An Embedded Device for Real-Time Noninvasive Intracranial Pressure Estimation

by

Jonathan Martin Matthews

B.S., Massachusetts Institute of Technology (2015)

Submitted to the
Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2016

Author .....
Department of Electrical Engineering and Computer Science
May 20, 2016

Certified by .....
Thomas Heldt
Hermann L.F. von Helmholtz Career Development Professor
Thesis Supervisor

Accepted by .....
Christopher J. Terman
Chairman, Master of Engineering Thesis Committee

An Embedded Device for Real-Time Noninvasive Intracranial Pressure Estimation

by

Jonathan Martin Matthews

Submitted to the Department of Electrical Engineering and Computer Science
on May 20, 2016, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Monitoring of intracranial pressure (ICP) is key in many neurological conditions for diagnosis and guiding therapy. Current monitoring methods are highly invasive, limiting their use to the most critically ill patients. Based on a previously developed approach to noninvasive ICP estimation from cerebral blood flow velocity (CBFV) and arterial blood pressure (ABP) waveforms, I have implemented the algorithm on an embedded device (LPC4337 microcontroller) that can produce real-time estimates of ICP from noninvasively-obtained ABP and CBFV measurements. I have also fabricated a medical device prototype complete with peripheral interfaces for ABP and CBFV monitoring hardware and display and recording functionality for clinical use and post-acquisition analysis. The current device produces a mean estimate of ICP once per minute and can perform the necessary computations in 410 ms, on average. Real-time estimates of noninvasive ICP differed from the original batch-mode MATLAB implementation of the algorithm by 0.34 mmHg (RMSE). The contributions of this thesis take a step toward the goal of real-time noninvasive ICP estimation in a variety of clinical settings.

Thesis Supervisor: Thomas Heldt
Title: Hermann L.F. von Helmholtz Career Development Professor

Acknowledgments

I would like to thank Professor Thomas Heldt and Dr. Andrea Fanelli for supervising the thesis and for the wise advice, indispensable mentorship, and wonderful friendship along the way.

I would also like to thank Gavin Darcey, David Lewis, and John Sweeney at the MIT Cypress Engineering Design Studio for their guidance and time in assisting me with the fabrication of the device.

This work was supported in part through a research grant from Maxim Integrated Products to the MIT Medical Electronic Device Realization Center.

Contents

List of Figures

Chapter 1

Introduction

A 52 year-old woman is eating dinner with her family when she suddenly experiences a severe headache and collapses to the ground. Her husband calls for an ambulance, and the woman is taken to the hospital where, after CT imaging, she is diagnosed with a subarachnoid hemorrhage. As she undergoes surgery and is treated with medication, one of the most important physiological parameters for her physicians to monitor is her intracranial pressure. If the pressure rises too high, her brain can become dangerously deprived of blood, putting her at further risk in her already-critical condition.

In this chapter, you will learn about intracranial pressure, its importance in neurocritical care, and, most importantly, the techniques by which it can be measured. You will also learn about the advantages and disadvantages of each measurement method that have collectively motivated the design and implementation of the medical device prototype that is the focus of this thesis. By the end of this section, you will also understand how the thesis fits into a larger effort in the field of noninvasive intracranial pressure measurement to provide safer and more robust healthcare for patients like the woman above.

1.1 An Intracranial Pressure Primer

1.1.1 Physiology

Intracranial pressure (ICP) is the hydrostatic pressure of cerebrospinal fluid (CSF). CSF surrounds the brain and fills the cerebral ventricles, mechanically and chemically supporting the tissue (Figure 1-1). It is produced by epithelial cells in the choroid plexus and slowly circulates through the ventricles and subarachnoid space, eventually draining into the venous bloodstream. Because the CSF resides, in part, in a fluid column in the spinal cord, ICP is affected considerably by gravitational effects. In a healthy person, normal ICP measured in the ventricles can vary between 7 mmHg and 15 mmHg when lying down and is around -10 mmHg when standing [2].

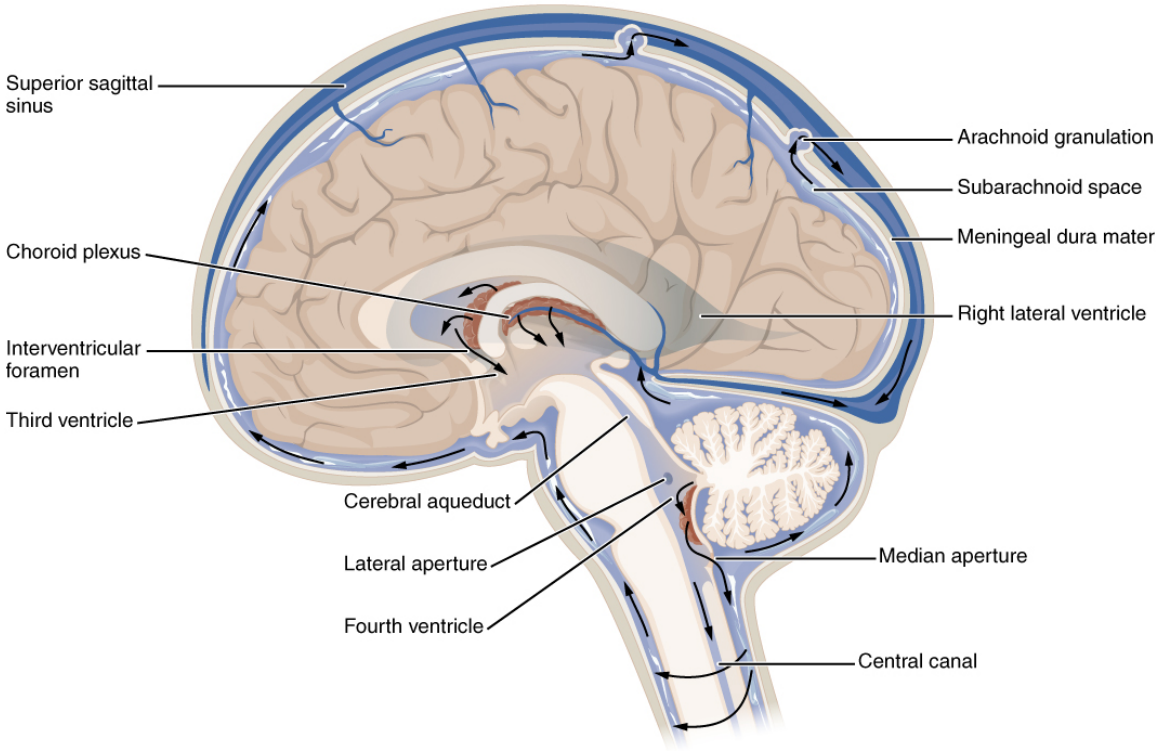

Figure 1-1: A schematic depicting the production, circulation, and reabsorption of cerebrospinal fluid (CSF). Figure taken from [6].

The Monro-Kellie hypothesis states that the volumes of CSF, brain tissue, and intracranial blood are constant [5]. Because of the rigidity of the skull, an increase in the volume of one compartment has to come at the expense of the volume occupied

by the other two and possibly an increase in ICP. The change in volume of CSF is determined by the difference between the rate of CSF production and drainage into the blood, so that in steady-state these rates are equal to one another and the volume of CSF is maintained constant. If the rate of production is increased, the volume of CSF will increase, in turn increasing ICP, and vice-versa for a decrease in the rate of production. As another consequence of the Monro-Kellie hypothesis, ICP depends upon the volume of the brain tissue and cerebral vasculature: as blood is pumped through the vessels in the brain, the compliant cerebrovasculature pulses with the heartbeat, rhythmically compressing the CSF and introducing a pulsatile component into the ICP waveform.

1.1.2 Pathophysiology

A significant increase in ICP can have a large impact on the adequate delivery of blood to brain tissue. Cerebral perfusion pressure (CPP) is the pressure gradient that drives the flow of blood through the capillaries of the brain and is defined as the difference between the upstream pressure and effective downstream pressure. The upstream pressure corresponds to the mean arterial blood pressure (MAP) while the effective downstream pressure is determined by ICP, as it is greater than the venous blood pressure at the level of the brain (this is the flow limitation effect of a Starling resistor) [15]. If compensatory mechanisms are impaired, ICP increases and CPP decreases, limiting the flow of blood to brain tissue. The decrease in perfusion is normally met with autoregulatory vasodilation, which has the result of increasing cerebral blood volume, further increasing ICP. The reduction of flow of blood to brain tissue deprives the tissue of oxygen and nutrients and often leads to ischemia and death if untreated.

1.2 Measurement Methods

The impact that ICP has on cerebrovascular health makes it imperative that physicians have a reliable method for measuring ICP in patients with conditions such as

brain tumor, cerebral edema, intracranial hemorrhage, and hydrocephalus among others. It is also often important to measure a patient's ICP during their recovery from a neurosurgical procedure, as an elevated ICP can indicate bleeding or infection. Here, I will discuss the methods used to measure or estimate ICP, as well as the advantages and disadvantages of each technology.

1.2.1 Pressure-sensor catheter

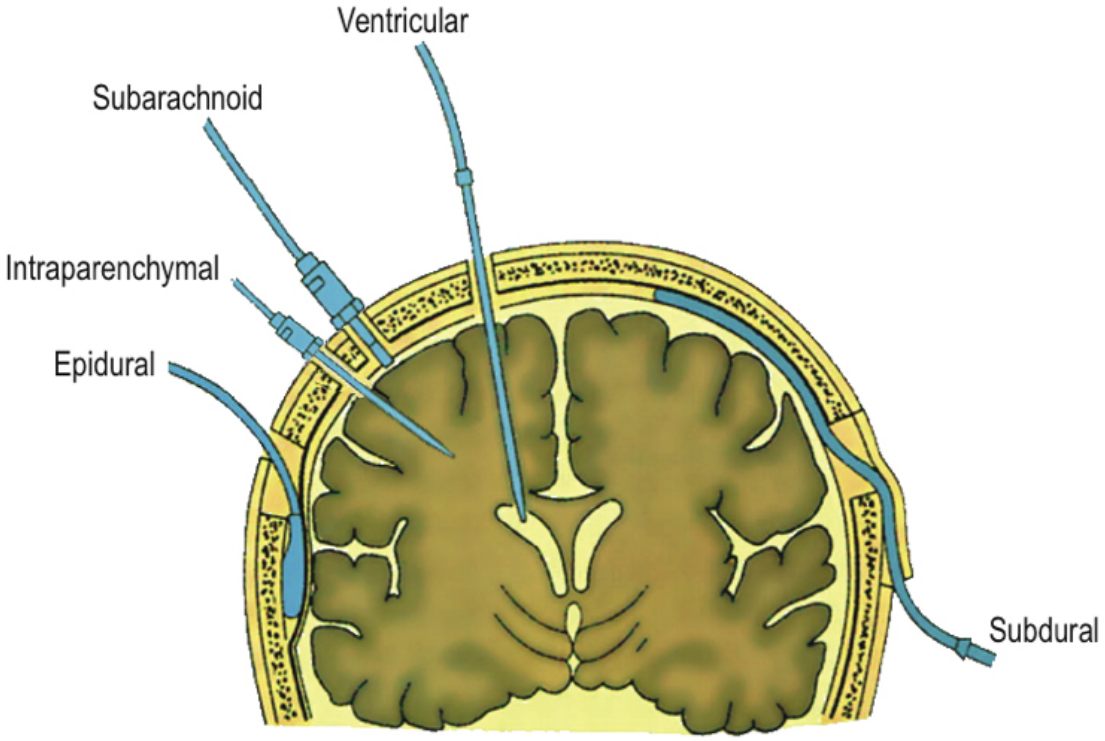

Figure 1-2: A fluid-filled catheter can be inserted into one of several spaces in the cranial vault to invasively estimate ICP. Figure taken from [6].

The clinical gold standard for ICP measurement is the direct surgical insertion of a fluid-filled catheter into the ventricular space (Figure 1-2). A hole is drilled through the skull and the intraventricular catheter threaded (blindly) through the brain tissue into one of the lateral ventricles where it makes direct contact with the CSF. These catheters are also often placed to drain excessive CSF and relieve pressure when ICP is elevated. Alternatively, a pressure-sensitive probe can be placed in the

brain parenchymal space and measure the pressure of the brain tissue. Though both the intraventricular and parenchymal pressure probes provide measurements of ICP, the insertion procedures are highly invasive, carrying with them the risk of bleeding, infection, and damage to vital brain structures. These risks have motivated the development of noninvasive methods for estimating ICP (nICP).

1.2.2 Noninvasive ICP estimation

nICP estimation focuses on using physiological signals that can be obtained in a minimally invasive fashion, such as arterial blood pressure (ABP), cerebral blood flow velocity (CBFV), or intraocular pressure (IOP), as surrogates for ICP [9].

ABP can be measured either through the insertion of a pressure-sensing fluid catheter into an artery or through the vascular unloading technique [14]. In the vascular unloading technique, an air cuff is secured around one of the fingers and the volume of the finger is measured with a plethysmograph. As blood pulsates through the finger, a computer system measures changes in the plethysmographic data and applies a pressure through the air cuff to keep the volume of the finger constant. The externally applied pressure corresponds to ABP.

CBFV is measured through transcranial Doppler ultrasound (TCD), which measures the Doppler shift in ultrasound waves applied through the skull as they reflect off of the red blood cells (mostly) moving through arteries.

IOP is measured through applanation tonometry whereby an external pressure is applied to the eye until the corneal surface is flattened; this "applanation" pressure corresponds to IOP.

The set of approaches towards noninvasive ICP estimation that use these physiological signals can be divided into two overall classes. Methods in the first class are learning-based, identifying patterns between ICP and physiological variables that can easily be measured noninvasively. These methods require a large training dataset of invasive ICP measurements of patients with a wide range of pathologies and physiologies to be useful. The second class of algorithms involves applying physiologically-motivated models to mechanistically relate noninvasively-measured signals to ICP.

One such method is based on a model of blood flow through the central retinal artery whereby the intracranial segment of the artery is compressed by ICP and the extracranial segment is compressed by IOP [10]. ICP can then be estimated by applying an external pressure to the eye until features of the CBFV waveform are the same in both segments.

The device described in this thesis is based on the Kashif algorithm for noninvasive ICP estimation, which is of the physiologically-motivated class of approaches [4]. The Kashif algorithm is based on a lumped-parameter model of blood flow through the cerebrovascular system and uses ABP and CBFV measurements at the middle cerebral artery (MCA) to estimate the mean ICP. A detailed description of the algorithm will be presented in Chapter 2. The algorithm has previously been implemented in MATLAB on a desktop computer. This runs in "batch-mode", acting on patient data after they have been recorded. This is nonideal in the clinical scenarios described in Section 1.1.2 in which transient ICP changes are immediately relevant and clinically actionable.

1.3 A Real-Time Embedded Device Implementation

This thesis presents a microcontroller-based medical device prototype (Figure 1-3) that applies the Kashif algorithm to ABP and CBFV signals acquired in real-time to produce an estimate of mean ICP once every 60 seconds and can perform the necessary computations in less than one second. The estimates from the device differed from the batch-mode MATLAB implementation with a RMSE of 0.34 mmHg. The device is housed in a sleek enclosure that can be connected to the analog signal outputs from standard ABP and CBFV measurement hardware. ICP estimates, as well as the ABP and CBFV signals, are displayed on an LCD screen on the front panel and are also saved to a SecureDigital (SD) card within the device. The device is powered by a USB port that also provides serial streaming of ICP estimates to a computer. The embedded device offers the advantages of low cost and size as well as product modularity over a general-purpose computer, and is a key next step towards an ideal

clinical application.

1.4 Thesis Structure

The next chapter describes a lumped-parameter model of the cerebrospinal and cerebrovascular fluid spaces and the steps of the Kashif algorithm to estimate ICP. The structure of the batch-mode implementation and its strengths and weaknesses are also discussed.

Chapter 3 focuses on the detailed implementation and testing of the device, and also enumerates and explains the software and hardware tools that were used. I also mention the challenges that were encountered during the device development.

Chapter 4 presents the performance of the device including comparison to the batch-mode implementation and invasive ICP, the speed and memory usage of the algorithm, and the physical specifications of the final assembled prototype device. I also discuss the advantages and limitations of the device based on these results.

In Chapter 5, I review the contributions of the thesis to the field of noninvasive ICP estimation and briefly discuss the possible next steps for the project.

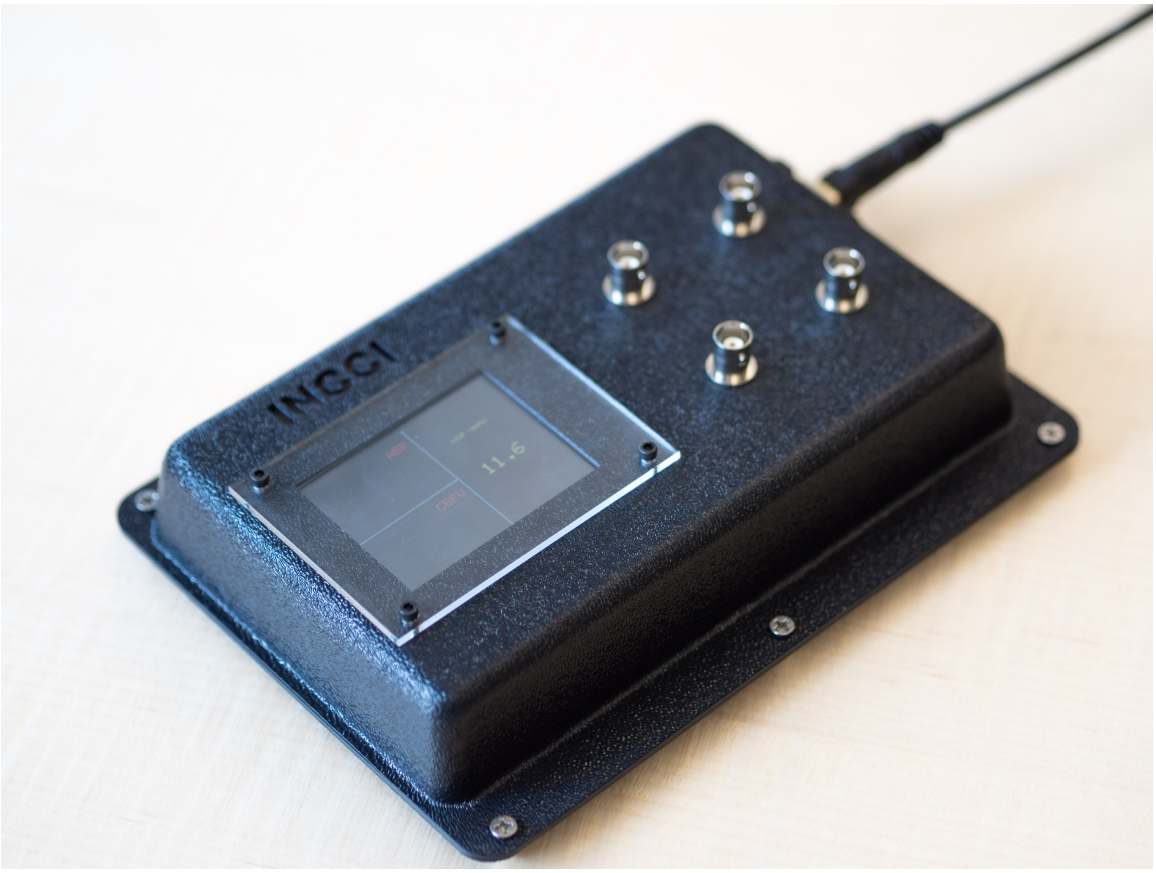

Figure 1-3: The assembled medical device prototype for real-time noninvasive ICP estimation.

Chapter 2

Background

An accurate noninvasive method for estimating intracranial pressure (ICP) could have a tremendous impact on lowering the risk of neurocritical monitoring and could benefit a much larger patient population as a diagnostic tool for hemorrhagic or ischemic stroke, traumatic brain injury, hydrocephalus, and brain tumors [4]. The Kashif noninvasive ICP estimation algorithm is motivated by a model of cerebrovascular physiology that relates ICP to arterial blood pressure (ABP) and cerebral blood flow velocity (CBFV). In this chapter, you will learn in detail about this physiological model, the steps of the Kashif algorithm, and the batch-mode implementation of the algorithm in MATLAB that is the foundation upon which this thesis builds.

2.1 A Dynamic Model of Cerebrovascular Space

The cranial vault contains the vasculature, brain tissue, and cerebrospinal fluid (CSF). As mentioned in the last chapter, these compartments are dynamically coupled due to the Monro-Kellie doctrine as well as the flow of fluid from one compartment to another. Ursino and Lodi [15] and others have developed models to simulate this coupling and the associated dynamics. Kashif reduced such bigger models to a minimal physiological model and used it to estimate ICP [4].

2.1.1 The Ursino-Lodi model

The Ursino-Lodi model [15] of the cerebrovascular space focuses on the relationship between the compartments of the cerebral vasculature, brain tissue, and the CSF space in a lumped-parameter framework (Figure 2-1). Blood enters the skull with a certain cerebral blood flow (CBF, or q) and pressure (P_a). The arteries and arterioles are represented with both capacitive (C_a) and resistive (R_a) components, corresponding to the compliance of and pressure drop across these vessels. Arterial capacitance and resistance vary with time due to changes in vascular smooth muscle tone. These changes are caused by autoregulatory and autonomic control responses to maintain an appropriate perfusion of the brain with blood.

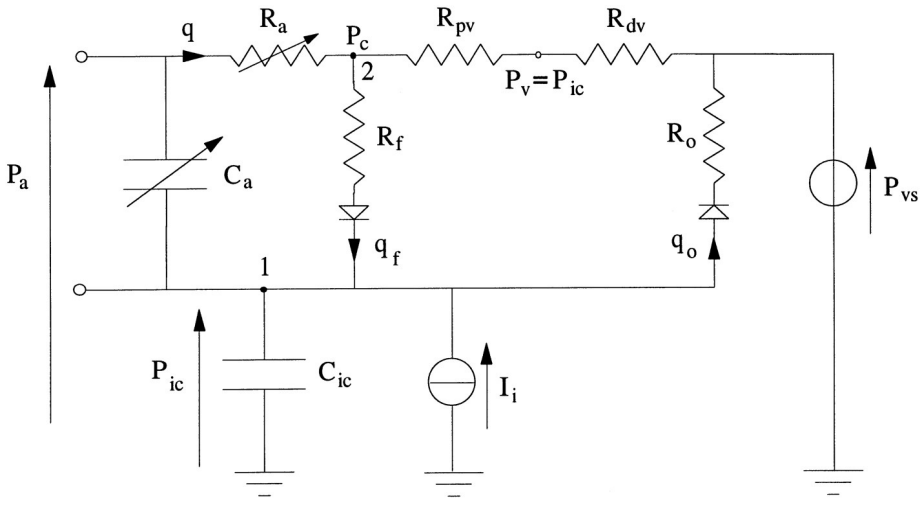

Figure 2-1: The Ursino-Lodi lumped-parameter model of blood and cerebrospinal fluid flow through the cerebrovasculature. Figure adapted from [15].

In the capillary beds, the CBF encounters a further pressure drop (across R_{pv}) to cerebral venous pressure (P_v). As well, CSF filtrate is secreted from the capillaries, namely in the choroid plexus in the four cerebral ventricles, into the CSF space. This secretion is physiologically driven by active transport mechanisms, but the model represents it through passive, hydrostatically-driven mechanisms. CSF fills the ventricles, cisterns, subarachnoid space, and sulci of the brain and exerts a hydrostatic pressure on the surrounding tissues that is ICP (or P_{ic}). The brain tissue also has an elastic property, represented in the lumped-parameter model as a capacitor (C_{ic}).

CSF is steadily reabsorbed into the circulatory system through cerebral veins at venous sinus pressure (P_{vs}) where it joins the venous blood flow exiting the cerebrovasculature. Importantly, ICP is greater than P_v. This has the consequence of collapsing the cerebral veins and producing the Starling resistor effect of flow limitation through a collapsible tube. The effective downstream pressure of the capillaries is then determined by ICP instead of systemic venous pressure; this phenomenon is represented in the model by replacing P_v with ICP (P_{ic}).

2.1.2 The Kashif model

Kashif *et al.* developed a simplified version of the Ursino-Lodi model for the purposes of estimating ICP from (1) ABP at the level of the cerebral vasculature and (2) CBF through the middle cerebral artery (MCA) [4]. They determined that the processes of CSF production from the choroid plexus and reabsorption into the venous system are much slower than that of blood flow through the cerebrovasculature. The CSF production and reabsorption flows (q_f and q_o) in the Ursino-Lodi model can therefore be neglected over the course of tens to hundreds of cardiac cycles and the corresponding pathways can be treated as open circuits over these time scales (Figure 2-2).

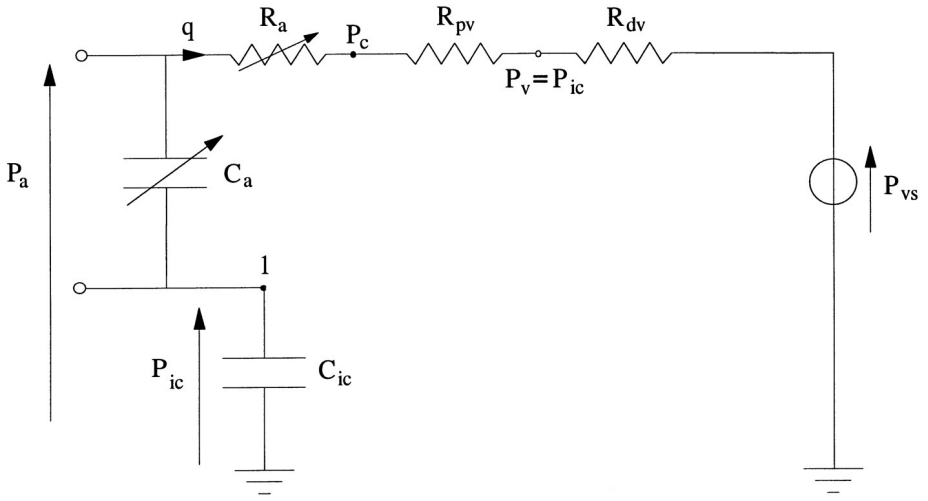

Figure 2-2: The slow rate of CSF production and reabsorption are fully neglected in the Kashif model. Figure adapted from [15].

This simplification leaves two parallel pathways through the model, one purely

resistive and one purely capacitive, which can be further simplified to a single RC circuit (Figure 2-3): a source voltage $p_a(t)$ and current $q(t)$ represents the instantaneous ABP and CBF waveforms, respectively, through the MCA; a single resistor (R) represents the overall effective resistance of the cerebral vasculature; and a single capacitor (C) represents the effective compliance of both the vasculature and the brain tissue. ICP consequently represents the pressure surrounding the vasculature and brain tissue downstream of the microcirculatory resistance (R).

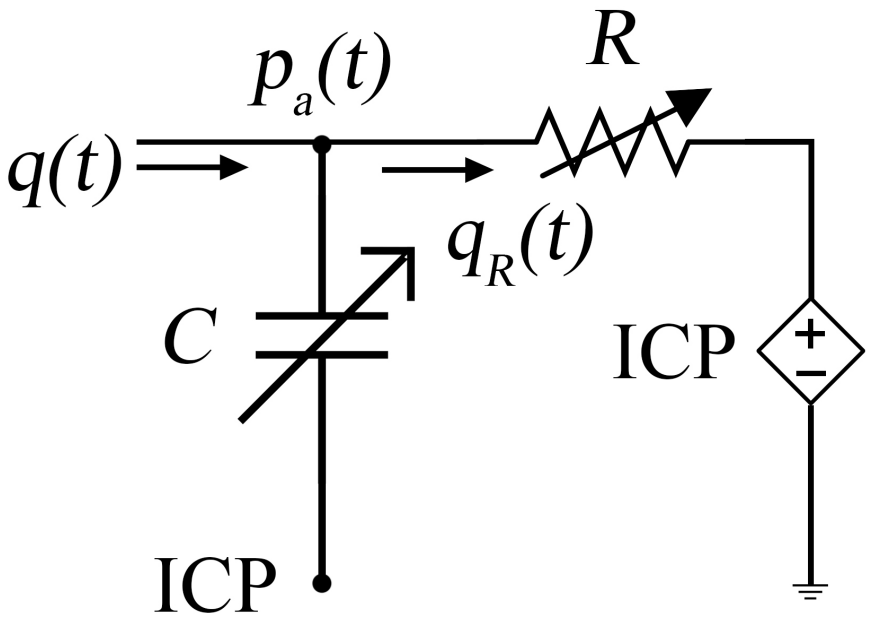

Figure 2-3: The fully simplified RC electrical circuit representation of the cerebrospinal fluid space. Figure adapted from [4].

Kirchoff's current law for the model gives an expression linking the measurements $q(t)$ and $p_a(t)$ to $ICP(t)$ and the model parameters $R(t)$ and $C(t)$ as

$$q(t) = \frac{p_a(t) - ICP(t)}{R(t)} + C(t)\frac{d(p_a(t) - ICP(t))}{dt} \tag{2.1}$$

Here, R and C are time-varying parameters that depend on a number of physiological factors, including autoregulation and autonomic activity. The Kashif approach assumes that they are constant over a small enough estimation window. ICP is also not a constant value, as explained briefly in the first chapter; the Monro-Kellie

doctrine implies that the morphology of the ICP waveform will be similar to that of the ABP waveform. Additionally, ICP can vary because of mismatches in CSF production and reabsorption and because of other space-filling lesions.

The Kashif approach simplifies Equation 2.1 by considering only the mean ICP, which is assumed to be constant at the mean value of $ICP(t)$ within a sufficiently small estimation window. Equation 2.1 can then be reduced to

$$q(t) = \frac{p_a(t) - ICP}{R} + C\frac{dp_a(t)}{dt} \tag{2.2}$$

which is the basis for the Kashif algorithm.

2.2 Estimating Intracranial Pressure

2.2.1 The estimation algorithm

The algorithm is broken down into three steps: (1) estimating C, (2) estimating R, and (3) using the estimated C and R to solve for ICP. The first step takes advantage of the rapid upstroke in ABP during systole. At this point, the capacitive current, $C\frac{dp_a(t)}{dt}$, is large compared to the resistive current. The flow in the model can be approximated to be determined mainly by the compliance pathway as

$$q(t_{upstroke}) \approx C\frac{dp_a(t)}{dt}|_{t=t_{upstroke}} \tag{2.3}$$

Equation 2.3 can be integrated over the range $[t_b, t_e]$, the time window bordering the systolic ABP upstroke

$$[p_a(t_e) - p_a(t_b)]C = \int_{t_b}^{t_e} q(t)\mathrm{d}t \tag{2.4}$$

to solve for C. As $q(t)$ and $p_a(t)$ are susceptible to noise, artifact, and interference, this calculation is carried out over many heartbeats in an estimation window typically of size 60 beats, to produce a system of equations each of the form of Equation 2.4. The least-squares-error solution $\hat{C}$ can be obtained as an estimate of C from this

system of equations.

The second step begins by considering the flow through the resistive branch of the model,

$$\hat{q}_R(t) = \frac{p_a(t) - ICP}{R} = q(t) - \hat{C}\frac{dp_a(t)}{dt} \tag{2.5}$$

which can now be calculated using our estimate $\hat{C}$ and finite differencing of $p_a(t)$. Based again on the assumption that mean ICP is constant throughout the estimation window, R can be computed by considering two time instants t_1 and t_2 so that

$$ICP = p_a(t_1) - R\hat{q}_R(t_1) \tag{2.6}$$

$$ICP = p_a(t_2) - R\hat{q}_R(t_2) \tag{2.7}$$

and thus

$$[\hat{q}_R(t_2) - \hat{q}_R(t_1)]R = p_a(t_2) - p_a(t_1) \tag{2.8}$$

This can be applied for any time points t_1 and t_2; however, they are selected to minimize the noise sensitivity of $\hat{q}_R(t)$ and hence maximize $\hat{q}_R(t_2) - \hat{q}_R(t_1)$ by locating the minimum and maximum points in the ABP waveform (maximizing the right-hand side). Similar to the first step, Equation 2.8 is computed for every heartbeat in the estimation window to produce a system of equations that can be used to produce a least-squares-error solution $\hat{R}$ as an estimate of R.

Lastly, the mean $p_a(t)$ and $\hat{q}_R(t)$ over the estimation window can be used together with the estimate $\hat{R}$ to produce an estimate of mean ICP as

$$\widehat{ICP} = \overline{p_a(t)} - \hat{R}\overline{\hat{q}_R(t)} \tag{2.9}$$

Through these three steps, the Kashif algorithm estimates mean ICP from ABP and CBF waveforms over estimation windows sufficiently small to assume constant mean ICP and constant effective lumped resistance and compliance of the cerebrovas-

culature, brain tissue, and CSF space.

2.2.2 Obtaining measurements of ABP and CBF

Strictly following the parameters of the model, the inputs to the Kashif algorithm are ABP at the level of the MCA, and CBF into the MCA. These measurements are impossible to obtain directly through noninvasive techniques, especially ABP at the MCA. Instead, the more easily obtainable ABP at a peripheral artery and cerebral blood flow velocity (CBFV) through the MCA are used as proxy measurements for the true ABP and CBF at the MCA [4].

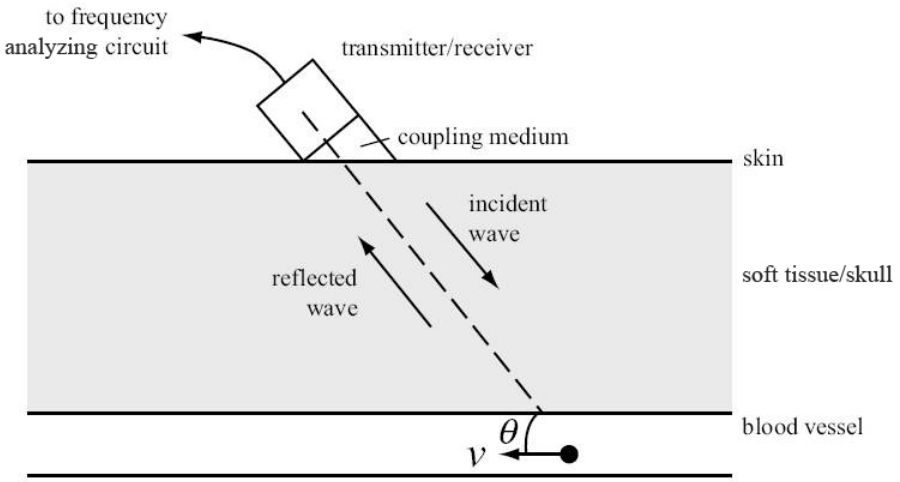

Figure 2-4: Doppler ultrasound technology is used to measure the velocity of blood flow through vessels in the body. Figure taken from [11].

Transcranial doppler ultrasound (TCD) is a widely used technology for measuring CBFV (Figure 2-4). An ultrasound probe emits sound waves through the skull and into the brain tissue and vasculature that reflect off of tissue interfaces with an amplitude proportional to the difference in the acoustic impedances of the tissues. These echoes are detected by a transducer that is next to or the same as the emitter. The time at which the echo is detected after emission will therefore correspond to the depth of the tissue interface. Static tissue boundaries produce reflections but no shifts in the ultrasound frequency. A moving target, however, will produce a shift in frequency according to the Doppler principle. The magnitude of the received signal

will depend on scattering and absorption, but on the angle of insonation — ideally this will be along the velocity vector of the material (i.e. $\theta = 0$).

The left and right MCAs run normal to the sagittal plane in the brain and so these vessels can be probed by placing a TCD transducer near the temples. By locating an ideal insonation angle and measuring the amplitude, travel time, and Doppler frequency shift of the ultrasound echoes, the MCA can be located and the CBFV through the vessel can be determined.

The relationship between the blood pressure waveform at the radial artery and at the MCA has not been completely characterized. We expect there to be morphological differences between the peripheral and MCA ABP waveforms. These differences have not been experimentally established, but would be expected based on the compliance and resistance of the vasculature and reflections of pressure waves in vessels of different sizes and muscle tone states as well as reflections off of vessels in branching sites. Furthermore, the peripheral and MCA pressure waves differ by the relative time of arrival of the pressure waveform. The Kashif algorithm does not compensate for morphological differences, but does attempt to (approximately) correct for the timing differences [4] (see Section 2.2.4).

Peripheral ABP can be measured by inserting a catheter into the radial artery and connecting it to a pressure transducer. This method is the gold standard for continuous blood pressure measurement. A sphygmomanometer is commonly used to measure systolic and diastolic blood pressure, but it does not produce an actual waveform of the blood pressure signal. Arterial catheterization allows for the accurate determination of the blood pressure waveform, however it brings along the risk of bleeding, infection, and possible nerve damage [3]. Alternatively, the arterial blood pressure waveform can be collected noninvasively using the volume-clamp method of a Finapres-type device, as described earlier.

The TCD probe and Nexfin vascular unloading technology together afford continuous and noninvasive measurements of CBFV and peripheral ABP that are related to the CBF and ABP at the MCA for use in the Kashif ICP estimation algorithm.

2.2.3 Locating heartbeat onset times

Equation 2.4 and Equation 2.8 in the Kashif algorithm need to be carried out on a per-heartbeat basis for each heartbeat in the estimation window. In order to do so, the ABP and CBFV waveforms need to be segmented into individual heartbeats. The open-source PhysioNet `wabp` algorithm, originally developed by Zong *et al.*, detects the onset times of ABP pulses based on the first derivative of the continuous ABP signal [17].

In `wabp`, the ABP signal is low-pass filtered with a cut-off frequency of 16 Hz to suppress high-frequency noise and then passed through a slope sum function (SSF) — a sliding windowed sum of the positive first derivatives of the ABP signal — to selectively enhance the systolic upstroke. A threshold is applied across the SSF signal to locate the pulse windows; onset points are then identified within each window as the crossing point of 1.0% of the maximum of the SSF signal within that window. The threshold is updated to be 60% of the maximum of the SSF signal in the previous pulse to adapt to changes in mean arterial pressure and pulse pressure over time. The detected onset points are passed to the Kashif algorithm and used to segment the ABP waveform into windows corresponding to each heartbeat.

2.2.4 ABP and CBFV signal alignment

Another key necessity for running the Kashif algorithm is that the ABP and CBFV signals must be properly aligned. Because CBFV is measured at the MCA and ABP is measured at the radial artery, the delay between the two waveforms due to the difference in the distance that the blood pressure wave travels must be taken into account. There are also time delays that are intrinsically part of the signal acquisition pathways for both CBFV and ABP. The Nexfin and TCD monitor carry out distinct measurement algorithms that differ in efficiency and run on processors with different speeds. These factors produce a considerable time-shift between the two signals that can be observed in the raw output from the Nexfin and TCD.

The signal realignment process to compensate for these time shifts focuses on the

alignment of the heartbeat onsets times in the ABP and CBFV waveforms. First, the `wabp` algorithm is run on the CBFV signal to locate the heartbeat onset times, which are then individually paired to their closest onset times in the unaligned ABP signal. Then, the heartbeat window durations are computed for both signals according to the onset times that have been paired. Letting $D_{ABP}[n]$ and $D_{CBFV}[n]$ represent the duration of heartbeat n in the ABP and CBFV signal, respectively, and with N representing the total number of heartbeat durations, the quality of the alignment is calculated as

$$S(D_{ABP,k}, D_{CBFV}) := \frac{1}{N} \sum_{i=0}^{N-1} |D_{ABP}[i-k] - D_{CBFV}[i]| \tag{2.10}$$

The ideal alignment (or lag) K is determined by considering a set of alignments obtained by shifting ABP relative to CBFV by k heartbeats and computing

$$K = \arg\min_{k}(S(D_{ABP,k}, D_{CBFV})), k \in [-5, 5] \tag{2.11}$$

This corresponds to the alignment that produces the least average absolute difference in heartbeat durations between the ABP and CBFV signals.

Applying this alignment procedure will compensate for any delay between the ABP and CBFV signals and make the heartbeat onset times in both waveforms precisely line up with one another. This alignment may not be reflective of the true ABP and CBFV dynamics at the MCA, however, as the compliance of the vasculature will cause the CBFV upstroke to occur slightly after the ABP upstroke. Preserving this delay during alignment is an ongoing area of research in the group. For the purposes of the implementation discussed in this thesis, the above heartbeat alignment procedure is used.

2.3 The importance of implementation

Kashif *et al.* implemented the algorithm in a MATLAB script that runs in batch mode on pre-recorded ABP and CBFV data. The script assumes that the recorded

signals are already aligned and that the ABP corresponds to the pressure at the level of the MCA. This script is a batch-mode implementation, which produces estimates of mean ICP from data that has been recorded at a hospital and transferred onto a computer. The batch-mode implementation has been tested by referencing ICP estimates to ICP measurements recorded invasively with a parenchymal probe and demonstrated a mean error of 1.6 mmHg, which is smaller than the normal 2-3 mmHg fluctuations in ICP due to respiration and cerebrovascular volume pulsatility [5].

While the batch-mode MATLAB implementation has performed well, it is nonideal in clinical scenarios where sudden changes in ICP are immediately relevant. It is useful for analysis of ICP trends, but it cannot be used for monitoring. Invasive ICP probes are used for monitoring critically ill patients and so a substitute noninvasive solution must be built with a real-time application in mind. The remainder of this thesis describes the design and implementation of a new incarnation of the Kashif algorithm on a microprocessor platform that can produce a mean estimate of ICP once per minute from data acquired in real-time, taking a step towards the goal of real-time noninvasive continuous ICP estimation in a variety of clinical settings.

Chapter 3

Methods

As detailed in the last chapter, the Kashif algorithm produces noninvasive estimates of intracranial pressure (ICP) by:

1. reading arterial blood pressure (ABP) and cerebral blood flow velocity (CBFV) waveforms,

2. time-aligning the two signals to account for physiological delays and processing times during data acquisition,

3. detecting heartbeat onset times with the `wabp` algorithm, and

4. computing mean ICP through estimating R and C for each data window.

In this chapter, you will learn about how the MATLAB batch-mode implementation of this procedure was ported to a microprocessor platform and modified to produce estimates once per minute from data acquired from a patient in real-time. You will also learn about the design and fabrication of a complete medical device prototype that interfaces with a TCD monitor and Nexfin ABP measurement system and reports both the current estimated ICP on an LCD display for monitoring and records the estimates on an SD card for later analysis.

3.1 A Strategy for Real-Time Estimation

The batch-mode implementation slides a 60-heartbeat-wide estimation window across prerecorded ABP and CBFV waveforms, producing an ICP estimate for each window along the way. To do so, `wabp` is run on the fully-collected waveforms, taking advantage of the entire signal to adapt the heartbeat onset threshold. In a real-time implementation, ABP and CBFV data need to be collected and processed on-the-fly and shifted into the estimation window. Once the window is filled, `wabp` and the Kashif algorithm can be run to produce a single ICP estimate. As the algorithm runs, the next window will be filled with new samples.

The size of the estimation window is determined by a balancing of several factors — namely the reliability of heartbeat onset detection, available memory on the microprocessor, sampling frequency, the minimum estimation frequency that will be useful to a physician, and a tradeoff between fast response time (small windows) and noise averaging (large windows). The shorter the estimation window, the less reliable the detected heartbeat onset times will be due to the adaptive nature of `wabp`. Likewise, the shorter the estimation window, the higher the variability of the ICP estimates will be as less data is inherently averaged in the least-squares estimation steps. On the other hand, a shorter window will take up less memory on the microprocessor and produce ICP estimates more frequently for clinical use. A lower sampling frequency may not fully capture the smaller features of the waveforms and may even introduce aliasing at frequencies lower than 30 Hz. At the same time, a higher sampling frequency will take up more microprocessor memory. Physician collaborators at Boston Medical Center suggested that ICP estimates presented at least once per minute — and hence an estimation window of about 60 seconds will suffice for neurocritical care monitoring.

3.2 Core Tools

Before detailing the implementation, I will briefly enumerate and describe the set of core tools that were used in developing the embedded device prototype. This digression is to define many of the terms and provide foundational knowledge that will be used in the later sections.

3.2.1 Hardware

The NXP LPC4337 microcontroller unit (MCU) [12] was selected for its large SRAM capacity, processor speed, and available peripherals (Figure 3-1) and also due to compatibility with a related project in the Microsystems Technology Laboratories at MIT involving novel TCD technology [7]. The MCU is based on a 200 MHz ARM Cortex-M4/M0 dual-core processor unit and features 136 kB of on-chip static random-access memory (SRAM), a high-speed universal serial bus (USB) controller, a SD card interface, 164 general-purpose input/output (GPIO) pins, and two 10-bit analog-to-digital converters (ADCs) that can sample at up to 400 kHz. The 136 kB of RAM is spread out across five distinct locations — one 40 kB block, two 32 kB blocks, and two 16 kB blocks [12]. There is also 1 MB of flash memory built into the MCU for possible longer-term storage, however the removable SD card is preferentially used for this purpose so that saved data is easily accessible for analysis on a computer.

The NGX LPC4330-Xplorer board (Figure 3-2) is a breakout board for the LPC4337 MCU with a large set of peripherals, importantly featuring an on-board microSD card slot, USB port, and 10-pin Joint Test Action Group (JTAG) header for debugging [13]. All of the unused digital and analog inputs and outputs are broken out to headers, which allowed for rapid development and prototyping. The LPC-Link 2 board was used to program and debug the MCU.

3.2.2 Development suite

LPCXpresso is the integrated development environment (IDE) standard for LPC MCUs [8]. The IDE is Eclipse-based, but is used for C/C++ application and library

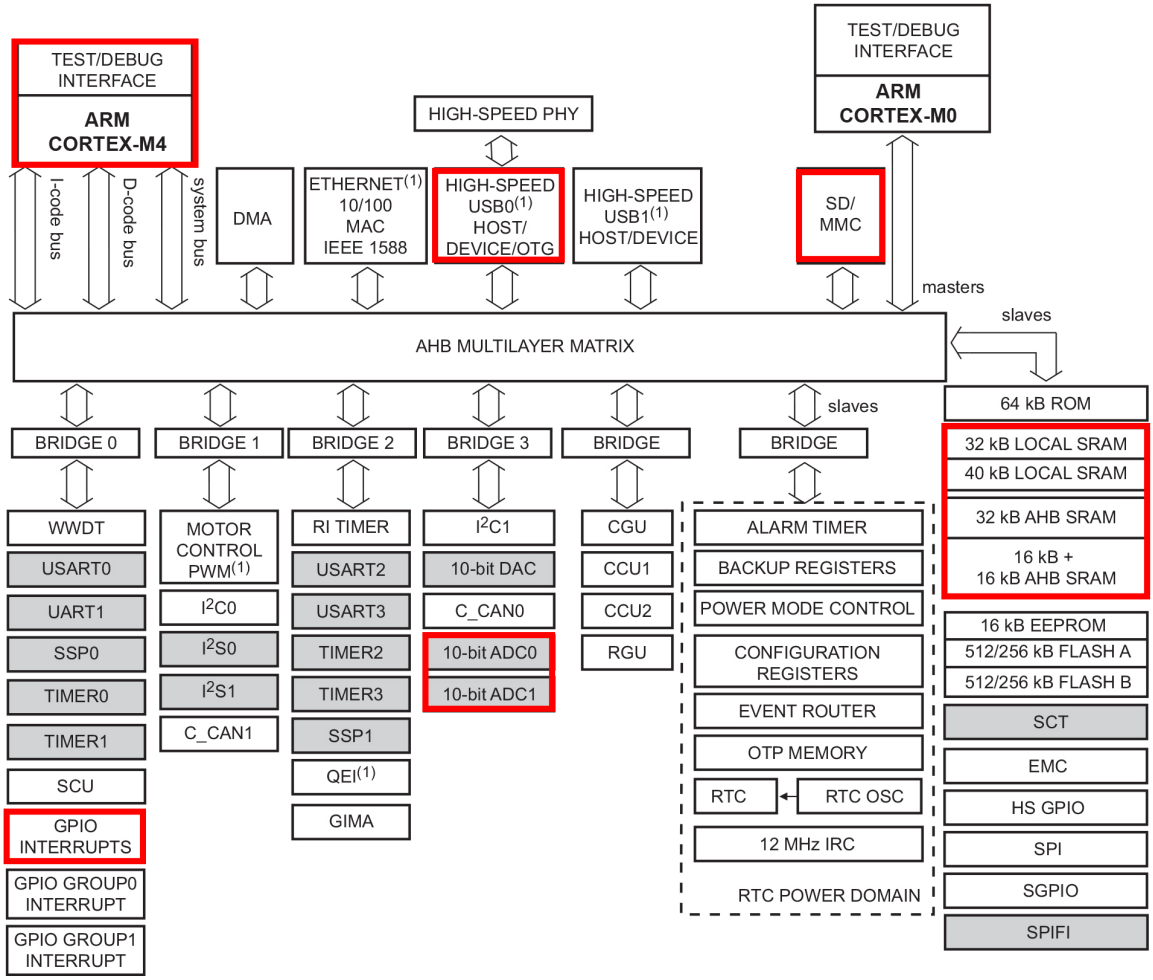

Figure 3-1: A block diagram of the available peripherals on the NXP LPC4337 microcontroller. Important blocks are highlighted, including the built-in SRAM, ADCs, GPIOs, SD controller, USB controller, and debug interface. Figure adapted from [12].

development. As well, the platform can be used for free for code sizes up to 256 kB, which is more than enough for programming the 136 kB of SRAM on the LPC4337. C++ was selected as the implementation language for the device, primarily for the organizational advantages of object-oriented programing that offer easier modular development and testing than C.

LPCXpresso is packaged with LPCOpen, an open-source platform that provides simple access to the basic libraries for controlling the LPC4337, such as setting the system control unit (SCU) and GPIO registers to select digital pin functionality. LPCOpen is written in C, however the libraries can still be leveraged in a C++

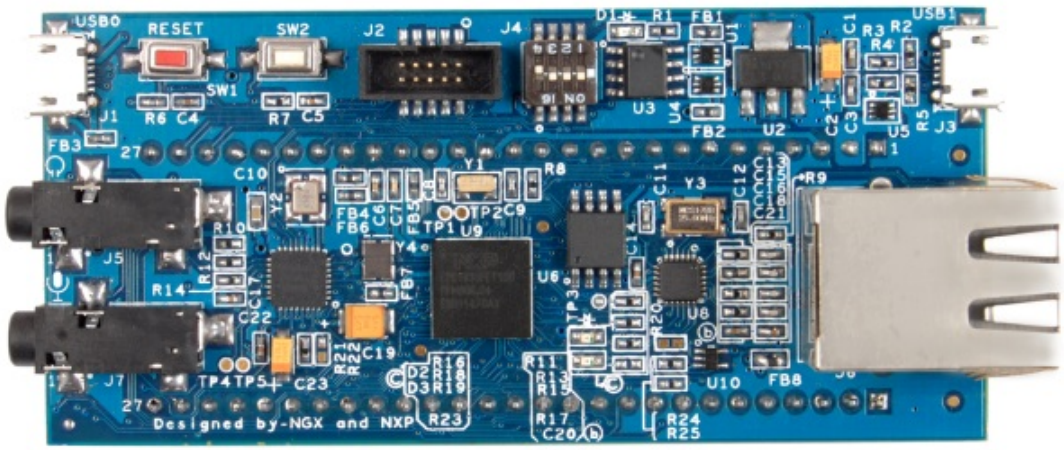

Figure 3-2: The NGX LPC4330-Xplorer board was used for the medical device prototype. Figure taken from [13].

application. LPCOpen also comes with several example projects, including an SD and USB controller as well as a demonstration of ADC polling and interrupt handling. These example projects and libraries were used extensively in the development of the embedded device.

3.2.3 Porting and profiling

MATLAB scripts and functions, namely the batch-mode implementation of the nICP algorithm, can be easily ported to a C++ program through the MATLAB Coder tool. MATLAB Coder supports nearly all of the MATLAB language and produces efficient and readable C and C++ source code. However, the generated code is not optimized for a microcontroller implementation where memory is scarce. A major step in the device implementation process was to reduce the memory usage of the ported algorithm. To assess memory usage, the Valgrind memory management suite

was used [16]. In particular, Valgrind's `massif` heap profiler tool can measure peak usage of dynamically allocated memory, which was helpful in minimizing the ported algorithm to fit into MCU memory.

3.3 Implementation

The process of implementing the ICP estimation algorithm on the embedded device consisted of several major subtasks, which I will enumerate and describe here.

3.3.1 Building the data pipeline

To become familiar with developing on the LPC4337 and the core tools for the implementation process, a basic data pipeline was built that consisted of loading ABP data onto the board through USB streaming, running the `wabp` algorithm, and then writing the detected heartbeat onset times to an SD card.

A Python script was written to encode prerecorded ABP data (collected at 50 Hz) from Addenbrooke's Hospital in Cambridge University and to send 60 seconds of the waveform over a USB virtual communication port (VCOM). The LPCOpen USB library was imported for loading the data onto the board. As each sample is received, it is shifted into a buffer in RAM; once the necessary 3,000 samples are collected, the buffer is passed to `wabp`.

MATLAB Coder was used to port a MATLAB implementation of `wabp` to C++. The generated code required 1.8 MB of RAM, far exceeding the 136 kB available on the LPC4337, and so needed to be optimized to fit into MCU memory — this optimization process will be discussed in Section 3.3.3. The heartbeat onset times were then written to a microSD card using the LPCOpen SD/MMC library.

3.3.2 Porting to C++

Three scripts were ported from MATLAB to C++ — `wabp`, the Kashif algorithm (`getICP`), and a preprocessing script (`preprocess`) that upsampled the Adden-

brooke's Hospital data from 50 Hz to 125 Hz. The first step in porting the MATLAB scripts to C++ was to clean up the existing codebase by removing unused variables and code blocks and commenting unclear steps. As well, the files were modified to be in a functional format instead of a script format and reorganized so that all arrays were preallocated, as MATLAB Coder does not handle dynamic growth of arrays (which was used throughout the original `getICP` script).

MATLAB Coder was then used to generate C++ functions for each script using default settings except for specifying "NXP Cortex-M4" as the selected hardware. This selection helps MATLAB Coder to define the data type sizes and endian-ness among other parameters that will be appropriate for the target system. To ensure that the generated code performed correctly, test rigs were built to compare the results obtained through the C++ implementation to those obtained through the original MATLAB implementations. These test rigs were also useful for ensuring correctness while the code was iteratively modified to minimize memory usage. All of the generated functions were tested on a PC using GCC for building and GDB for debugging before being implemented on the MCU.

The signal alignment procedure discussed in the last chapter had been implemented in MATLAB, though the script was simple enough to be manually ported to C++. The C++ implementation is listed in Appendix A.

Lastly, the data pipeline was modified to stream 60 seconds of CBFV data across USB VCOM in addition to ABP and record the estimated ICP for that window on an SD card.

3.3.3 Memory optimization

The C++ implementations of `wabp` and `getICP` that were generated from MATLAB required 1.8 MB and 348 kB of RAM, respectively, which far exceeds the available memory on the MCU; `preprocess` required only 92 kB of RAM. As mentioned, the LPC4337 features a total of 136 kB of SRAM. There are many inefficiencies in memory usage that were manually reduced to fit the code into the board's memory.

The greatest memory inefficiency in the generated implementations is that mem-

ory is rarely reused. Large allocations are requested to store temporary workspace data even when memory that was allocated earlier is available and no longer being used. Remedying this issue was straightforward, though tedious, by manually changing the C++ files to reuse workspace memory whenever possible. As well, the real-time approach allows for many array sizes to be reduced; for example, it is now assumed in both `wabp` and `getICP` that the maximum number of heartbeats possible in the 60-second window is, somewhat arbitrarily, 250 beats. Any further heartbeats in excess of 250 will be ignored, though an estimate will still be produced for that window. Additional optimizations included using single-precision floating-point variables instead of double-precision variables and performing operations in-place at the cost of speed whenever possible.

Another difficult memory-related challenge arises when considering that the MCU's SRAM is spread out over 5 separate memory locations in block sizes of 40 kB, 32 kB, 32 kB, 16 kB, and 16kB. Some of the allocated arrays are over 30 kB large but cannot span more than one RAM block. Because of this, the arrangement and packing of allocations becomes especially important. Embedded devices by design do not run an operating system and so there is no built-in virtual memory management or caching that would make this task simple. A lightweight software interface was developed to provide virtual contiguous memory access to the separate memory blocks, allowing for tight packing of allocated space and for the large arrays to be accessed even when spanning two or even three RAM blocks. The core of the tool is the `ContinuousMemory` object, which maintains base references and sizes of all five RAM blocks on the MCU. When memory is "allocated", a `CMemPointer` object is created to point to a base location in the `ContinuousMemory` virtual memory space. When a `CMemPointer` is dereferenced, the `ContinuousMemory` object determines the physical location of the virtual address in the SRAM blocks and provides access to that location. The C++ implementation of this lightweight interface is included in Appendix B.

3.3.4 Peripherals for real-time estimation

The data pipeline described in Section 3.3.1 uses USB VCOM to send a single 60-second window of prerecorded data to the MCU. Real-time estimation involves continuous acquisition of data on-the-fly and so this pipeline needed to be modified slightly. While signal alignment and the `wabp` and `getICP` functions are being executed on an estimation window, the next window needs to be collected. Peripheral interrupt handlers can be enabled on the LPC4337 that run a specified function whenever a peripheral triggers a particular event. Custom code was appended to the end of the built-in USB interrupt handler to process bytes as they are received, package them into ABP and CBFV samples, and shift the samples into the estimation window buffer.

As well, the real-time device must interface with the TCD monitor and Nexfin to acquire ABP and CBFV signals as they are measured. The TCD monitor and Nexfin both have analog output ports that report CBFV and ABP by supplying voltages at 9 mV per cm/sec and 10 mV per mmHg, respectively. The interface pipeline was designed in two iterations: (1) streaming real-time data over USB VCOM through a computer interfacing with the TCD monitor and Nexfin analog outputs and (2) interfacing with the TCD monitor and Nexfin analog signal outputs directly and digitizing them with the ADCs built into the LPC4337.

For iteration (1), a computer was connected to a National Instruments Data Acquisition System (NI-DAQ) and loaded with a LabVIEW virtual instrument (VI) that sampled from the NI-DAQ ADCs connected to the TCD monitor and Nexfin at 125 Hz (Figure 3-3). The VI handles digitizing the analog signal outputs from both devices, scaling and plotting the signals as ABP and CBFV on-screen, recording the signals to a file, and streaming the signals to the LPC4337 at 125 Hz. As well, the VI reads in the ICP estimates sent back from the device and displays them on-screen. For iteration (2), the built-in ADCs were set up to operate in interrupt mode at the minimum sampling rate of 73 kHz. A counter in the ADC interrupt handler lowered this effective sampling rate to append samples to the estimation window buffer at

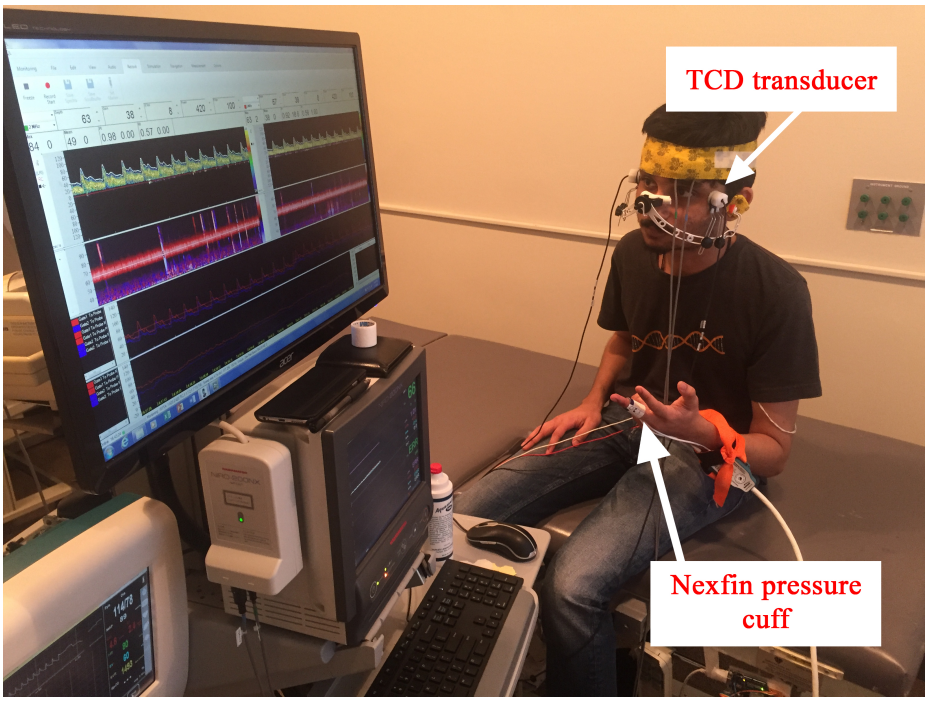

Figure 3-3: A data acquisition cart was built to acquire ABP and CBFV signals from the Nexfin and TCD probe for the first trial.

125 Hz. In both iterations, the `preprocess` function was not needed as data was supplied at the `wabp`-native 125 Hz.

ICP estimates also need to be presented to the physician as they are produced. An Adafruit 2.8" thin-film-transistor liquid-crystal display (TFT LCD) was used to display the most recent ICP estimate [1]. The LCD has its own built-in controller and memory and can be programmed through 4-pin Serial Peripheral Interface (SPI). To lighten the load on the LPC4337, an Arduino Nano was used to control the LCD. Eleven GPIO pins on the LPC4337 were configured to output mode and connected to digital inputs on the Arduino. Estimated ICP is encoded to an 11-bit number and presented on the GPIO pins. This binary encoding is then decoded back to a floating-point value on the Arduino and presented on the display. A custom plotting function for the LCD was also written to display the ABP and CBFV waveforms on the LCD display and is included in Appendix C.

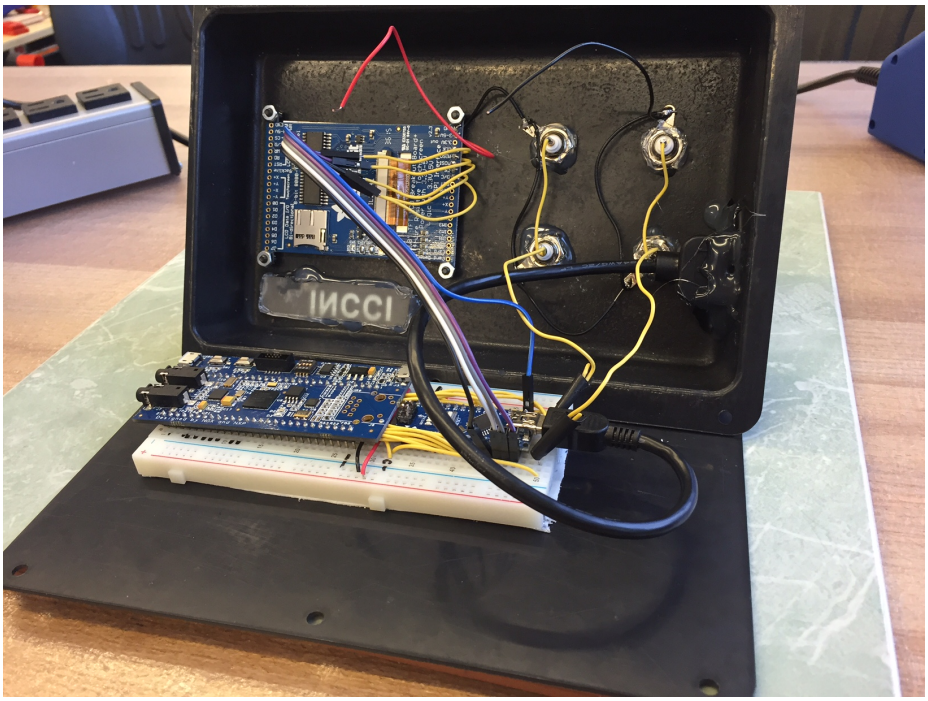

Figure 3-4: The messy innards of the medical device prototype. An enclosure, fabricated out of black acrylonitrile butadiene styrene (ABS, makes it much more user-friendly.

3.3.5 Housing

The embedded device is truly a medical device prototype and so packaging the device and peripherals in a custom casing was desired. The casing was made out of black acrylonitrile butadiene styrene (ABS) that was thermoformed into a box shape. Holes for the LCD, USB port, and BNC analog input jacks for the TCD monitor and Nexfin were cut with a laser cutter. A clear acrylic panel was secured in front of the LCD for protection and hot glue applied to the BNC jacks to make attachment easier through the rotational locking mechanism. As well, two additional BNC jacks were attached and wired to the input jacks as "passthroughs" to allow other devices to use the TCD monitor and Nexfin analog signals. The SD card is available for use after long-term data collection by opening the case (Figure 3-4).

Chapter 4

Results and Discussion

The medical device prototype described in the last chapter was tested on ABP and CBFV data from a traumatic brain injury patient at Addenbrooke's Hospital in Cambridge University as well as data recorded in real-time from a volunteer subject. ICP estimates were compared to those obtained by running the batch-mode implementation on the same dataset. This chapter includes these results together with analysis of the memory usage, performance, and physical specifications of the final assembled device. As well, I here examine how these results reflect the limitations and advantages of the device and discuss the future avenues of research that they suggest.

4.1 Comparing to Batch-Mode

In the first trial dataset, obtained from Cambridge University, 17 minutes of archived radial artery ABP and CBFV data were used from a comatose traumatic brain injury patient (Figure 4-1, middle and bottom). The patient was in the intensive care unit for the duration of the recording. The archived waveforms were streamed to the device to simulate a real-time recording. In the second dataset, 20 minutes of ABP and CBFV data were collected from a volunteer subject using the Nexfin and a TCD monitor (Figure 4-2, middle and bottom), streamed in real-time to the device, and saved to the SD card along with the computed ICP estimates. The subject was calmly sitting upright in a chair for the duration of the experiment.

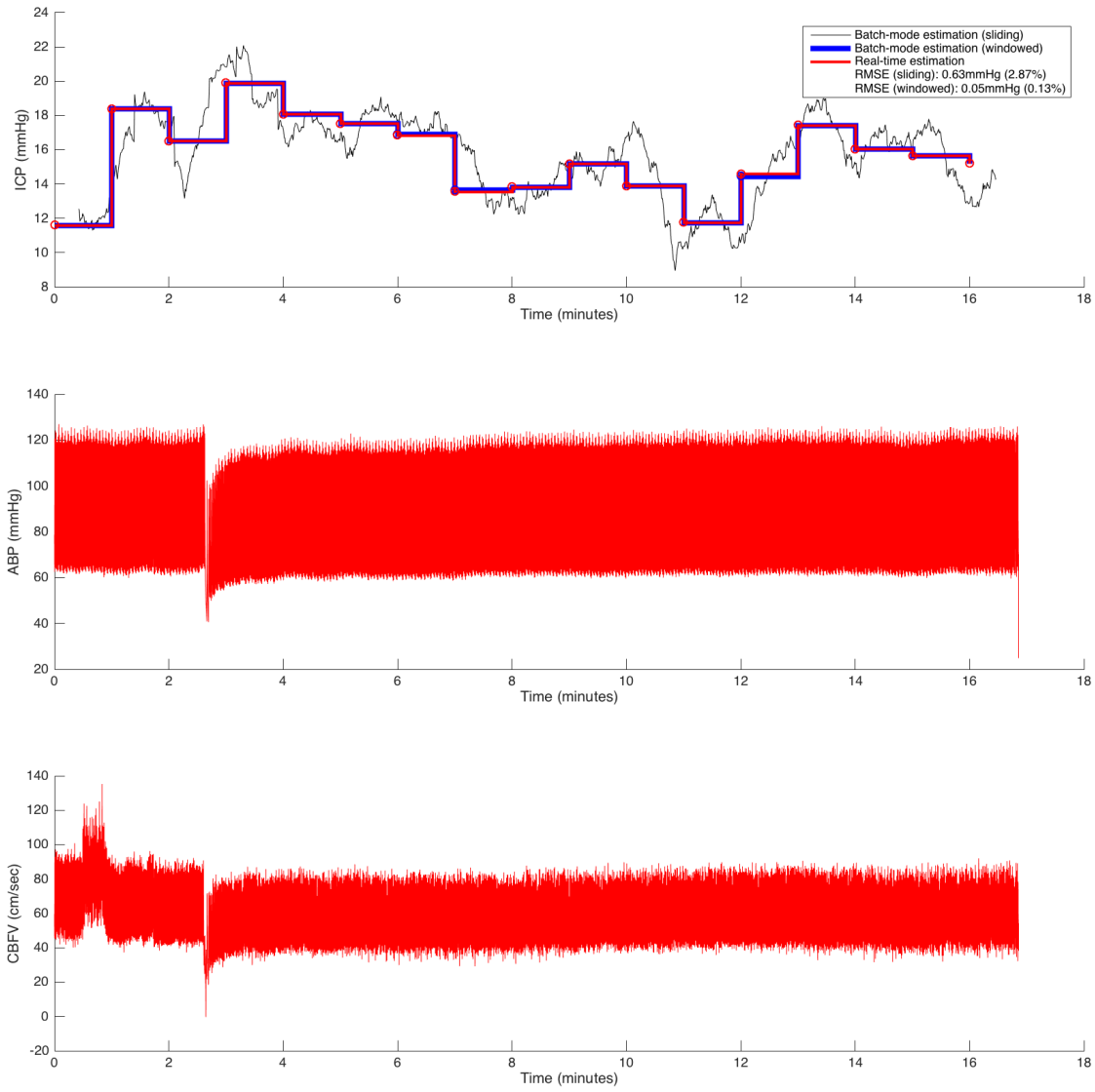

Figure 4-1: ICP estimation results from data obtained at Addenbrooke's Hospital, Cambridge University. Top plot: comparison of ICP estimates produced by the sliding and non-overlapping batch-mode implementation to those produced by the real-time implementation on the MCU. Middle plot: ABP waveform. Bottom plot: CBFV waveform.

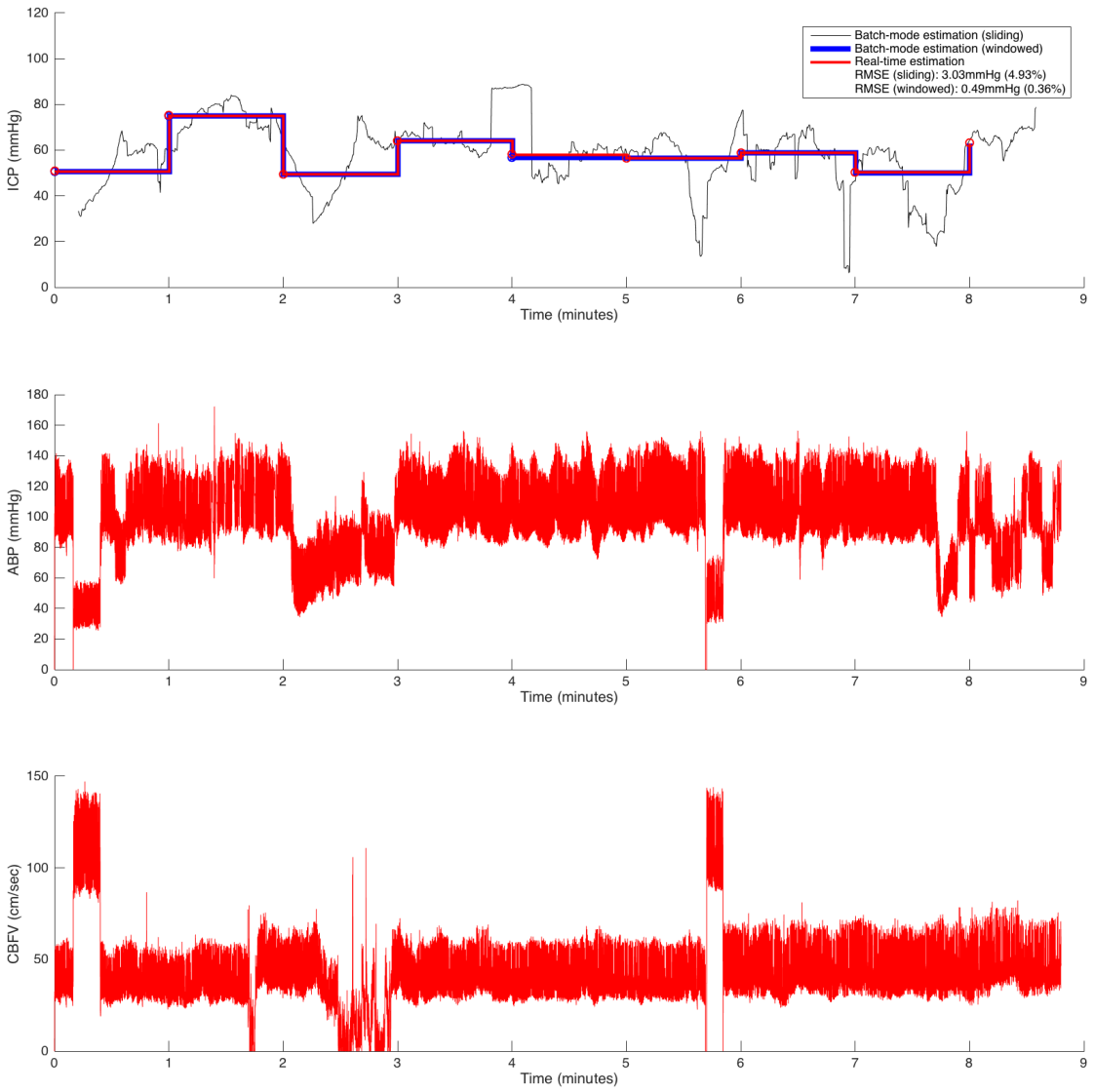

Figure 4-2: ICP estimation results from 20 minutes of data obtained in real-time. Top plot: comparison of ICP estimates produced by the sliding and non-overlapping batch-mode implementation to those produced by the real-time implementation on the MCU. Middle plot: ABP waveform. Bottom plot: CBFV waveform.

To compare the real-time approach to the batch-mode approach, the ICP estimates were compared to ICP estimates obtained by running the MATLAB batch-mode algorithm on the same waveforms. The real-time ICP estimates differed from the batch-mode estimates by a root-mean-square error (RMSE) of 0.63 mmHg and a mean-absolute-percentage error (MAPE) of 2.87% in the Cambridge trial (Figure 4-1, top) and by a RMSE of 3.03 mmHg and a MAPE of 4.93% in the 20-minute trial (Figure 4-2, top). In the 20-minute trial, the ICP estimates were regularly above 40 mmHg and appeared unstable, reaching as high as 80 mmHg and as low as 30 mmHg within less than one minute. The poor quality of these estimates is most likely due to the unreliability of the ABP and CBFV waveforms, which can be seen to be unstable themselves.

The discrepancy between the estimations produced by the two implementations is largely due to the difference in estimation windows. In the batch-mode algorithm, a 60-beat window is slid across the prerecorded dataset, one heartbeat at a time; in the real-time algorithm, 60 seconds of data are collected and then processed to produce a single ICP estimate. The least-squares regression steps in the Kashif algorithm are handling different systems of equations in the two approaches, which can produce different estimates of R and C. To verify this, the batch-mode algorithm was run using non-overlapping windows of 60 seconds and compared to the real-time estimates. The ICP estimates differed by an RMSE of 0.05 mmHg and MAPE of 0.13% in the Cambridge trial and by an RMSE of 0.49 mmHg and MAPE of 0.36% in the 20-minute trial.

The remaining discrepancy is entirely attributed to `wabp`. `wabp` is run on the entire dataset at once in the batch-mode implementation (in both the sliding and non-overlapping runs) but is run once per 60-second window in the real-time implementation. As discussed, `wabp` is an adaptive algorithm, and so having access to the full 17- and 20-minute waveforms will produce different results than those produced from the windowed 60-second waveforms. By running the batch-mode algorithm with heartbeat onsets computed on 60-second non-overlapping windows, the ICP estimates produced are precisely equivalent to the estimates produced by the LPC4337.

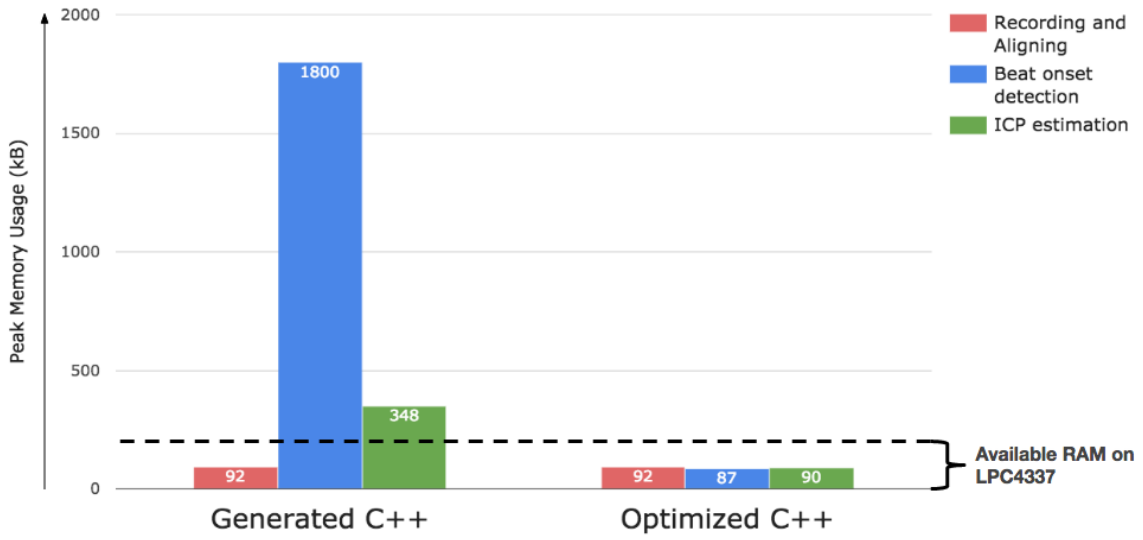

Figure 4-3: The C++ implementations generated through MATLAB Coder required more memory than was available on the LPC4337. Optimization of these implementations brought the total memory usage to under 100 kB for all steps.

4.2 Memory Usage

The C++ implementations of the `preprocess`, `wabp`, and `getICP` scripts generated by MATLAB Coder, respectively, required 92 kB, 1.8 MB, and 348 kB of memory. Because the LPC4337 has 136 kB of available SRAM, the latter two implementations were optimized to use as little memory as possible. After the optimization process, `wabp` and `getICP` respectively required 87 kB and 90 kB of RAM (Figure 4-3).

As discussed in the last chapter, the major memory efficiency boosts were achieved through reusing allocated memory, constraining array sizes based on reasonable physiological parameter ranges, and performing operations in-place at the cost of speed. While this approach was successful in the end at fitting the code onto the board, other possible approaches include using other memory stores on the LPC4337 and LPC4330-Xplorer board or developing on a different MCU with more memory. Constraining memory usage to only SRAM (instead of perhaps the SD card or on-board flash memory) simplified memory management, as additional software and computational time would be needed to save waveforms and workspace data and retrieve

them on-the-fly.

4.3 Performance

The `wabp`, and `getICP` scripts generated by MATLAB Coder respectively took, on average, 10 ms and 19 ms to process 60 seconds of 125 Hz ABP and CBFV data on a computer with a 2.5 GHz processor. The memory-optimized implementations, respectively, took, on average, 13 ms and 27 ms to complete the same task. This increase in processing time is a result of the use of in-place operations to conserve memory at the cost of speed. The total runtime of the ICP estimation procedure took 52 ms, on average, on the computer. The computer's processor is 12.5 times faster than the Cortex-M4 on the LPC4337, and one might therefore expect an upper-bound MCU runtime of around 650 ms was expected. The MCU took 410 ms, on average, to compute each ICP estimate. As the computer also uses time-sharing and memory caching, it was predictable that the MCU would perform considerably faster than the upper-bound runtime. An ICP estimate can therefore be computed after collecting 60 seconds of ABP and CBFV data at 125 Hz with less than a second of processing. This opens up the future possibility for a sliding-window real-time implementation of the ICP estimation algorithm to produce continuous nICP estimates.

4.4 Final Device Specifications

The final assembled device prototype (Figure 4-4) measures 21cm x 15.2cm x 5.3cm and weighs 330g. It features a backlit LCD screen that displays the estimated ICP as well as the ABP and CBFV waveforms obtained through analog inputs to two BNC jacks on the front panel. These BNC jacks are passed through to two more BNC jacks so that other devices have access to the TCD monitor and Nexfin analog signals. The device is powered at 5V by a USB-B supply, which can also be connected to a computer to read streaming ICP estimates. The device produces an estimate of mean ICP once per minute and can perform the necessary computations in less than

a second. ICP estimates are also saved to an SD card inside of the device casing.

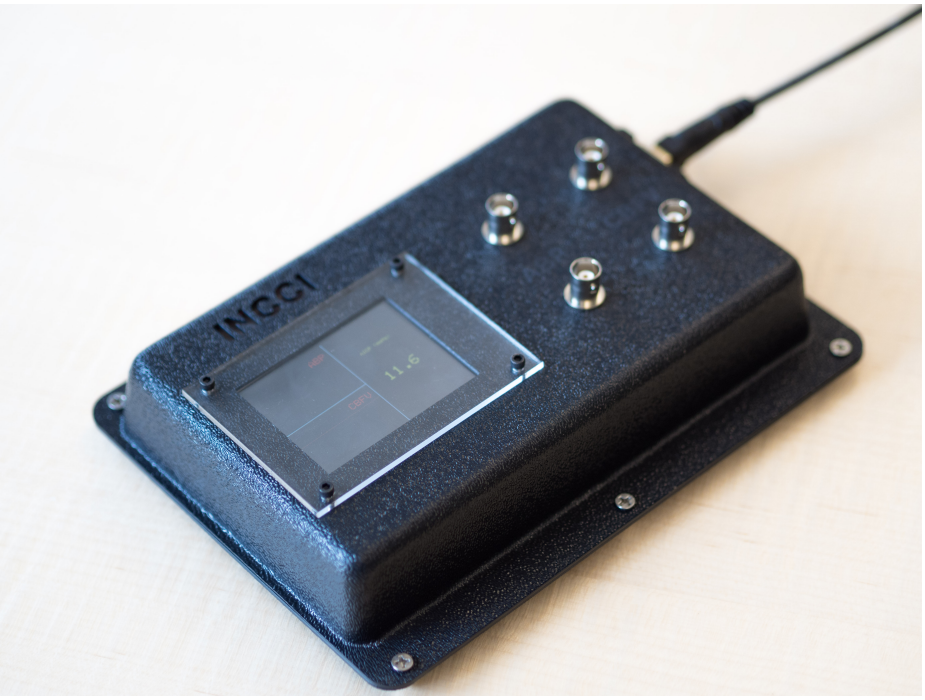

Figure 4-4: The assembled medical device prototype for real-time noninvasive intracranial pressure (ICP) estimation. Shown in Figure 1-3, but also here for convenience and closure.

Chapter 5

Contributions and Future Work

5.1 Contributions

This thesis has contributed to the endeavor of developing a reliable, noninvasive, and real-time method for measuring intracranial pressure by

- porting the Kashif algorithm to a microprocessor platform and optimizing it for real-time ABP and CBFV waveform acquisition
- fabricating a medical device prototype complete with peripheral interfaces for TCD and ABP monitoring hardware and display and recording functionality for clinical use and post-acquisition analysis
- verifying that a real-time windowed approach performs comparable to the batch-mode MATLAB implementation of the Kashif algorithm.

These contributions take a clear step toward the goal of real-time noninvasive ICP estimation in a variety of clinical settings.

5.2 Future Work

While the performance of the device has demonstrated the successful application of the Kashif algorithm to a real-time microprocessor-based application, there are several areas that need to be expanded upon to improve its clinical usefulness.

5.2.1 Nexfin ABP as a proxy for MCA ABP

A key input to the Kashif algorithm is the ABP at the level of the MCA, which is impossible to measure through noninvasive techniques. The Nexfin produces noninvasive measurements of ABP at a finger artery and so the pressure waveform must be transformed to represent the pressure at the MCA. This transformation involves the height difference between the finger and the MCA, morphological differences due to a number of physiological factors (see Section 2.2.2), and a time delay due to both the difference in the arrival time of the pressure wave as well as delays in the signal acquisition pathway (such as Nexfin processing time). Of these, only a simple alignment procedure has been implemented in the current device to match the heartbeat onset times of the ABP and CBFV waveforms. The alignment procedure should be improved and implemented on the device to preserve the physiological delay between MCA ABP and CBFV due to vascular compliance. The relationship between the ABP at the finger artery and the MCA should be more fully characterized and the corresponding transformation should also be implemented on the device.

5.2.2 Sliding estimation

The testing results of the real-time implementation on the device highlighted the difference between ICP estimates produced with a sliding window and those produced with non-overlapping windows. The device can run the computations necessary to carry out the Kashif algorithm in less than one second, and so there is a possibility for implementing a sliding window approach on the device. As ABP and CBFV waveforms are obtained, the samples would be shifted into the LPC4337 buffers and the estimation procedure carried out as fast as possible. This approach would allow the device to produce estimates at a rate more similar to that of invasive ICP measurement techniques and perhaps be more useful to a physician for neurocritical care monitoring.

Appendix A

ABP and CBFV Alignment Implementation

The MATLAB implementation of the signal alignment procedure discussed in Section 2.2.4 was manually ported to a C++ implementation.

```
#define LAGS_TO_TEST 10

int distance(uint16_t a, uint16_t b) {
        return abs((int)a - (int)b);
}

void zero(int16_t** signal, int length) {
        for(int i = 0; i < length; i++) {
                (*signal)[i] = 0;
        }
}

int getMedian(int16_t* data, int size) {
        std::vector<int16_t> to_sort(data, data+size);
        std::sort(to_sort.begin(), to_sort.begin()+size);
        return to_sort[to_sort.size()/2];
}
```

```
int getMinDifferenceLag(int16_t* ibd_abp, int16_t* ibd_cbfv, int
    num_ibds) {
        int16_t a_medianIB = getMedian(ibd_abp, num_ibds);
        int16_t b_medianIB = getMedian(ibd_cbfv, num_ibds);

        float min_correlation = -1;
        int min_lag = 0;

        for(int lag = -LAGS_TO_TEST/2; lag <= LAGS_TO_TEST/2; lag++) {
                int ibd_a_start, ibd_b_start = 0;
                if(lag < 0) {
                        ibd_a_start = 0;
                        ibd_b_start = -lag;
                } else if(lag > 0){
                        ibd_a_start = lag;
                        ibd_b_start = 0;
                }


                int total = 0;
                int n = 0;
                for(int index = 0; index < num_ibds-abs(lag); index++) {
                        if(abs(ibd_abp[index+ibd_a_start]-a_medianIB) <=
                             ((float)a_medianIB)*1.5 &&
                                        abs(ibd_cbfv[index+ibd_b_start]-
                                            b_medianIB) <= ((float)
                                            b_medianIB)*1.5) {
                                total += abs(ibd_abp[index+ibd_a_start]-
                                    ibd_cbfv[index+ibd_b_start]);
                                n++;
                        }
                }

                float correlation = (float)total/(float)n;
                if(min_correlation < 0 || correlation < min_correlation)
                    {
```

```
                    min_correlation = correlation;
                    min_lag = lag;
                }
            }

            return min_lag;
}

int align_signals(int num_abp_beats, uint16_t** abp_onsets, int
    num_cbfv_beats, uint16_t** cbfv_onsets,
                uint16_t* abp_align, uint16_t* cbfv_align) {
            int16_t* couples_abp = new int16_t[num_abp_beats];
            int16_t* couples_cbfv = new int16_t[num_abp_beats];

            zero(&couples_abp, num_abp_beats);
            zero(&couples_cbfv, num_abp_beats);

            int j = 0;

            uint16_t abp_start = (*abp_onsets)[0];
            uint16_t cbfv_start = (*cbfv_onsets)[0];

            // Find the closest beat for each beat
            for(int beat_number = 0; beat_number < num_abp_beats;
                beat_number++) {
                    uint16_t abp_onset = (*abp_onsets)[beat_number] -
                        abp_start;

                    // Find the closest beat:
                    int min_distance = distance(abp_onset,(*cbfv_onsets)[0]
                        - cbfv_start);
                    int closest_index = 0;
                    for(int partner_index = 1; partner_index <
                        num_cbfv_beats; partner_index++) {
                            int new_distance = distance(abp_onset, (*
                                cbfv_onsets)[partner_index] - cbfv_start);
```

```
                if(new_distance < min_distance) {
                        min_distance = new_distance;
                        closest_index = partner_index;
                }
        }

        uint16_t closest_onset = (*cbfv_onsets)[closest_index] -
            cbfv_start;

        // Resolve competitions for a particular beat
        if(beat_number == 0 || closest_onset != couples_cbfv[j
            -1]) {
                couples_abp[j] = abp_onset;
                couples_cbfv[j] = closest_onset;
                j++;
        } else {
                uint16_t competitor1 = abp_onset;
                uint16_t competitor2 = (*abp_onsets)[beat_number
                    -1];

                int distance1 = distance(competitor1,
                    closest_onset);
                int distance2 = distance(competitor2,
                    closest_onset);

                if(distance1 > distance2) {
                        couples_abp[j-1] = competitor2;
                } else {
                        couples_abp[j-1] = competitor1;
                }
        }
}

for(int i = 0; i < num_abp_beats; i++) {
        couples_abp[i] += abp_start;
        couples_cbfv[i] += cbfv_start;
```

```
}

int couples_length = j;
for(int i = 0; i < couples_length; i++) {
        couples_abp[i] = couples_abp[i+1]-couples_abp[i];
        couples_cbfv[i] = couples_cbfv[i+1]-couples_cbfv[i];
}

int differences_length = couples_length-1;

int beat_lag = getMinDifferenceLag(couples_abp, couples_cbfv,
    differences_length);

int16_t offset = 0;
*abp_align = 0;
*cbfv_align= 0;

if(beat_lag < 0) {
        offset = ((int16_t)(*abp_onsets)[0])-((int16_t)(*
            cbfv_onsets)[-beat_lag]);
} else {
        offset = ((int16_t)(*abp_onsets)[beat_lag])-((int16_t)(*
            cbfv_onsets)[0]);
}

offset += 1;

if(offset < 0) {
        *cbfv_align = -offset;
        for(int i = 0; i < num_cbfv_beats; i++) {
                (*cbfv_onsets)[i] = (int16_t)(*cbfv_onsets)[i] +
                    offset;
        }
} else {
        *abp_align = offset;
        for(int i = 0; i < num_abp_beats; i++) {
```

```
                    (*abp_onsets)[i] -= offset;
                }
            }

            return beat_lag;
        }
```

Appendix B

Continuous Memory Manager Implementation

A lightweight framework was used to provide virtual contiguous memory access to the separate RAM blocks on the LPC4337.

```
class ContinuousMemory {
public:
        /* Constructor. Creates a new instance of a ContinuousMemory.
         * Arguments:
         *      capacity (int) -- the maximum number of memory blocks
            that can be added.
         */
        ContinuousMemory(int capacity) {
                capacity = c;
                num_memories = 0;
                memories = new float*[capacity];
                sizes = new uint16_t[capacity];
        }
        virtual ~ContinuousMemory() {
                delete[] memories;
                delete[] sizes;
        }

        /* AddMemory. Adds a memory block to the continuous memory.
```

```
 * Arguments:
 *      memory (float*) -- a pointer to an allocated block of
    memory.
 *      size (uint16_t) -- the size of the allocated memory
    block in bytes/4.
 * Returns:
 *      bool -- True if the memory was successfully added. False
     otherwise.
 */
bool AddMemory(float* memory, uint16_t size) {
        // If we are at capacity, we cannot add more memories.
        if(num_memories >= capacity) {
                return false;
        }

        // Add the new memory.
        memories[num_memories] = memory;
        sizes[num_memories] = size;
        num_memories++;
        return true;
}

/* Set. Sets a 4-byte value in the continuous memory space.
 * Arguments:
 *  address (uint32_t) -- the address of the memory location in
    the continuous memory space. Valid addresses are from 0 to
    GetTotalSize().
 *  value (float) -- the value to write into the memory location
    .
 * Returns:
 *      bool -- True if the address was valid and the write
    completed successfully. False otherwise.
 */
bool Set(uint32_t address, float value) {
        // Locate the memory in which the address falls.
```

```
        for(uint8_t memory = 0; memory < num_memories; memory++)
            {
                if(address >= sizes[memory]) {
                        // Still looking...
                        address -= sizes[memory];
                } else {
                        // Found it!
                        memories[memory][address] = value;
                        return true;
                }
        }

        // The address is too large.
        return false;
}

/* Get. Retrieves a 4-byte value in the continuous memory space.
 * Arguments:
 *  address (uint32_t) -- the address of the memory location in
    the continuous memory space. Valid addresses are from 0 to
    GetTotalSize().
 * Returns:
 *      float -- The value found in the memory location. Will
    return 0 if the memory location was not found.
 */
float Get(uint32_t address) {
        // Locate the memory in which the address falls.
        for(uint8_t memory = 0; memory < num_memories; memory++)
            {
                if(address >= sizes[memory]) {
                        // Still looking.
                        address -= sizes[memory];
                } else {
                        // Found it!
                        return memories[memory][address];
                }
```

```
        }

        // The address is too large.
        return 0;
}

/* SetArray. This function serves the same purpose as the native
    memset() method, setting a block of memory to a particular
   value.
 * Arguments:
 *      start_address (uint32_t) -- the first address in
    continuous memory space to set to value. Valid addresses
    are from 0 to GetTotalSize().
 *      value (float) -- the value to write into the block of
    memory
 *      size (uint32_t) -- the number of 4-byte locations in
    which to write value. This will set start_address->
    start_address+size to value.
 * Returns:
 *      bool -- True if the address was valid and the write
    completed successfully. False otherwise.
 */
bool SetArray(uint32_t start_address, float value, uint32_t size
   ) {
        // Call Set() on each address from start_address to
           start_address+size.
        for(uint32_t addr = start_address; addr < start_address+
            size; addr++) {
                if(!Set(addr, value)) {
                        return false;
                }
        }
        return true;
}
```

```
/* Copy. This function serves the same purpose as the native
   memcpy() method, setting a block of memory to a set of
   values.
 * Arguments:
 *  dest (CMemPointer*) -- a pointer to a valid CMemPointer
    object that represents the starting address into which to
    write copied data.
 *  src (CMemPointer*) -- a pointer to a valid CMemPointer
    object that represents the starting address of data to copy
    .
 *      size (uint32_t) -- the number of 4-byte locations to
    copy.
 * Returns:
 *      bool -- True if the address was valid and the write
    completed successfully. False otherwise.
 */
bool Copy(CMemPointer* dest, CMemPointer* src, uint32_t size)(
   CMemPointer* dest, CMemPointer* src, uint32_t size) {
        for(uint32_t num = 0; num < size; num++) {
                if(!dest->Set(num, src->Get(num))) {
                        return false;
                }
        }
        return true;
}

/* GetTotalSize. Retrieves the total number of 4-byte (float)
   locations in the continuous memory.
 * Returns:
 *      uint32_t -- The total number of 4-byte (float) locations
     added to the continuous memory.
 */
uint32_t GetTotalSize() {
        uint32_t sum = 0;
        for(uint8_t memory = 0; memory < num_memories; memory++)
            {
```

```
                sum += sizes[memory];
            }
            return sum;
        }
private:
        int num_memories;
        int capacity;
        float** memories;
        uint16_t* sizes;
};

class CMemPointer {
public:
        /* Constructor. Creates a new instance of a CMemPointer.
         * Arguemnts:
         *      memory (ContinuousMemory*) -- a pointer to a valid
            ContinuousMemory object that manages the continuous memory
            space to which
         *
                    this CMemPointer will point.
         *      start_address (uint32_t) -- the address of this pointer
            in continuous memory space.
         */
        CMemPointer(ContinuousMemory* memory, uint32_t start_address) {
                start = start_address;
                mem = memory;
        }

        /* Get. Retrieves a value in the memory space indexed from the
           starting address of this CMemPointer object. This serves the
            same
         * purpose as array indexing:
         *
         * float* temp = new float[100];
         * cout << temp[3];
```

```
 *
 * CMemPointer* ctemp = new CMemPointer(memory, 0);
 * cout << ctemp->Get(3);
 *
 * Arguments:
 *      address (uint32_t) -- the address (indexed from the
    starting address of this CMemPointer object) to retrieve.
 * Returns:
 *      float -- The value in the retrieved area of memory.
 */
float Get(uint32_t address) {
        return mem->Get(start+address);
}

/* Set. Writes a value in the memory space indexed from the
   starting address of this CMemPointer object. This serves the
    same
 * purpose as array indexing:
 *
 * float* temp = new float[100];
 * temp[3] = 20.14;
 *
 * CMemPointer* ctemp = new CMemPointer(memory, 0);
 * ctemp->Set(3, 20.14);
 *
 * Arguments:
 *      address (uint32_t) -- the address (indexed from the
    starting address of this CMemPointer object) to write.
 *      data (float) -- the value to write into memory.
 * Returns:
 *      bool -- True if the address was valid and the write
    completed successfully. False otherwise.
 */
bool Set(uint32_t address, float data) {
        return mem->Set(start+address, data);
}
```

```
        // Accessor method for the CMemPointer start_address.
        uint32_t GetStart() {
                return start;
        }

        // Accessor method for the ContinuousMemory to which this
           CMemPointer points.
        ContinuousMemory* GetMemory() {
                return mem;
        }

private:
        uint32_t start;
        ContinuousMemory* mem;
};
```

Appendix C

Plotting ABP and CBFV on the LCD

A simple plotting function was implemented on an Arduino Nano based on the adafruit graphics library [1] to display ABP and CBFV waveforms on the LCD display.

```
void displayPlot(bool drawTitle, String text, int text_size, uint16_t
    color, int x, int y, int width, int height, float plot_min, float
    plot_max, float* values, int num_values) {
  uint16_t text_width = text.length() * text_size * 6 - 1;
  int16_t cursor_x = (int16_t)(width - text_width - 7 + x);
  int16_t cursor_y = (int16_t)(y + 15);
  if(drawTitle) {
    tft.setTextSize(text_size);
    tft.setTextColor(color);
    tft.setCursor(cursor_x, cursor_y);
    tft.print(text);
  }

  int plotLeft = x;
  int plotRight = width + x;
  int plotTop = cursor_y + text_size*8 + 5;
  int plotBottom = y + height -10;
  int plotWidth = plotRight-plotLeft;
  int plotHeight = plotBottom-plotTop;
```

```
  tft.fillRect(plotLeft, plotTop, plotWidth, plotHeight+1, ILI9341_BLACK
      );
  for(int i = 0; i < num_values; i++) {
    float value = values[i];
    int xPos = i*((float)plotWidth)/((float)num_values) + plotLeft;
    if(value < plot_min) value = plot_min;
    if(value > plot_max) value = plot_max;

    float a = ((float)plotHeight)/(plot_min-plot_max);
    float b = plotTop + ((float)plotHeight)/(1 - (plot_min/plot_max));
    int yPos = a*value + b;
    tft.drawPixel(xPos, yPos, color);
  }
}
```

Bibliography

[1] adafruit. *https://www.adafruit.com*, May 2016.

[2] M. Czosnyka and J.D. Pickard. Monitoring and interpretation of intracranial pressure. *Journal of Neurology, Neurosurgery, & Psychiatry*, 75:813–821, 2004.

[3] E.E. Frezza and H. Mezghebe. Indications and complications of arterial catheter use in surgical or medical intensive care units: analysis of 4932 patients. *American Journal of Surgery*, 64(2):127–31, 1998.

[4] F.M. Kashif, G.C. Verghese, V. Novak, M. Czosnyka, and T. Heldt. Model-based noninvasive estimation of intracranial pressure from cerebral blood flow velocity and arterial pressure. *Science Translational Medicine*, 4(129):129–44, 2012.

[5] B. Mokri. The Monro-Kellie hypothesis: applications in CSF volume depletion. *Neurology*, 56(12):1746–8, 2001.

[6] J. Noraky. A spectral approach to noninvasive model-based estimation of intracranial pressure. Master's thesis, Massachusetts Institute of Technology, 2014.

[7] S.J. Pietrangelo. An electronically steered, wearable transcranial doppler ultrasound system. Master's thesis, Massachusetts Institute of Technology, 2013.

[8] NXP Semiconductor. LPCXpresso Platform. *https://www.lpcware.com/lpcxpresso*, May 2016.

[9] D. Popovic, M. Khoo, and S. Lee. Noninvasive monitoring of intracranial pressure. *Recent Patents on Biomedical Engineering*, 2:165–79, 2009.

[10] A. Ragauskas, V. Matijosaitis, R. Zakelis, K. Petrikonis, D. Rastenyte, I. Piper, and G. Daubaris. Clinical assessment of noninvasive intracranial pressure absolute value measurement method. *Neurology*, 78(21):1684–1691, 2012.

[11] Motueka High School. *http://learningon.theloop.school.nz/moodle/file.php/8/Waves/blood_flow.jpg*, May 2016.

[12] NXP Semiconductor. LPC435x/3x/2x/1x Product Data Sheet. *http://www.nxp.com/documents/data_sheet/LPC435X_3X_2X_1X.pdf*, May 2016.

[13] NGX Technologies. LPC4330-Xplorer Quick Start Guide. *http://shop.ngxtechnologies.com/product_info.php?products_id=104*, May 2016.

[14] J. Truijen, J.J. van Lieshout, W.A. Wesselink, and B.E. Westerhof. Noninvasive continuous hemodynamic monitoring. *Journal of Clinical Monitoring and Computing*, 26(4):267–8, 2012.

[15] M. Ursino and C.A. Lodi. A simple mathematical model of the interaction between intracranial pressure and cerebral hemodynamics. *Journal of Applied Physiology*, 82(4):1256–69, 1997.

[16] Valgrind. *http://www.valgrind.org*, May 2016.

[17] W. Zong, T. Heldt, G.B. Moody, and R.G. Mark. An open-source algorithm to detect onset of arterial blood pressure pulses. *IEEE Computers in Cardiology*, 30:259–262, 2003.