNR. The PstVis provides portable display with an additional rail car system for flexible stimuli placement.

Our suite is fully integrated into our framework and enables a rapid, cost-effective BBCI application prototyping with ready-made human interface devices and convenient user interfaces from MATLAB.
Chapter 6

Demo Applications

This chapter demonstrates the usability of BAT framework and BODE Suite with three applications: a brain-controlled wheelchair, a brain-controlled robotic arm and a BCI speller.

6.1 Brain-Controlled Wheelchair

We have shown four applications in Section 3.3 to highlight the flexibility of our framework. In this section, we discuss the brain-controlled wheelchair application in more detail to show how the framework aids the development of a complete application. We discuss the development flow using MBD (Section 6.1.2) and the benefit of the hardware abstraction (Section 6.1.3) and domain-specific synthesis (Section 6.1.4).

Figure 6.1: Brain-Controlled Wheelchair Overview
6.1.1 Application

Using our BAT framework, we built a brain-controlled wheelchair application that augments the mobility of LIS users. The configuration of this application was already introduced in Section 3.3. Next, the application principle and realization will be discussed.

Figure 6.1 overviews the application which is executed in two stages: calibration/training and online running. In the training stage, users are asked to focus on each LED array multiple times while EEG is collected (multiple trials per stimuli). The software extracts the physiological information by computing the canonical correlation scores between the EEG at a given trial and linear combinations of sinusoids at the stimulus frequencies. Since the data has been labeled (supervised learning), these features are then used to build a probability distribution of EEG evidence given the attended stimuli. This probability model is used to perform maximum a posteriori inference during runtime, thus allowing only confident decisions and tight integration with context information.

In the running stage, each wheelchair direction (e.g. left turn, right turn, forward, and backward) is associated with an LED array. Users can express their desired motion direction by focusing their gaze on the corresponding flashing LED array. The system will choose the most probable action given all physiological evidence in the form of EEG correlation scores. Alternatively, the system could select the action with maximum canonical correlation to the measured EEG; however, as it will be shown in the results section (fig. 6.7), such decision rule tends to underperform when compared to the probabilistic classifier.
CHAPTER 6. DEMO APPLICATIONS

<table>
<thead>
<tr>
<th>Class</th>
<th>HW Type</th>
<th>Connectivity</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>DAQ</td>
<td>PAL</td>
<td>Remote TCP</td>
<td>Blackfin DSP527 + Analog Device ADS7606 (8 channel, 16 bit resolution)</td>
</tr>
<tr>
<td>DAQ</td>
<td>EEGu2</td>
<td>Remote TCP</td>
<td>BeagleBone Black + TI ADS1299 (16 channel, 24 bit resolution)</td>
</tr>
<tr>
<td>DAQ</td>
<td>g.USBamp</td>
<td>Local</td>
<td>Commercial (16 channel, 24 bit resolution)</td>
</tr>
<tr>
<td>Stimulus</td>
<td>PAL-LED</td>
<td>Remote TCP</td>
<td>Blackfin DSP527 + Xilinx Spartan3E xc3s500e (6 channel LED arrays)</td>
</tr>
<tr>
<td>Stimulus</td>
<td>BBB-LED</td>
<td>Remote DDS</td>
<td>BeagleBone Black + Xilinx Spartan3E xc3s500e (12 channel LED arrays)</td>
</tr>
<tr>
<td>Visualization</td>
<td>PC-Monitor</td>
<td>Local</td>
<td>24 inch monitor</td>
</tr>
<tr>
<td>Visualization</td>
<td>BBB-LCD</td>
<td>Remote DDS</td>
<td>BeagleBone Black + 10 inch LCD (HDMI)</td>
</tr>
</tbody>
</table>

Table 6.1: Demo APP Hardware Types and Connectivity

6.1.2 System Realization

Our design flow accelerates the application development. We first prototyped the inference algorithm in MATLAB and improved its performance using pre-recorded and synthetic EEG data (phase 1 in Figure 4.1). Then we developed application prototype in MATLAB and tested it on human subjects (phase 2). To interface with the human, the application (in Figure 6.3) instantiated Stimulus, Presentation, DAQ DevClass to transparently access various HW devices with different connectivity. The action vocabulary and icon set were configured as described in Table 3.1. While the robotic wheelchair was a complex system requiring robotics expertise, the standard interface of the action system allows the application designer to consider the action realization as a black box that faithfully executes inferred actions. With the hardware abstraction, designers could prototype and test the application without dealing with low-level hardware and communication. Finally, our HSyn automatically synthesized the application to embedded implementation in C/C++ deployed on a mobile platform (phase 3).

To realize a wide range of BBCI applications, we developed a low-cost, open embedded BBCI suite (hardware and drivers), including three generations of DAQ, two Stimulus and two visual Presentation subsystems. Table 6.1 lists the hardware in detail. The hardware varies in classification, hardware type (specification) and connectivity. The application transparently accesses
these distributed devices through DevClasses MATLAB interfaces.

### 6.1.3 Hardware Abstraction

Figure 6.3 depicts the application transparently accessing BBCI HW through DevClasses. Each distributed HW system runs the corresponding DevClass-Remote and communicates with the central MATLAB application via proxies. Proxies forward the HW control to DevClass-Remote to be executed and stream sensory data back to the application. To realize the application, we developed a low-cost, open embedded BBCI suite (hardware and drivers), including three generations of DAQ, two Stimulus and two visual Presentation subsystems. Table 6.1 overviews the hardware in details. The hardware varies in classification, hardware type (specification) and connectivity.

#### 6.1.3.1 DAQ

In this application, we used all three DAQ DevTypes to demonstrate the convenience of hardware abstraction. We developed an open embedded DAQ, EEGu2 [25]. EEGu2 integrates a customized cape (PCB) stacking on top of BeagleBone Black (BBB) for 16-channel 24-bit bio-signal acquisition, embedded processing (Cortex-A8), and intelligent power management for mobile operation (up to 12 hours). Two other DAQs are our older generation of the in-house DAQ, PAL-DAQ
CHAPTER 6. DEMO APPLICATIONS

(a) PAL-DAQ (PAL-LED integrated)  (b) EEGu2

Figure 6.4: DAQ: PAL and EEGu2

(8-channel, 16-bit resolution) and a commercial g.USBamp (g.tec Medical Engineering GmbH). While g.USBamp was locally connected to the host PC that ran the application, PAL-DAQ and EEGu2 ran remotely and communicated with the host via TCP. The DAQ DevClass provides hardware and location transparent access to those devices, which simplifies the rapid prototyping and switching between different DAQ types. Later the synthesized embedded application was deployed on EEGu2 (or PAL-DAQ).

6.1.3.2 Stimulus

Stimulus DevClass provides a unified interface to drive frequency or bit-pattern based stimulus. The visual Stimulus is realized with two in-house built devices: PAL-LED (in Figure 6.4a) and BBB-LED (in Figure 6.5a). Both hardware devices support run-time configuration of frequency (0 to 500Hz at 0.1Hz granularity), bit pattern (a sequence of 0s and 1s) and brightness of LED. BBB-LED stimulus subsystem integrates a BBB for processing and a customized cape with Xilinx Spartan3E xc3s500e FPGA. BBB communicates with the FPGA via SPI to drive up to 12 channel LED-array. The FPGA system clock (100MHz) is down-sampled to stimulus frequencies. Then the stimulus signal is modulated with Pulse Width Modulation (PWM) for brightness control. The other stimulus, the PAL-LED drives up to 6 channel LED-array. Both BBB-LED and PAL-LED operate in a distributed manners and communicate with other systems via DDS and TCP respectively.

6.1.3.3 Presentation

The Presentation DevClass displays figures and texts as the visual feedback to users. The Presentation is realized as either a locally connected desktop monitor or a distributed BBB-LCD.
CHAPTER 6. DEMO APPLICATIONS

subsystem shown in Figure 6.5a. BBB-LCD integrates a BBB controlling a 10-inch LCD via HDMI, all protected in a laser cut enclosure. Figure 6.5b depicts the run-time BBB-LCD Presentation displaying 4 action options (up, right, back and left) with 4 flashing LED stimuli mounted at corners. The host application communicates with BBB-LCD via DDS.

6.1.3.4 Robotic Wheelchair

The action system is realized with a semi-autonomous robotic wheelchair (in Figure 6.2b). The system has internal control loops over an number of motors and sensors (lidar, camera, etc.) mounted on the wheelchair. The system infers its pose (position and orientation) from the sensor data and a static map of the environment. Additionally, the wheelchair determines the accessibility and/or feasibility of actions based on the obstacle severity assessment (e.g. from avoiding obstacles to falling down stairs). The wheelchair communicates with other subsystems through DDS. The specific realization of the robotic wheelchair is out of the scope of this work. Details can be found in [56, 58].

Table 6.1 summarizes a variety of hardware and connectivity used in this application. Our framework abstracts the hardware and communication details and provides unified interfaces to distributed devices. As a result, algorithm designers can focus on prototyping applications in MATLAB which is later automatically deployed on embedded platforms through domain-specific synthesis. The hardware abstraction enables a portable application design which can be deployed on various distributed systems. We deploy this application on EEGu2.
6.1.4 Result Analysis

This section discusses the application functional result and the quality of domain-specific synthesis.

6.1.4.1 Functional Result

In our experiments, the EEG evidence was extracted from single channel located on the user’s visual cortex (Oz in the 10-10 system \[59\]). The data was recorded at a 250Hz sample rate and filtered online by a digital FIR bandpass filter (1 Hz - 45 Hz). 4 LED arrays were used for the 4 different system actions. The LEDs were set at 8.1, 9.2, 10.3, and 11.4 Hz. 16 trials per stimuli were collected for system training. Each trial was 4 seconds long with 1 second given to the user to fixate on the target between trials. Figure 6.6 shows an example EEG trace alongside its power spectral density during 8.1 Hz LED stimulation. The wheelchair action execution takes 3 seconds.

During the experiment, users were asked to perform a set trajectory in a house environment (in Figure 6.2a). The results show an average of 91% accuracy in the intent inference process with a 0.9 confidence threshold (maximum posterior probability inference). This threshold was used to reduce user frustration by preventing low confidence decisions. The average time-to-decision was 4 seconds among 9 users (one user was removed due to poor electrode placement and difficulty with instructions). A higher confidence threshold can produce a slightly more accurate inference at the cost of longer time-to-decision.

Figure 6.7 shows the average accuracy and information transfer rate (ITR) as a function
CHAPTER 6. DEMO APPLICATIONS

Figure 6.7: Average accuracy and information transfer rates as a function of trial length for 2 potential classifier (maximum CCA scores and bayesian classifier with CCA features)

of trial length. ITR characters how quickly the uncertainty is removed in the system’s belief of the user’s intent. ITR is defined as:

\[
\text{ITR} = \frac{1}{T} \max_{P_X} I(\hat{X}; X) \\
= \frac{1}{T} \max_{P_X} \sum_{x, \hat{x}} P_{X,\hat{X}}(\hat{x}, x) \log \frac{P_{\hat{X}|X}(\hat{x}|x)}{P_{\hat{X}}(\hat{x})}
\]

(6.1)

(6.2)

where \( T \) is the trial length, and \( I(\hat{X}; X) \) is the mutual information between target and estimated class (stimulus frequency). While the accuracy improves with increasing trial length (more information per trial), the information transfer rate is reduced since it takes a longer time to make a selection (delivering an amount of information). Depending on user performance, trial length could be reduced while still maintaining suitable accuracy. The highest average ITR (35.9 bits/min) is achieved with a trial length of 2.25 seconds. The results also show the advantage of using a Bayesian classifier; building a probabilistic model for each user improves the average ITR compared with maximum CCA.

6.1.4.2 Code Complexity

The footprint (code size) of the prototyped MATLAB application and BAT framework is listed in Table 6.2. Note that the framework footprint excludes DevClass backend and proxies.
which are already implemented in C++. The total footprint of MATLAB program is 1319LOC whereas the synthesized C/C++ program has 5163LOC (3.9x more LOC). Since the framework is provided as a library prior to the application development, the framework LOC should be excluded from the algorithm designer workload. As a result, algorithm designers only need to prototype the algorithm application in 264LOC MATLAB code which is then synthesized into 4171LOC C/C++ embedded implementation (15.8x more LOC). Therefore, algorithm designers are motivated to efficiently prototype assistive applications in high-level MATLAB environment using our framework and take advantage of the HSyn domain-specific synthesis for the automated embedded deployment.

<table>
<thead>
<tr>
<th>Lines of Code (LOC)</th>
<th>MATLAB</th>
<th>C/C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>Algorithm APP</td>
<td>264</td>
<td>4171</td>
</tr>
<tr>
<td>Framework (MATLAB part)</td>
<td>1055</td>
<td>989</td>
</tr>
<tr>
<td>Total</td>
<td>1319</td>
<td>5163</td>
</tr>
</tbody>
</table>

Table 6.2: Application Footprint

6.1.4.3 Productivity Gain

The benefit of domain-specific synthesis can be expressed as productivity gain. To facilitate the analysis, the synthesis time of generating embedded implementation is compared against the time of manual code migration from MATLAB to C/C++. Measuring manual coding effort in a statistically relevant manner is beyond the scope of this dissertation as it requires a sufficiently large group of developers and normalizing their prior knowledge. Instead, the manual development time is estimated based on prior research results in the field. The reported implementation productivity, measured in correct lines of code written per hour, varies: 15LOC/hour [60], 3-30LOC/hour for code refactoring from concurrent to serial programming [61], and 14-20LOC/hour for general software projects [62]. We use 15LOC/hour as an approximate manual development performance in code migration.

Table 6.3 shows that 4171LOC of C/C++ would need to be manually migrated. Compared to a few seconds of synthesis time (Intel I5 1.9GHz and 8GB memory), manual embedded code migration could take over 11 days. Automated embedded deployment is six orders of magnitude faster than the error-prone and costly manual code migration. We acknowledge the difficulty in quantifying productivity gain as the manual development performance varies with development tools,
CHAPTER 6. DEMO APPLICATIONS

<table>
<thead>
<tr>
<th>Properties</th>
<th>Algorithm APP</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOC of MATLAB</td>
<td>264</td>
</tr>
<tr>
<td>LOC of C/C++</td>
<td>4171</td>
</tr>
<tr>
<td>Average Synthesis Time</td>
<td>2.5s</td>
</tr>
<tr>
<td>Estimated Manual Time</td>
<td>11.6days</td>
</tr>
</tbody>
</table>

Table 6.3: Productivity Gain

designer’s expertise, etc. But the orders of magnitude of productivity gained through automatic synthesis will not be significantly affected by ever more precisely measured manual development time.

6.1.4.4 System Utilization

The application is a state machine with conceptually two states: acquisition state and processing states. The processing state comprises intent inference (signal processing), robotic wheelchair control based on inferred intent, Stimulus and Presentation control, etc. Figure 6.8 shows the conceptual execution flow of the application over time. While the DAQ consistently collects EEG samples at 250Hz sampling frequency, the processing of application algorithm is interspersed over time. In Figure 6.8, the application request $N$ samples at $t_0$ and DAQ returns $N$ samples at $t_1$ (DAQ buffer emptied). During the data processing from $t_1$ to $t_2$, DAQ keeps buffering samples. When the application requests next $N$ samples after finishing processing at $t_2$, the application is

![Demo APP Execution Flow](image)

Figure 6.8: Demo APP Execution Flow
CHAPTER 6. DEMO APPLICATIONS

blocked until sufficient samples are returned at \( t_3 = t_1 + N \times T_{\text{sampling}} \). Then application processing resumes at \( t_3 \) and the cycle repeats. The application runs in stable states in a manner that the samples collected in each acquisition period \( (N \times T_{\text{sampling}}) \) are processed during the next acquisition period. In each acquisition period, the processor is busy during the signal processing (overlapped with the acquisition) and idle for the rest of the acquisition period.

To quantify the processing power utilized by the application, Equation 6.3 defines the system utilization as the proportion of the processor busy time (execution time of processing \( N \) samples) over \( N \) sampling periods (actual acquisition time of \( N \) samples).

\[
\text{System Utilization} = \frac{\text{processing time of } N \text{ samples}}{\text{acquisition time of } N \text{ samples}} = \frac{T_{\text{processing } N \text{ samples}}}{N/f_{\text{sampling}}} \quad (6.3)
\]

The system utilization presents the percentage of processing resources used by the application. To minimize the measurement error introduced by the OS scheduling, we approximate the system utilization by measuring the processing time of \( k \times N \) samples at the full processor speed assuming all samples are available at once, while the theoretical acquisition time is calculated as \( k \times N/f_{\text{sampling}} \). The measurement assumes a minimal processing resource consumption from kernel space when the application is fully utilizing the processor.

<table>
<thead>
<tr>
<th>Deployment</th>
<th>MATLAB APP</th>
<th>Synthesized APP</th>
</tr>
</thead>
<tbody>
<tr>
<td>System Utilization</td>
<td>0.34%</td>
<td>2.19%</td>
</tr>
</tbody>
</table>

Table 6.4: System Utilization \((f_{\text{sampling}} = 250Hz)\)

An average system utilization is measured at 250Hz sampling frequency over 3600 seconds. Table 6.4 shows 0.34% system utilization of the MATLAB application running on a computer with I5 processor 1.9GHz (mapped on a single core) and 8GB memory. The synthesized application uses 2.19% processor resource on BBB (Cortex-A8 1GHz and 512MB memory). The low system utilization in both scenarios indicates the potential benefit of low power embedded deployment and the room for computational expansion.
6.2 Brain-Controlled Robotic Arm

Figure 6.9 overviews the brain-controlled robotic arm application (a capstone project). The application adopts the SSVEP-based BCI paradigm to extract human intent from human brain activity and control a robotic arm on human’s behalf. SSVEP uses the fact that focusing on a steadily flashing LED results in visual cortex EEG signals with the dominant frequency of the flickering and its harmonics. By pairing multiple LEDs (with different illumination patterns) to system actions (e.g. moving robotic arm), the user’s intent can be extracted by estimating the frequency of the attended stimuli from EEG. The application comprises four subsystems - EEGu2, StimTron, PstVis, and a robotic arm controlled by BBB. All four subsystems operate in a distributed manner and communicate with each other through DDS over TCP.

Figure 6.10b shows the experiment setup. PstVis presents six action icons to the user to move robotic arm to five locations and pick/drop the item. The robotic system keeps the status of
moving the item. When pick/drop action is selected, the robot arm will drop the item if it currently holds it and vice versa. *StimTron* drives six stimuli through LED arrays flashing at 7.1Hz, 8.2Hz, 9.3Hz, 10.4Hz, 11.5Hz and 12.6Hz respectively. Each LED array is carried by a rail car placed close to the associated action icon. *EEGu2* samples single channel EEG from the user visual cortex at 250Hz sampling frequency. Three passive electrodes are place at forehead for right leg drive, left ear for ground and visual cortex (Oz in the 10-10 system [59]). The user gazes at a stimulus when intending to perform the system action associated with that stimulus.

We used CCA to extract the relevant EEG features. The standard CCA approach consists of computing the maximum Pearson’s correlation between a linear combination of all available channels of EEG data, \( E \in \mathbb{R}^{d \times T_f} \), and a linear combination of template signals \( (Y_i) \) [57]:

\[
\rho_i = \max_{\alpha, \beta} \rho(\alpha E, \beta Y_i)
\]

where \( \rho(\cdot, \cdot) \) refer to Pearson’s correlation function between two vectors of samples, \( f_s \) is the sampling frequency, \( T \) is the trial length in seconds, and \( d \) is the number of EEG electrodes used. The template signal \( Y_i \) is derived from the assumption that EEG can be approximated by a few sinusoids under SSVEP; therefore, for each \( f_i \in F \), the template \( Y_i \) is built as:

\[
Y_i = \begin{bmatrix}
\sin(2\pi f_i t) \\
\cos(2\pi f_i t) \\
\vdots \\
\sin(2\pi H f_i t) \\
\cos(2\pi H f_i t)
\end{bmatrix} \in \mathbb{R}^{2H \times T_f}
\]

for \( t = \left[ \frac{1}{f_s}, \frac{2}{f_s}, \ldots, T \right] \), and where \( H \) is the number of harmonics considered. The action associated with the stimulus of maximum CCA score is inferred as the user desired action.

Using the framework, designers first prototyped CCA algorithm in MATLAB and then tested it on human subjects using the suite. Designers could conveniently program the *EEGu2, StimTron* and *PstVis* from corresponding MATLAB *DevClass* interfaces. Finally, designers used *HSyn* to automatically generate the embedded implementation of the MATLAB application prototype. The generated embedded application was deployed on *EEGu2* for mobile operation. This capstone project validated the usability of *EEGu2, StimTron* and *PstVis* in our *BODE Suite*. The suite (hardware) and the framework (software) enables a rapid development of embedded assistive BBCI application.
6.3 BCI Speller

People with locked-in syndrome (LIS) experience severe communication challenges. Several alternative and augmentative communication schemes have been proposed as potential solutions such as physical switches with row/column scanning, keyboards controlled with eye gaze, among other. As the physical capabilities of affected individuals decrease, such systems become less reliable. Although EEG based computer interfaces have shown promise and great potential to help individuals with LIS, current solutions tend to be expensive and difficult to deploy. We have extended the SSVEP-based Shuffle speller application depicted in Figure 6.11 using our portable BODE Suite. We realize SSVEP using EEGu2 for EEG acquisition and StimTron for visual stimulation. The PstVis is expanded to use a monitor for better visual effect of shuffling characters.

Like in the robot arm application, we used CCA as our feature extraction algorithm. To improve the quality of the classification pipeline, we built a probabilistic model over the EEG feature vector defined as the set of individual CCA coefficients for each stimulation class:

$$x_t = [\rho_1, \ldots, \rho_M]^T$$

The probabilistic models are based on non-parametric kernel density estimation. The conditional probability density functions of EEG features given the class label are:

$$\hat{f}_{EEG}(x|c = c) = \frac{1}{|N_c|} \sum_{k \in N_c} G(x; x_k, \sigma_c^2)$$

where $N_c$ corresponds to the set of indexes for the EEG trials acquired during training. Here, $x_k$ is the EEG feature for the k-th trial from the training set, $\sigma_c$ is the kernel bandwidth parameter that
Figure 6.12: Calibration and testing sessions of spelling application. During calibration, the user is asked to focus on the corresponding target LED array. To spell, letters are assigned to different icon boxes before the LEDs flash. By attending one of the LED stimuli, the probability mass of the chosen letters will increase according to the classifier’s confidence. Once the probability of a letter exceeds a predefined threshold, that letter will be chosen as the user’s intent.

determines how smooth the KDE will be (bias-variance trade-off) – obtained using Silverman’s rule. More details can be found in [48].

Calibration and spelling sessions as shown in Figure 6.12 were conducted with 5 users. EEG was sampled at 250 Hz using a EEGu2 biosignal amplifier and Au passive electrodes. A single electrode was placed at Oz as determined by the International 10/20 System. The EEG signals were digitally bandpass filtered (FIR, linear phase, passband [0.5, 45] Hz). During calibration, the user stared at the flashing target LED stimulus for 3 seconds. 20 trials were collected for each of the 4 stimulus frequencies (8, 9, 11, 13 Hz). The optimal trial time for spelling was chosen according to the maximum information transfer rate (ITR) criterion. ITR is defined as:

\[
ITR = \frac{1}{T} \max_{P_X} I(\hat{X}; X) \\
= \frac{1}{T} \max_{P_X} \sum_{x, \hat{x}} P_{\hat{X}, X}(\hat{x}, x) \log \frac{P_{\hat{X}|X}(\hat{x}|x)}{P_{\hat{X}}(\hat{x})}
\]

where \(T\) is the trial length, and \(I(\hat{X}; X)\) is the mutual information between target and estimated class (LED array frequency). Fig. 6.13 shows the average accuracy as well as information transfer rate for the speller application. Throughout the experiments, although we can reach a maximum accuracy
CHAPTER 6. DEMO APPLICATIONS

![Figure 6.13: Average Accuracy and Information Transfer Rate (ITR)](image)

of 95% with 3 seconds of stimulation, the optimal information transfer rate is achieved with much lower trial length ($\approx 1.3$ seconds); the extra accuracy gained with longer stimulation is not necessary to make quick confident decisions and does not harm user experience. With the coding framework proposed in [48] and a maximum ITR of approximately 43 bits/min, a user could theoretically write 3 to 5 words per minute (assuming English has an entropy of 2.62 bits per letter).

6.4 Summary

This chapter demonstrates the benefit of our BAT framework, rapid design workflow, and BODE Suite with three applications: a brain-controlled wheelchair, a brain-controlled robotic arm and a BCI speller. Using the framework, the designers can rapidly prototype assistive BBCI applications and integrate distributed components from multiple disciplines. Our design workflow enables a rapid vertical integration from algorithm prototyping to embedded deployment with automated domain-specific synthesis. To interface with the human, designer can simply utilize our plug-and-play BODE Suite with convenient user interface from MATLAB. Overall, our framework, rapid design workflow, and suite provide support in designing, developing, and deploying a wide range of assistive BBCI applications.
Chapter 7

Rapid Heterogeneous Prototyping From Simulink

In addition to electrophysiological signals, future assistive BBCI applications can also use video feeds to extract physiological data, for example, capturing users in a camera and inferring user’s physical pose using computer vision. However, the raw video streams cannot be directly integrated for the intent inference, but relevant features need to be extracted beforehand. Such feature extraction is typically very compute intensive, which exceeds the computation capabilities of general-purpose embedded processing. One approach is to employ hardware assisted processing. However developing hardware accelerators for real-time feature extraction is time-consuming and complex.

This chapter explores the domain-specific synthesis of video processing targeting heterogeneous platforms to leverage their low power and high performance benefits. Algorithm designers prototype and fine-tune algorithms using high-level environment, such as Simulink [63]. In Simulink, users benefit from large library resources and an interactive user interface for algorithm design and system modeling. Furthermore, Simulink can synthesize algorithms onto homogeneous platforms (either CPU or FPGA), through Simulink Embedded Coder or HDL Coder [63].

However, a homogeneous architecture may not meet performances or power constraints of demanding applications (e.g. streaming applications). Designer shift attention to heterogeneous Multiprocessor System-On-Chip (MPSoC) which comprises multiple CPUs, memories and hardware accelerators. MPSoCs can improve performance and power efficiency through specialized hardware components (i.e. accelerators). However, current tools do not offer an easy path from Simulink onto
a general heterogeneous architecture.

While additional heterogeneous components increase performance, they widen the gap between Simulink prototyping and heterogeneous implementation. The tremendous effort of manually creating a heterogeneous solution by stitching together homogeneous synthesis results (created in isolation) becomes a bottleneck and lengthens the time-to-market. To bridge this gap, this chapter introduces SimSH which provides an automatic path from a Simulink model to a heterogeneous implementation.

Figure 7.1 overviews SimSH. It takes a platform architecture and a Simulink specification model as input. Using the architectural database, our SimSH generates the necessary interfaces and produces a SW and HW implementation (as binary for processor(s) and bitstream for FPGA(s)). The user analyzes the computation and communication workload of the model and determines processing element (PE) allocation and model-to-PE mapping.

The contributions are the following:

- Introducing a SimSH that provides an automatic path from Simulink to a heterogeneous platform, given PE allocation and mapping. The SimSH empowers algorithm developers rapidly synthesize the application avoiding tedious and error-prone manual implementation efforts.

- The SimSH automatically inserts necessary communication and synchronization across PEs via Communication Refinement. The synthesized layered communication is influenced by the OSI standard [64] to enable reusability and scalability over varying architectures.

- A communication optimization is introduced which detects an underutilized bus, and increases efficiency through pack/unpack to fully utilize the bus.

We demonstrate the benefits using Sobel Edge Detection [65], and map it to a heterogeneous platform of Blackfin DSP and Xilinx FPGA. The results demonstrate significant benefits in terms of (a) rapid realization (within minutes), and (b) increased performance and energy efficiency (both 2.68x over SW implementation).

---

1. This work in this chapter was done in 2014 when Embedded Coder only provides limited synthesis capability targeting heterogeneous platform. Since then, The Mathworks Inc. has released heterogeneous platform support targeting the Zynq platform.
7.1 Related work

Synthesizing Simulink algorithm models to specifications has emerged in recent research. In [66, 67], authors proposed a framework for software code generation from Simulink and validation on MPSoC architecture. In [68], authors generate software for MPSoC from Simulink model and map them onto virtual platform (VP) implemented in FPGA. In [69], authors generate in addition to SW the VP via a combined algorithm and architecture model (CMMA). Unlike [66, 67, 68, 69] which only target multiprocessor architecture and SW generation, we target a general heterogeneous architecture, including CPUs and hardware accelerators (e.g. FPGA). SimSH also explicitly addresses the communication across HW and SW.

The work in [70, 71] convert Simulink models to a System Level Design Language (SLDL) for System Level Design. The work introduces an interesting profiling approach and focuses on design space exploration (DSE). However, it stays at abstract simulation level, unlike our which aims for heterogeneous target execution.

Simulink R2014a [63] also supports concurrent execution code generation. However, it does not specifically address communication optimization. Furthermore, Simulink only targets specific heterogeneous architectures (such as Zynq with single CPU and up to 2 FPGAs), while our work targets a general heterogeneous architecture. Different from the industry approach, SimSH reveals both design methods and usage. It allows users in the academic community to easily expand the tool to support other platforms.

7.2 Hw/Sw CoDesign Framework

The input of the SimSH is a Simulink specification model. Simulink [63] is a Model-Based Design (MBD) tool for system modeling and verification. A Simulink model is described as a set of
CHAPTER 7. RAPID HETEROGENEOUS PROTOTYPING FROM SIMULINK

functional blocks and subsystems, i.e. a grouping of blocks linked by signals.

Figure 7.2 illustrates our SimSH in more detail. It takes a Simulink model as input and guides the user in allocating and mapping blocks based on profiling. Synthesis occurs in 3 phases: Front-end Synthesis, Communication Refinement, and Back-end Synthesis, yielding the SW/HW implementation.

SimSH includes a profiler to investigate the application’s computation and communication workload. It employs Algo2Spec [71] to generate a SLDL specification model (in SpecC), and then profiles the specification using scprof [72]. The profiler reports computation and traffic demands in terms of number of operations, individually for each operation and data type. The profiling exposes computational and communication hot spots of the application.

Figure 7.2: SimSH Flow
GUIDED BY THE PROFILING RESULTS, THE USER MANUALLY ALLOCATES PROCESSING ELEMENTS (PE) AND MAPS THE MODEL ONTO THE PES ACCORDINGLY YIELDING A MAPPED SPECIFICATION MODEL. ALLOCATION IS RECORDED BY ANNOTATING THE SIMULINK MODEL. IN PARTICULAR, THE BLOCK OF A CERTAIN TYPE OF COMPUTATION IS LIKELY MAPPED ONTO THE CORRESPONDING PE THAT IS DESIGNED TO OPTIMIZE THIS TYPE OF COMPUTATION. DETAILS OF PROFILING SIMULINK APPLICATIONS AND THE SYNTHESIS OF SYSTEM SPECIFICATIONS CAN BE FOUND IN [72]. CONVERSELY, THEY ARE OUT OF SCOPE OF THIS ARTICLE, WHICH INSTEAD FOCUSES ON THE SYNTHESIS FRAMEWORK.

THE INPUT SIMULINK APPLICATION MODEL IS SHOWN IN FIGURE 7.2. FOR THE DISCUSSION OF THIS EXAMPLE, ASSUME THAT BLOCKS B AND C ARE COMPUTATIONALLY HEAVY AS REVEALED BY THE PROFILER. USER THEN MAPS THEM ON A HARDWARE WHILE OTHER BLOCKS STAY IN SOFTWARE.

IN THE FRONT-END SYNTHESIS, THE MAPPED SPECIFICATION MODEL IS SPLIT INTO HARDWARE MODELS AND SOFTWARE MODELS AND THEN SYNTHESIZED INTO SOFTWARE IMPLEMENTATION IN C/C++ AND HARDWARE IMPLEMENTATION IN HARDWARE DESCRIPTION LANGUAGE (HDL). IN THIS STEP, THE FUNCTIONALITY OF ALL BLOCKS IN THE MODEL IS SYNTHESIZED FOR DIFFERENT PES WHILE THE COMMUNICATION ACROSS THE PES IS MISSING. TO ADDRESS THAT, WE INSERT THE PROXY IN THE MODEL THAT ENCAPSULATES THE CROSS-PE COMMUNICATION WHICH WILL BE FURTHER REFINED.


IN THE BACK-END SYNTHESIS, SIMSH INTEGRATES THE CROSS-COMPILATION ENVIRONMENT FOR SOFTWARE COMPILATION AND XILINX ISE [73] FOR HIGH-LEVEL SYNTHESIS. IT FINALLY GENERATES SOFTWARE BINARY FOR PROCESSORS AND BITSTREAM FOR FPGAS. THE WORK IN THIS CHAPTER MAKES ASSUMPTIONS AND RESTRICTIONS: (a) THE USER SELECTS ALLOCATION AND MAPPING MANUALLY. (b) IT IS BOUNDED BY SIMULINK EMBEDDED CODER AND HDL CODER RESTRICTIONS AND ONLY SUPPORTS DISCRETE EVENT MODELS USING FIXED STEP SOLVER.

7.2.1 FRONT-END SYNTHESIS

IN THE RESULT OF THE PROFILER-GUIDED ALLOCATION AND MAPPING, A MAPPING ANNOTATED SIMULINK SPECIFICATION MODEL IS THE INPUT OF THE FRONT-END SYNTHESIS, AS SHOWN IN THE UPPER GRAPH IN FIGURE [7.3]
CHAPTER 7. RAPID HETEROGENEOUS PROTOTYPING FROM SIMULINK

While the functional blocks and inter-PE communication is mapped on a PE, the cross-PE communication is implicitly mapped on the shared bus. The front-end synthesis explores the inter-PE communication optimization and inserts *Proxy* for further refinement.

![Diagram](image)

**Figure 7.3: Communication Optimization over Underutilized Bus**

### 7.2.1.1 Communication Optimization

Given a group of blocks mapped on each PE in the mapped model, the traffic between blocks mapped on different PEs is influential to the overall performance. It usually incurs longer latency than inter-PE communication. To achieve efficiency, it is important that user transactions (as generated by the specification) match the underlying interconnect.

SimSH detects the under-utilized bus by comparing cross PE signals’ data width and bus width. Figure 7.3 shows an opportunity for communication optimization. In the original mapped specification model (upper graph), a single transaction (*P-bit width* from *A* to *BC*) is less than the bus width (*NPQ-bit width*), which under-utilizes the bus. The bus utilization can be optimized by concatenating multiple user transactions accordingly. To do this, SimSH inserts in the mapped model a *pack* and *unpack* block at both sides of a cross-PE communication. This bundles multiple user transfers utilizing the bus width (lower graph).

Figure 7.4 visualizes *pack* and *unpack* as parametrizable blocks. On the top, *pack* buffers *NQ* *P-bit* input and concatenates them to a single *NPQ-bit* output. This reduces the data rate by factor *NQ* between input and output. On the bottom, *unpack* slices a *NPQ-bit* input and into *NQ* *P-bit* outputs, increasing the data rate by *NQ*. In result, while the processing blocks (*A*, *BC*, *D*) remain untouched, transfers are bundled and bus utilization is increased.

If the target heterogeneous architecture supports bus burst transfer or DMA, SimSH can more aggressively concatenate transactions at cost of latency. The added blocks (*pack*, *unpack*)
minimally increase computation. The benefits through better utilizing the bus outweigh the minimal computation overhead as the concatenation ratio increases (see Section 7.3).

Overall, communication optimization updates the mapped model with fewer transfers across the blocks mapped on different PEs.

7.2.1.2 Model Splitting and Proxy Generation

SimSH then splits the mapped model into a set of target models, one for each PE. Types include a SW model for a processor (e.g. CPU, DSP) or a HW model (e.g. for FPGA). Each target model only contains the blocks mapped to the particular PE.

Figure 7.5 shows in the top half the mapping annotated model. Blocks A and D are mapped to a DSP (SW) and B, C to FPGA-1 (HW). To illustrate a more general complex example, blocks (X, Y, Z) and the backward communication c3 are included. In result of mapping, the communication across PE boundaries (c1, c2 and c3) needs to be established. For this, Proxy Generation replaces blocks mapped on another PE with a local Proxy. A proxy acts as a placeholder and bundles data input and output of the current PE. Proxy Generation traces the interface types in the Simulink model, and inserts proxies maintaining the interfaces. Figure 7.5 shows the results. Block that are mapped on FPGA-1 (HW), i.e. BC (Unpack NQ:1 and Pack 1:NP) are replaced with a Proxy-BC in the SW model. Proxy-BC on the DSP sends c1 and reads c2 and c3 from HW model. In result, A and D execute as if block B, C were still in SW.

![Diagram](image)

Figure 7.4: *pack* and *unpack* for Concatenating N Transfers
7.2.1.3 Homogeneous Synthesis

SimSH invokes the Simulink Embedded Coder [74] and Simulink HDL Coder [75] to generate target SW and HW implementations. Each of these can optimize internally to generate efficient code. To maximize the potential, we do not synthesize each block individually. Instead, blocks mapped to the same PE are grouped into a super-block and then synthesis is invoked on that super-block. Furthermore, as each inserted Proxy retains the boundary interfaces of the blocks it replaces, it can be scheduled identically to the original specification model. All inserted blocks: pack, unpack and Proxy block are composed of synthesizable blocks (for both SW and HW). Section 7.2.2 discusses Proxy composition and refinement in more detail.

Manually scheduling blocks mapped on the same PE is challenging due to Simulink semantics. To circumvent scheduling ambiguities, SimSH Proxy blocks are Simulink blocks and synthesized by Simulink together with the computation modules. We observe Simulink Embedded Coder generates sequential implementations and Simulink HDL Coder generates pipelined implementations. In addition, by Simulink synthesizing the Proxy, it benefits from all optimizations of the Simulink synthesis.

Overall, front-end synthesis generates computation blocks and the communication within one PE. The inter-PE communication via Proxy needs further refinement as discussed in the next section.
7.2.2 Communication Refinement

SimSH automatically refines communication into a layered implementation following the OSI standard [64] as shown in Figure 7.6: application layer, transport layer, network layer and physical layer. A transaction initiated at the application layer is decomposed into packets at the transport layer, converted into bus transactions in the network layer and finally transferred via the physical layer. The layered design hides the underlying hardware (from the physical layer up), as well as application specifications (from the application layer down). The OSI layered communication implementation allows a wide application of the Proxy principle to a host of heterogeneous architectures. It simplifies expanding the database for new architectures.

At Application Layer, Proxy is a placeholder for blocks mapped to other PEs. It retains identical boundary interfaces of those remote blocks, replicating each port (e.g. in direction, width, data type, and update rate). E.g. SW Proxy-BC in Figure 7.5 implements input port $c_1$, output ports $c_2$ and $c_3$, identical to HW block BC.

At Transport Layer, as shown in Figure 7.6, the SW Proxy-BC instantiates a proxy_recv block for each input and a proxy_send block for each output port. In case of a rate change in the replaced blocks, a Rate Adapter is inserted. In SW, proxy_recv and proxy_send are connected through the Rate Adapter to allow different read and write transactions rates. proxy_recv , proxy_send and Rate Adapter are constructed from a Simulink synthesizable subset to simplify code generation.
Implementing the hardware proxy requires strict timing, as it interfaces with the network layer from the database. To guarantee the timing, one approach is to generate the proxy out of communication primitives. In fact, communication refinement extracts the system composition and connection from the HW target model into a XML file generally following the IP-XACT standard [76]. It captures all relevant port characteristics, which guides the communication refinement to generate a proxy_recv or proxy_send for each port in the HW Proxy. The FIFOs in proxy_recv or proxy_send decouple execution of synthesized application from communication code.

For synchronization across HW and SW, our transport layer uses a buffered asynchronous communication. We use this as we observed that Simulink HDL Coder can synthesize HW blocks into a pipelined design to relax the pressure for the high-level synthesis. We utilize this concept to realize synchronization across each HW/SW boundary by adding an additional cycle delay.

Listing 7.1 shows the pseudo API of read and write transaction in the transport layer. All the transactions at the transport layer and above are hitherto addressed by block ID and port ID from Simulink model and therefore transparent to all underlying heterogeneous architectures.

```
/* Transport Layer of SW Proxy-BC */
proxy_send0(inport_c1){
    c1 = inport_c1;
    send('Proxy-BC', c1);
}
proxy_recv0(outport_c3){
    recv('Proxy-BC', c3);
    outport_c3 = c3;
}
proxy_recv1(outport_c2){
    recv('Proxy-BC', c2);
    outport_c2 = c2;
}
```

Figure 7.7: HW/SW Synchronization
Simulink Embedded Coder synthesizes the SW model into a step function triggered by a periodic timer. After execution of A, Proxy BC issues a write transaction, immediately followed by a read transaction (as governed by the Rate Adapter). It reads the BC result of the previous iteration, and SW continues with D. Hence, HW and SW execution are overlapped. HW starts executing upon availability of the data and produces the output.

As shown in Figure 7.7, BC produces results which are read by D in the next iteration (assuming the same rate for simplicity). This additional iteration delay makes the implementation of HW completely independent of the speed of SW. Therefore, the maximum delay of HW is relaxed to be as large as the complete loop of software execution. In a result, the HW can run multiple orders of magnitude slower than the bus speed, which has a potential for more energy saving. But we don’t explore it in this case because we are targeting on FPGA.

The Network Layer of the Proxy provides addressing and data marshalling as shown in Figure 7.6. We follow a two-layer addressing, similar to Simulink’s identification (block ID and port ID). The network layer maps Simulink’s addressing onto the physical address.

Depicted in Figure 7.8, block ID and port ID follow the HW address prefix from the most significant bit (MSB) to the least significant bit (LSB). The address range for block ID and port ID is dependent on the number of blocks and ports in the model.

Listing 7.2 shows two steps: addressing and marshalling. The SW send and recv function in SW Proxy first convert the block ID and port ID to the physical address and then marshal the transaction payload and eventually call the bus API. In HW model, a 2-level decoder (out of the database) is instantiated to select the HW block and the Proxy FIFO.

The Physical Layer in SW model contains the bus driver from the database. It provides a set of native API for bus transactions called from the network layer. In addition, it also wraps the SW application with some top level architecture specific initialization. In HW model, the physical layer instantiates the bus Interface (IFC) component and the top level FPGA pin mapping [77] as well as the User Constraint File (UCF). The IFC component can directly read data from the bus and interprets a bus writing as signals on the bus lines.

Overall, Table 7.1 summarizes the layering scheme from timing and dependency aspects.
CHAPTER 7. RAPID HETEROGENEOUS PROTOTYPING FROM SIMULINK

Listing 7.2: Proxy API at Network Layer

```c
/* Proxy Network Layer */
send(BlockX, PortY) {
  addr = convert2addr((BlockX, PortY);
  API_BUS_SEND(addr, Port.data);
}
recv(BlockX, PortY) {
  addr = convert2addr(BlockX, PortY);
  API_BUS_RECV(addr, Port.data);
}
```

<table>
<thead>
<tr>
<th>OSI Layer</th>
<th>Timing Accuracy</th>
<th>Application Specific</th>
<th>Platform Specific</th>
</tr>
</thead>
<tbody>
<tr>
<td>App Application</td>
<td>loosely</td>
<td>high</td>
<td>none</td>
</tr>
<tr>
<td>Proxy Transport Network</td>
<td>approximate</td>
<td>medium</td>
<td>medium</td>
</tr>
<tr>
<td>Database Physical</td>
<td>cycle accurate</td>
<td>none</td>
<td>high</td>
</tr>
</tbody>
</table>

Table 7.1: Timing and Dependency of Proxy

The timing accuracy (precision of synthesis) increases along the top-down layering refinement of communication based on OSI model. Besides, the higher OSI layer is more application specific and less platform specific. Hence, the most timing accurate element (bus interface), instantiated from the database, is completely platform specific and independent on applications. Conversely, the application requires full synthesis (with least timing requirements), while proxies are partially parameterized.

In the result of the front-end synthesis and communication refinement, SimSH has generated a complete SW and HW implementation in bare-C and VHDL.

7.2.3 Back-end Synthesis

Back-end synthesis is responsible for synthesizing the C/C++ and HDL code into the appropriate target binaries/bitstream. Based on the selected target architecture, the back-end synthesis integrates cross-compilation environments (e.g. BF527) and Xilinx ISE [73] to automate the SW compilation and HW high level synthesis.

The process of back-end synthesis is automated. SimSH generates Makefiles to automate the SW cross-compilation. For HW back-end synthesis, SimSH generates Xilinx ISE project files and invokes ISE for HW high level synthesis via command line [73].
CHAPTER 7. RAPID HETEROGENEOUS PROTOTYPING FROM SIMULINK

7.3 Experimental Results

To demonstrate the benefits of the framework, we use the Sobel Edge Detect [65]. Sobel Edge Detect detects the edges in an image by comparing each pixel with its neighbors. It computes the gradients of the current pixel via a matrix multiplication of a Sobel operator and the matrix of current neighboring pixels. If the gradient of a pixel is larger than a certain threshold, this pixel is detected as a part of an edge.

![Sobel Edge Detection Algorithm](image)

Figure 7.9 depicts the Simulink model, mainly as a pipeline of Image Load,Serialize, Sobel Edge Detect, Deserializer and Image Print. Image Load simply loads a 320x240 gray image (8 bits/pixel) and Serialize sends each pixel to Sobel Core. Then, Sobel Core outputs the binary decision whether the pixel is part of an edge. Finally, Deserialize assembles the image. For simplicity, we group Image Load,Serialize into a super-block (subsystem) A, Sobel Core into BC, and Deserialize and Image Print into D.

To analyze the computation demands and to guide the mapping process, the application is profiled. In Figure 7.10, the profiling results of Sobel Edge Detect show that Sobel Core occupies 79.1% of the total computation demand. User maps Sobel Core onto the FPGA and the rest on DSP, as shown in blue in Figure 7.9.

This experiment targets a heterogeneous platform of a Blackfin BF527 Digital Signal Processor (DSP) 600MHz [77] and a Xilinx FPGA Spantan3E XC3S500E 100MHz linked by 16-bit External Bus Interface Unit (EBIU) 100MHz on chip.

7.3.1 Application-specific Synthesis Results

In result of the mapping, Sobel Core’s input and output cross PE boundaries (Figure 7.11). For each 8-bit input pixel, the Sobel Core outputs a 1-bit result indicating the edge. Considering a 16-bit EBIU bus, only 1/2 and 1/16 of the bus width is utilized. SimSH detects the underutilized
CHAPTER 7. RAPID HETEROGENEOUS PROTOTYPING FROM SIMULINK

Figure 7.10: Sobel Edge Detect Performance Estimation

To realize the input concatenation, SimSH inserts pack_SW after the Serialize in the SW model (Figure 7.11 bottom). Before sending, the pack_SW marshals two 8-bit pixels into one 16-bit bus transaction. Upon receiving, unpack_HW fires the Sobel edge Detect twice, once with each pixel. This cuts the number of input transactions into half. Similarly, concatenation of 16 Sobel Core output reduces the amount of output transactions 16x.

SimSH splits the resulting model into a HW model and a SW model. In the SW model, it replaces the Sobel Core with emphproxy_sobel as a placeholder which consumes pixels and outputs
decisions mimicking as if Sobel Core would still be in SW. Similarly, in the HW model, SimSH replaces Image Load, Serialize and Deserialize, Image Print with two proxies to receive pixels and send results.

Then, SimSH invokes imulink Embedded Coder [74] and Simulink HDL Coder [75] to generate target SW (C/C++) and HW (HDL) implementations. Here, the generated HDL for the Sobel Core has 13 pipeline stages. Invalid outputs due to pipeline fill are discarded by the HW Proxy.

<table>
<thead>
<tr>
<th>12-bit FPGA Address Prefix</th>
<th>4-bit Block ID</th>
<th>4-bit Port ID</th>
<th>12-bit Free Address Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>19</td>
<td>15</td>
<td>11</td>
</tr>
</tbody>
</table>

Figure 7.12: Proxy Network Layer Addressing

During the communication refinement, the logical addressing based on block ID and port ID is refined to physical addressing of EBIU bus. Figure 7.12 depicts the address allocation following Simulink two-layer addressing: in a 32-bit address, the address range from 19 down to 16 is allocated to block ID and address range from 15 down to 12 is for port ID. The most significant 12 bits are reserved by the memory system for FPGA address range while the least significant 12 bits are still free.

Therefore, the total available 20 bit address range (excluding the reserved 12 bits) supports up to $2^{20}$ combination of block ID and port ID. In this experiment, the allocated 4-bit block ID and 4-bit port ID supports up to 16 blocks and 16 ports, while we only need three blocks and two ports.

To route Bus transaction to the correct Simulink block entity on FPGA, the corresponding two level decoders are generated: two 4-to-16 bit decoder select the hardware block entity and the input/output Proxy FIFO respectively.

The physical layer refinement of the HW Proxy is dependent on the database: the EBIU bus driver in bare-C and Interface (IFC) component in VHDL [77]. We implement the IFC as the EBIU bus driver in FPGA. It reacts to EBIU control line: reading from data line during read transaction and writing to EBIU control line and data line during write transaction. Furthermore, the physical layer also encapsulates the top level FPGA pin mapping as well as the User Constraint File (UCF).

### 7.3.2 Evaluation

To illustrate the benefits of the HW/SW co-design and the communication optimization, we compare five different implementations (varying mapping and optimization level) as illustrated in Figure 7.13. **Pure SW Solution** maps the whole specification model on BF527 DSP. **HW-SW**
Co-Design (no opt) maps Sobel Core on the Xilinx Spartan3E FPGA and the remaining blocks on BF527 DSP. HW-SW input pack/unpack optimizes the input communication of HW module by inserting pack/unpack on the path from SW to HW. HW-SW output pack/unpack optimizes the output communication of the Sobel Core. HW-SW input/output pack/unpack optimizes both input and output of the Sobel Core.

The baseline solution is the pure SW solution, mapping the whole Simulink model on DSP. It results in the longest execution time about 45.6 Mcycles. HW-SW [no-opt] maps the most computationally intensive Sobel Core to FPGA, with the rest running on DSP, all in a pipelined fashion. The total execution time drops to 20.3Mcycles, yielding a 2.25x speed up. However, HW-SW [no-opt] includes a communication overhead across DSP and FPGA of 22.2% of the total execution time.

Optimizing the path from SW to HW, input pack/unpack solution reduces traffic overhead slightly but yields an longer total execution time than HW-SW [no-opt]. The overhead of executing pack (in SW) outweighs the communication performance gain which is small due to the low input concatenation ratio of 2 : 1 (pack 2 pixels). Conversely, when only optimizing the path from HW to SW in [output pack/unpack] solution, the overall performance increases, as the output concatenation ratio 16 : 1 is much higher than input concentration.
Finally, optimizing both paths, SW to HW and HW to SW, achieves a 2.68x speed up against the pure software solution. The total communication time (0.58Mcycles) of HW-SW \textit{input/output pack/unpack} decreases 10 fold compared to unoptimized HW-SW \texttt{[no-opt]} solution (5.8Mcycles). Meanwhile, the communication time (0.58Mcycles) with 3.4% of the total execution time is no longer a significant delay contributor.

To assess power efficiency, we measure board-level power of our platform. It remains fairly constant at around 680mW regardless of load and FPGA usage. As the DSP runs at the fixed frequency, this indicates that the FPGA (whose load is changed) is a minor contributor towards the total dynamic power. Nonetheless, HW/SW Co-design and further communication optimization shorten total execution time in the heterogeneous execution. Hence, the energy efficiency increases linearly with performance speedup. Our optimized HW/SW solution is 2.68x more energy efficient.

Table 7.2: FPGA Utilization of HW/SW Optimized Solution

<table>
<thead>
<tr>
<th>Slice</th>
<th>Total</th>
<th>Application</th>
<th>Proxy+Glue Pack+Unpack</th>
<th>Database</th>
</tr>
</thead>
<tbody>
<tr>
<td>Usage (out of 9312)</td>
<td>547</td>
<td>170</td>
<td>177</td>
<td>200</td>
</tr>
<tr>
<td>Utilization</td>
<td>5.8%</td>
<td>1.8%</td>
<td>1.9%</td>
<td>2.1%</td>
</tr>
</tbody>
</table>

In the HW/SW optimized solution, the FPGA utilization of the Spartan 3E is only 5.874% as shown in Table 7.2. The generated \textit{Proxy, pack, unpack} and other glue logic (bus IF) occupy 32%. The application specific \textit{Soble Core} is small in this example. This indicates significant room for other implementations, e.g. duplicating \textit{Sobel Core} on FPGA. However, this algorithm optimization is out of the scope of the SimSH. On the other hand, the DSP is fully utilized at nearly 100% for all five implementations due to the overlapped HW/SW execution.

### 7.4 Summary

This chapter introduces a Simulink-based SimSH to bridge the gap between the algorithm design in Simulink and its implementation on a heterogeneous platform. Given an allocation and a mapping decision, our SimSH automatically synthesizes the Simulink model onto the heterogeneous target and refines the synchronization and communication across processing elements. Furthermore, the SimSH optimizes communication by detecting an underutilized bus and concatenating transactions.
CHAPTER 7. RAPID HETEROGENEOUS PROTOTYPING FROM SIMULINK

accordingly. In the result, it allows the developer to focus on the algorithm exploration and tuning and rapidly prototype it on a heterogeneous target platform.

We have demonstrated the benefits using Sobel Edge Detection [65], and targeted a heterogeneous architecture with a Blackfin processor and Spartan3E FPGA. Our proposed SimSH achieves up to a 2.68x speedup and energy efficiency with communication optimization against a pure software solution. In future work, we will investigate into automatic mapping decisions for a given platform.
Chapter 8

Conclusion

BBCI-based assistive applications extract human intent through BBCI and augment users’ interaction with the physical world. These applications have a significant societal impact that can help, for example, amputees to regain mobility, locked-in people speak to their beloved ones, and many others. However, designing assistive applications poses many challenges due to requiring concerted multidisciplinary efforts, using specialized devices (with various types and connectivity) to interface with the human, and deploying the high-level algorithm design to embedded systems. A holistic approach is needed to provide a common view across multiple disciplines, a workflow for rapid development, and necessary human interface devices for BBCI.

This dissertation addresses the challenges in designing, developing, and deploying assistive BBCI applications in three aspects: horizontal integration to bridge disciplines, vertical integration from prototyping to deployment, and human interface devices for BBCI. Without losing generality, we formalize assistive BBCI applications and introduce a holistic BAT framework to capture common functionality encapsulated in generic modules, each defining the responsibility of one discipline and its interface across discipline boundaries. Our framework provides a holistic view of the applications and integrates efforts from multiple disciplines to build an application. Utilizing the flexible publish-subscribe communication across pre-defined module interfaces, our framework allows a modular, distributed composition. The modular design and externalized configurations enables our framework to support a wide range of applications.

To accelerate application development, our framework adopts a MBD approach from computation modeling, to rapid prototyping with simplified hardware access (via hardware abstraction), finally to automated embedded deployment (via HSyn’s domain-specific synthesis). HSyn demonstrates several magnitudes of productivity gain compared to manual embedded deployment. In
result, algorithm designers can simply focus on the algorithm design and rapidly develop and deploy assistive applications without requiring hardware-specific and embedded systems knowledge.

Utilizing off-the-shelf human interface devices sets high barriers of entry due to high device cost and integration effort. We introduce an affordable, open device (BODE) suite tailored for BBCI and supported by hardware abstraction for rapid prototyping and development. We identify BBCI utilizing three categories of devices interfacing with the human and our suite provides these three interface devices, including physiological data acquisition (EEGu2), simulation (StimTron) and presentation (PstVis). Our suite is fully integrated into our framework enabling a distributed composition of devices, convenient programming interfaces in MATLAB, and automated deployment on any device in our suite. EEGu2 provides 16-channel, 24-bit resolution acquisition with 25dB SNR and battery-powered embedded processing for mobile operation. StimTron generates 12-channel visual and tactile stimulation with configurable bit pattern and frequency from 0.1Hz to 409.6Hz at 0.1Hz resolution. PstVis presents visual feedback and additionally incorporates a rail car system for flexible stimulus placement.

We demonstrate the flexibility and usability of our BAT framework and BODE Suite with three assistive BBCI applications: a brain-controlled wheelchair, a brain-controlled robotic arm, and a BCI speller. Using our rapid design workflow, designers first prototype applications in MATLAB and then test them on human subjects utilizing our plug-and-play human interface devices. Finally HSyn provides an automatic path from MATLAB application prototype to embedded implementation deployed on our EEGu2. HSyn dramatically reduces the development time of embedded deployment by six orders of magnitude as compared to manual conversion. Our flexible framework (software), affordable suite (hardware), and rapid design workflow constitute a comprehensive solution to tackle the design, development, and deployment challenges in assistive BBCI applications and empower researchers to easily convert their creativity into products.
Chapter 9

List of Publications

Here, we provide the list of publications related to this dissertation. Among the publication, three conference papers ([25][26][78]) and two journal articles have been already submitted.


- S. Feng, F. Quivira, G. Schirner, “BODE: An Embedded Open Device Suite for Body/Brain-Computer Interface”. (Journal under submission)


BIBLIOGRAPHY


BIBLIOGRAPHY


BIBLIOGRAPHY


BIBLIOGRAPHY


