EI2520 TELPM Project Course

An open-source drive for controlling electrical machines

Project Report

Abhishek Maji, Arpit Gupta, Callum Hoare, Lee Shaobin
Lu Zhao, Sindhuja Balaji, Tamas Kovacs

Supervisors
Luca Peretti (KTH), Kristian Rönnberg (ABB)

January 7, 2019

Abstract

The objective of this project was to demonstrate that a fully-viable electric motor drive controller could be developed based on inexpensive, open-source electronics. The BeagleBone Black (BBB) and BeagleWire (BW) development boards were to be used as the basis for the proof-of-concept controller (coined the *DogDrive*). Motivating this effort was the expectation that open-source software and hardware could bring significant development opportunities to the proprietorial drives market with the increasing move towards vehicle electrification.

To complete the objective, an overall design for the drive controller was developed based on vector control techniques for an induction motor. The control model was designed based on mature control strategies, combined with the unified pulse width modulation scheme and implemented in the controller. The implementation uses both the BBB and the BW to their strengths but required the interface between them to be carefully developed. Available encoders and sensors were specified to enable the controller interfaces to be designed, although these interfaces were not implemented in their entirety. Finally, an induction motor model was implemented in the controller and in Simulink to enable system testing, independent from a physical motor.

The project was largely successful in its implementation of the controller, clearly demonstrating the core drive functionality required. However, it highlighted that the strict timing requirements for the drive control could not be met with the standard BBB operating system. Real time operating systems may potentially allow the BBB / BW platform to still be an open-source solution but this is left as an investigation for future work.

Contents

1 Terminology

- API: Application Programming Interface
- BBB: Beagle Bone Black (used interchangeably with DSP)
- BW: Beagle Wire (used interchangeably with FPGA)
- CPU: Central Processing Unit
- DogDrive: Given name for the drive controller developed in this project.
- DFO: Direct Field Orientation
- DSP: Digital Signal Processor
- DTS: Device Tree Structure
- DTO: Device Tree Overlay
- DTOB: DTO blob
- EMF: Electromotive Force
- FPGA: Field Programmable Gate Array
- GPIO: General-Purpose Input/Output
- GPMC: General Purpose Memory Controller
- GSOC: Google Summer of Code
- HMI: Human Machine Interface
- IGBT: Insulated Gate Bipolar Transistor
- IRQ: Interrupt Request
- IFO: Indirect Field Orientation
- IM: Induction Motor
- MOSFET: Metal-Oxide-Semiconductor Field-Effect Transistor.
- PI: Proportational Integral
- RG: Reference Generator
- SVPWM: Space Vector Pulse Width Modulation
- SPWM: Sinusoidal Pulse Width Modulation
- SPI: Serial Peripherary Interface
- SSH: Secure Shell
- VSC: Voltage Source Converter

2 Introduction

2.1 Background

The focus of this project is to create a drive controller for an electrical machine using open source hardware and software. The context for the project sits between two current global trends; the increasing demand for electric drives and the popularity of open-source electronics.

The existing drive market is dominated almost entirely by large corporate entities that use strictly guarded proprietary software and expensive, bespoke-designed embedded hardware controllers. In general, the software and embedded hardware industry has undergone significant development in recent years through the increasing prevalence of open-source tools. This is evidenced by the availability of Linux, Raspberry Pi, Arduino and general open-source software repositories. Open-source tools allow continuous improvement and development from a global community that can continually refine and remove isolated and possibly out-dated pockets of knowledge.

The increase in electric drive demand can be understood largely by the move towards vehicle electrification to address the threat of climate change, as shown in Figure 1. This move will greatly increase the prevalence and usage of electric drives.

By developing a drive based on open source software and hardware, it is believed that it could bring the open-source benefits to drive design and advance the ongoing development of the field. Ideally, the benefits developed in the open-source area would be reflected in all drive controllers and help assist the transition to electrification.

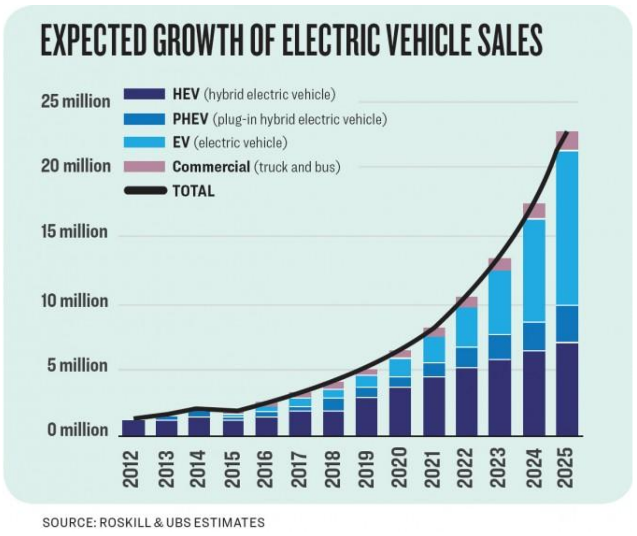


Figure 1: Expected EV sales [1]

This project builds on work previously completed by ABB researchers (L Peretti and K Rönnberg) in 2016. This previous work looked at creating a drive controller using a BeagleBone Black (BBB). The BBB is an open source embedded platform which has developed a large user base since its release in 2013. It is a low cost, low power, easily deploy-able digital signal processor (DSP) which has allowed its extended use in areas far beyond its original intent as an open source training board such as robotics, control systems and low load computing servers[2]. Alongside this, a large community has formed providing a solid support base for aiding in troubleshooting and queries. The development by the ABB team focused on determining whether the existing well-established drive algorithms were able to execute within the required timing criteria for adequate motor performance. This is the key challenge the team identified in the ability for the open-source controller to be utilised in a drive application.

The typical switching frequency of a motor drive is approximately 2-16kHz [3], in order to obtain accurate and robust control the controller must measure, process and output the required signals within the strictly specified time frame of the switching period. To enforce these constraints, a Pulse-Width-Modulated (PWM) signal must be generated based on system interrupts. During detection of an interrupt signal the algorithm needs to generate a new PWM signal in accordance to system parameters. However, this would include a significant delay of varying length that is present between the reception of the system interrupt and the generation of the control signal. The ABB team found that the effort to manage this delay was of too great a cost to pursue further and the project was put on hold. This project hiatus continued until early 2018 when a new open-source device was announced on the market, the BeagleWire.

The BeagleWire is an open-source, field programmable gate array (FPGA) development board which is designed specifically to integrate with the BBB. Through the precise timing of FPGAs and their ability to perform simultaneous operations, it gives the combined BeagleWire and BBB a strong potential to meet the sensitive drive timing requirements.

The objective of the project is to develop a drive controller on the combined BeagleWire and BBB platform. The intent is to demonstrate whether this platform is sufficient for meeting the technical requirements of a drive controller and that a fully open-source drive controller is achievable.

2.2 Objective

To develop a proof-of-concept drive controller on the BBB and BeagleWire development boards and test this controller to demonstrate that the open-source electronics can provide adequate performance. For easy-reference and promotional purposes, the controller has been named the "DogDrive" and will be referred to in this manner throughout the report.

Drive controllers can be of greatly varying complexity so the project has been split into specific deliverables as outlined below. The primary deliverables are those that are critical to demonstrating whether a drive controller can be achieved with the chosen electronics. The secondary deliverables are functions required for a working drive controller but are not required for a proof-of-concept and will only be implemented if time allows.

To limit the scope of the controller, it will be specifically designed for an induction motor using vector control techniques.

Primary Deliverables

1. Implement a functioning interface between DSP and FPGA
2. Implement vector control algorithms in DSP
3. Implement PWM interface on FPGA
4. Implement external sensor interface on the FPGA
5. Implement protection functions on the FPGA

Secondary Deliverables

1. Implement a functioning real-time human machine interface (HMI)
2. Integrated test of drive controller to physical sensors and gate drivers

2.3 Structure

This intent of this report is to give the reader a comprehensive understanding of the project from the theoretical basis to the manner of implementation and actual results obtained. In conjunction with the software repository (DogDrive), this report should give sufficient information for groups looking to further the work presented.
The report is separated as follows:

- Introduction - Outlines motivation for developing the drive controller and the sub-objectives for its achievement
- Technical Description - Describes the overall design developed for the DogDrive from a hardware and software architecture perspective. This provides context for the theoretical and implementation descriptions.
- Theory - Details the theoretical basis for the controller design and the algorithms implemented in the DSP and FPGA. Specifically, the motivation behind the motor model, vector controller and modulation scheme and the physical basis for the speed and sensor measurement.
- Method - Provides a detailed guide to the software implemented on the DSP and FPGA and the interface between them.
- Application - Presents the results of the controller implemented in terms of performance and relates back to individual objectives and overall goal as a proof-of-concept demonstration.
- Closure - Concludes the project and advises the authors' recommended path for future work.

3 Technical Description

This section provides a technical overview of the designed controller. This includes an overview of the key controller hardware and the software architecture. Additionally, the purpose of the periphery hardware which is not strictly part of the controller is given to provide context for the reader unfamiliar in electric motor drives. As described in Section 5, not all of these functions were implemented in the final controller, however, it is important for the reader to understand the context of the overall design.

Generally, a drive controller is the part of an electric drive that controls the motor to ensure it runs at the desired speed. It does this by reading in physical values from sensors mounted to the motor, calculates the required voltages to be applied to the motor phase windings to obtain the reference speed and determines how the supply voltage should be switched to achieve this [4].

Figure 2 provides a block diagram of the overall drive. The main controller elements are the **DSP** and **FPGA** blocks and the **interface** between them, while each sub-block represents a piece of software functionality. The items outside the DSP and FPGA are the drive hardware peripherals.

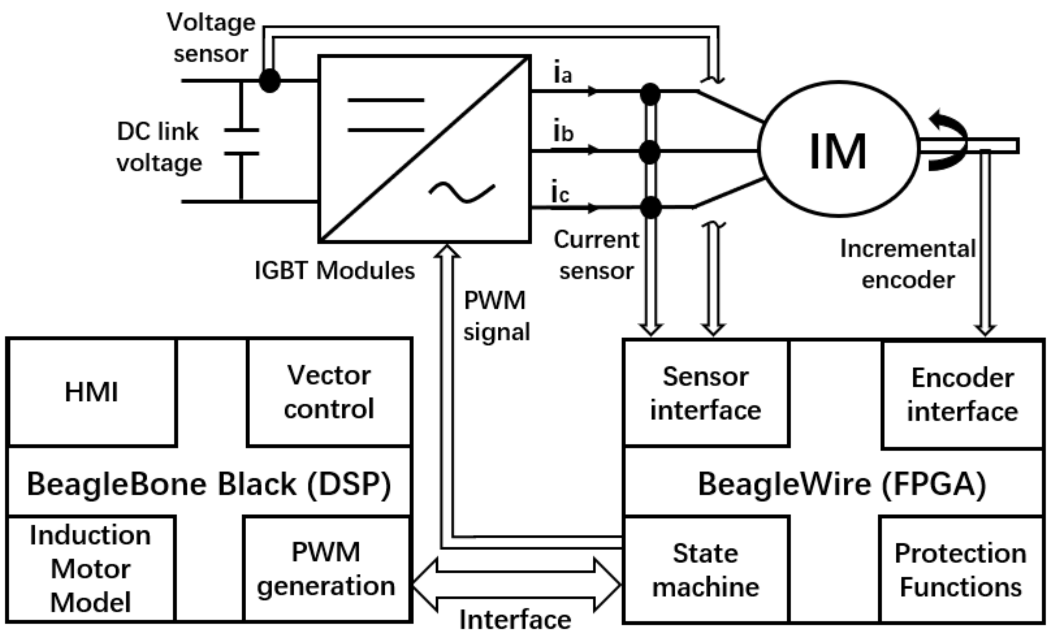


Figure 2: The principal interconnection block scheme

3.1 DSP

The DSP is part of the BeagleBone Black (BBB) which is an open source development board based on an 1 GHz ARM cortex_A8 processor and runs a Linux based operating system (in this case Debian Stretch 9.5). The board has two 46 pin expansion headers which is how the BeagleWire (BW) mounts and interfaces to the BBB [2].

The main control algorithm, the **vector control**, is handled by the DSP. At a high level this control is completed using two cascaded control loops for speed and current control. These control loops use the actual values provided by the sensor data which has been collected by the FPGA. The output of these control loops are the phase voltage references that are required to achieve the desired speed. These voltage references are passed to the **PWM generation** function block which uses the measured DC link voltage value to modulate the signals that go through the FPGA to the insulated gate bipolar transistor (IGBT) modules, i.e. it determines the switch on and switch off times for each module.

The DSP also completes two other functions, the HMI and the Induction Motor Model. The **HMI** provides the interface to the controller, allowing the user to view the measured data collected from the sensors and interact with the motor (start, stop and change the speed). The **Induction Motor Model** is implemented on the DSP as a debugging tool. It is a simulated version of an induction motor which allows the controller to be run without a physical motor / sensors and observe the behaviour of the controller.

3.2 Interface

The DSP and FPGA both need to be configured in a highly specific way in order to be able to pass the sensor information, PWM switching times and error codes between them. While the BW is designed to interface with the BBB, to ensure that this is completed in line with the drive requirements requires configuration on both the BBB and BW. The implementation is addressed in detail in Section 5.2.

3.3 FPGA

The FPGA is part of the BW, an open-source development board based on the Lattice iCE40HX FPGA. It was released in 2018 and is designed to integrate to the BBB. As it utilises most of the pin headers to the BBB, it includes 4 peripheral modules (PMOD), outputs for external interfacing. It was created by two members of the BeagleBone open-source community (M Welling and P Mezydlo) and has a small software library that has been created for it (BeagleWire) [5].

There are four main functions implemented on the FPGA. Primarily, it acts as a **state machine** to ensure correct timing of all the functions of the controller, in particular controlling the PWM output pins from the switching times communicated from the DSP. The state machine also controls the timing for collecting data from the **sensor interface** which reads data from the physical current and voltage sensors and converts it to a representative value that can be communicated to the DSP. Similarly, the **encoder interface** reads the data from the physical encoder and converts it to a speed value that can be communicated to the DSP. Finally, the FPGA includes a **protection function** block that ensures the current, voltage and speed limits are not exceeded and shuts down the motor if they are.

3.4 Peripherals

For completeness, the purpose of the hardware peripherals are briefly outlined below. The sensors and encoders are discussed further in Section 4 as they are used to inform the design of the software interface.

- **Voltage sensor** - Measures the voltage on the DC link for use in modulation scheme.
- **Current sensor** - Measures the current on the three phases for use in current controller.
- **Incremental encoder** - Encoder detects the rotation of the motor shaft for calculating speed.
- **IGBT Modules** - Takes the PWM output pins from FPGA as switch on input for each phase. This is the input to the module gate driver, which controls the switching of the IGBTs. It is assumed that the modules address minimum switching time and deadband. [6]

3.5 Simulink Model

To confirm the operation of the DSP implemented vector control and induction motor model, a Simulink model is created of the same control scheme and model. The results of the Simulink model are used to compare against the output of the DSP. In both cases the algorithms described in Section 4 are used. The Simulink model used for this comparison is largely similar to that developed in KTH course “EJ2230 Control in Electrical Energy Conversion”. For this reason, the implementation of the model will not be discussed in this report. However, an overview can be found in Appendix D. [7]

4 Theory

This section describes the physical and mathematical models behind the algorithms that are implemented in the controller. It provides the detailed theory behind the function blocks described in Section 3. Since the HMI, state machine and protection functions do not have a strong theoretical basis they are not discussed.

Note that some of the concepts presented in this section are quite specialised and while the explanations are intended to be generally approachable, a basic understanding of electrical machine models is assumed.

4.1 Induction Motor Model

In this section, a dynamic model of the induction motor is derived in order to implement it within the DSP and Simulink models. This also provides background for the vector control to be developed on. The dynamic inverse-Γ-equivalent circuit shown in the Figure 3 is considered for this purpose [7], [8], [9].

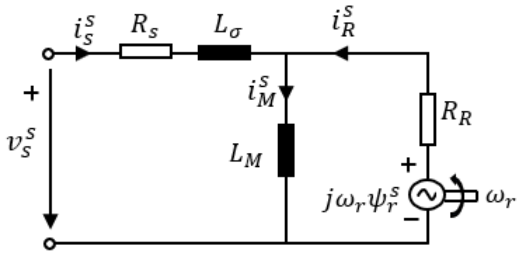


Figure 3: Dynamic inverse-Γ equivalent circuit for the induction motor

The induction machine in the stator reference frame shown in the Figure 3 is described by the following two (stator and rotor) vectorial differential equations as in [7] - [9]:

$$L_\sigma \frac{d\mathbf{i}_s^s}{dt} = \mathbf{v}_s^s - R_s\mathbf{i}_s^s - \frac{d\boldsymbol{\psi}_R^s}{dt} \tag{1}$$

$$\frac{d\boldsymbol{\psi}_R^s}{dt} = R_R\mathbf{i}_s^s - \Big(\frac{R_R}{L_M} - j\omega_r\Big)\boldsymbol{\psi}_R^s \tag{2}$$

Where $\mathbf{v}_s^s$, $\mathbf{i}_s^s$, and $\boldsymbol{\psi}_R^s$ are the space vectors for stator voltage, stator current and stator flux linkage respectively. The superscript 's' represents the stationary reference frame while the subscripts 's' and 'R' stand for stator and rotor respectively. L_σ, R_s, R_R, and L_M represent leakage inductance, stator resistance, transformed rotor resistance and magnetizing inductance of the induction motor respectively. The electrical rotor speed, ω_r is computed from mechanical dynamics of the induction motor as follows:

$$J\frac{d\omega_m}{dt} = \frac{3n_p}{2K^2}\,\mathrm{Im}\Big((\boldsymbol{\psi}_R^s)^*\mathbf{i}_s^s\Big) - b\omega_m - \tau_l \tag{3}$$

Where ω_m, τ_l are the rotor mechanical speed and the load torque respectively. n_p and K represent number of pole pair and space vector scaling constant respectively.The mechanical rotor speed is related to the electrical rotor speed by $\omega_r = n_p\omega_m$.

In the stationary reference frame $\mathbf{i}_s^s$ and $\boldsymbol{\psi}_R^s$ are represented as:

$$\mathbf{i}_s^s = i_\alpha + ji_\beta \tag{4}$$

$$\boldsymbol{\psi}_R^s = \psi_\alpha + j\psi_\beta \tag{5}$$

Substituting (4), and (5) in (1), (2), and (3) and separating the real and imaginary parts of (1), and (2), current, flux and mechanical dynamics of the induction motor are derived as follows:

$$\frac{di_\alpha}{dt} = \frac{1}{L_\sigma}\Big(v_\alpha - (R_s + R_R)i_\alpha + \frac{R_R}{L_M}\psi_\alpha + \omega_r\psi_\beta\Big) \tag{6}$$

$$\frac{di_\beta}{dt} = \frac{1}{L_\sigma}\Big(v_\beta - (R_s + R_R)i_\beta + \frac{R_R}{L_M}\psi_\beta - \omega_r\psi_\alpha\Big) \tag{7}$$

$$\frac{d\psi_\alpha}{dt} = R_R i_\alpha - \frac{R_R}{L_M}\psi_\alpha - \omega_r\psi_\beta \tag{8}$$

$$\frac{d\psi_\beta}{dt} = R_R i_\beta - \frac{R_R}{L_M}\psi_\beta + \omega_r\psi_\alpha \tag{9}$$

$$\frac{d\omega_r}{dt} = \frac{3n_p^2}{2K^2J}\Big(\psi_\alpha i_\beta - \psi_\beta i_\alpha\Big) - \frac{b}{J}\omega_r - \frac{n_p}{J}\tau_l \tag{10}$$

Equations (6)-(10) are illustrated by the block diagram in Figure 4. As shown in Figure 4, there are three feedback loops where the outer loop is that of the back EMF. The mechanical dynamics of the induction motor are connected in parallel to the flux dynamics, and interact non-linearly with the electrical subsystem [7].

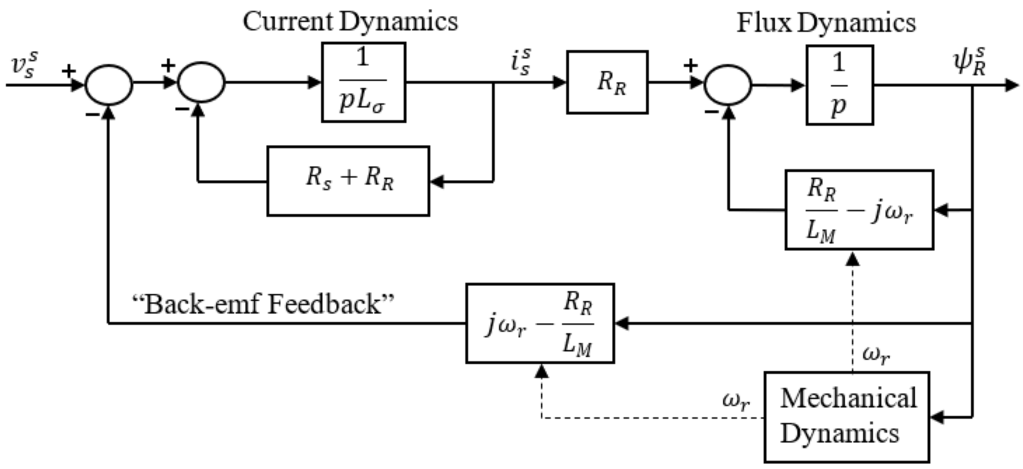


Figure 4: Block diagram of the induction motor dynamics. Solid and dashed lines indicate space-vector and scalar signal paths respectively

The induction machine model is now normalized with respect to base values in section A. The normalized induction machine model is represented as follows:

$$\frac{di_{\alpha n}}{dt_n} = \frac{1}{L_{\sigma n}}\Big(v_{\alpha n} - (R_{sn} + R_{Rn})i_{\alpha n} + \frac{R_{Rn}}{L_{Mn}}\psi_{\alpha n} + \omega_{rn}\psi_{\beta n}\Big) \tag{11}$$

$$\frac{di_{\beta n}}{dt_n} = \frac{1}{L_{\sigma n}}\Big(v_{\beta n} - (R_{sn} + R_{Rn})i_{\beta n} + \frac{R_{Rn}}{L_{Mn}}\psi_{\beta n} - \omega_{rn}\psi_{\alpha n}\Big) \tag{12}$$

$$\frac{d\psi_{\alpha n}}{dt_n} = R_{Rn}i_{\alpha n} - \frac{R_{Rn}}{L_{Mn}}\psi_{\alpha n} - \omega_{rn}\psi_{\beta n} \tag{13}$$

$$\frac{d\psi_{\beta n}}{dt_n} = R_{Rn} i_{\beta n} - \frac{R_{Rn}}{L_{Mn}} \psi_{\beta n} + \omega_{rn} \psi_{\alpha n} \tag{14}$$

$$J_n \frac{d\omega_{rn}}{dt_n} = \Big(\psi_{\alpha n} i_{\beta n} - \psi_{\beta n} i_{\alpha n}\Big) - b_n \omega_{rn} - \tau_{ln} \tag{15}$$

Equation (15) is not formally equal to the equation (10), as the factor $3n_p/2K^2$ has been removed by normalization.The induction machine model as described by equation (11) - (15) is in continuous time domain. To allow discrete-time implementation, discretization of the continuous model is needed . In this work, backward Euler discretization method [10] is used to discretize equations (11)-(15) as shown:

$$i_{\alpha n}(k+1) = \frac{1}{1 + \frac{R_n T_s}{L_{\sigma n}}} \Big(i_{\alpha n}(k) + \frac{T_s}{L_{\sigma n}} v_{\alpha n}(k+1) + \frac{R_{Rn} T_s}{L_{Mn} L_{\sigma n}} \psi_{\alpha n}(k) + \frac{T_s}{L_{\sigma n}} \omega_{rn}(k) \psi_{\beta n}(k) \Big) \tag{16}$$

$$i_{\beta n}(k+1) = \frac{1}{1 + \frac{R_n T_s}{L_{\sigma n}}} \Big(i_{\beta n}(k) + \frac{T_s}{L_{\sigma n}} v_{\beta n}(k+1) + \frac{R_{Rn} T_s}{L_{Mn} L_{\sigma n}} \psi_{\beta n}(k) - \frac{T_s}{L_{\sigma n}} \omega_{rn}(k) \psi_{\alpha n}(k) \Big) \tag{17}$$

$$\psi_{\alpha n}(k+1) = \frac{1}{1 + \frac{T_s R_R n}{L_{Mn}}} \Big(\psi_{\alpha n}(k) + R_{Rn} T_s i_{\alpha n}(k+1) - T_s \omega_{rn}(k) \psi_{\beta n}(k) \Big) \tag{18}$$

$$\psi_{\beta n}(k+1) = \frac{1}{1 + \frac{T_s R_{Rn}}{L_{Mn}}} \Big(\psi_{\beta n}(k) + R_{Rn} T_s i_{\beta n}(k+1) + T_s \omega_{rn}(k) \psi_{\alpha n}(k+1) \Big) \tag{19}$$

$$\omega_{rn}(k+1) = \frac{1}{1 + \frac{T_s b_n}{J_n}} \Big(\omega_{rn}(k) + \frac{T_s}{J_n} (\psi_{\alpha n}(k+1) i_{\beta n}(k+1) - \psi_{\beta n}(k+1) i_{\alpha n}(k+1) - \tau_{ln}(k+1)) \Big) \tag{20}$$

where T_s is the sampling time period, $R_n = R_{sn} + R_{Rn}$ represents the effective normalized resistance as seen from the stator side, and $x(k)$ represents the value of $x(t)$ at $t = kT_s$. The Equations (16)-(20) are used to model the induction motor in the DSP and to compute proportional and integral gains of the speed and current regulators.

4.2 Vector control

Vector control implies that a three-phase voltage source converter (VSC) is controlled using space vectors [7], [11]. In vector control, specifically field-oriented control in our case, a synchronously rotating $d-q$ frame is constructed, and current control is performed in this frame. The reason for such a choice of reference frame is that the quantities to control become constant rather than constantly oscillating in steady state and thus it is possible to control those quantities by using PI regulators. As the name suggests, the rotating $d-q$ reference frame is often aligned to the rotor flux of the induction motor.

Using the flux-aligned reference frame, it is possible to control the torque and the flux individually similar to a dc motor [7]. However, the rotor flux of an induction motor can not readily be measured. Therefore, a flux estimator of some kind is rudimentary to any IM vector control system. There are two types flux estimators: the current model and the voltage model [7], [11]. The current model is based on the rotor circuit of the IM while the voltage model relies on the stator circuit. For the DogDrive controller, the current model based flux estimator is implemented for its superior performance at higher speed [11].

For the operation above the base speed the flux must be reduced, since the stator voltage is limited by the converter. This is made by manipulating the reference i_d^{ref} for the flux producing current component by field weakening controller [12], [13]. This leads to the overall schematic diagram shown in Figure 5.

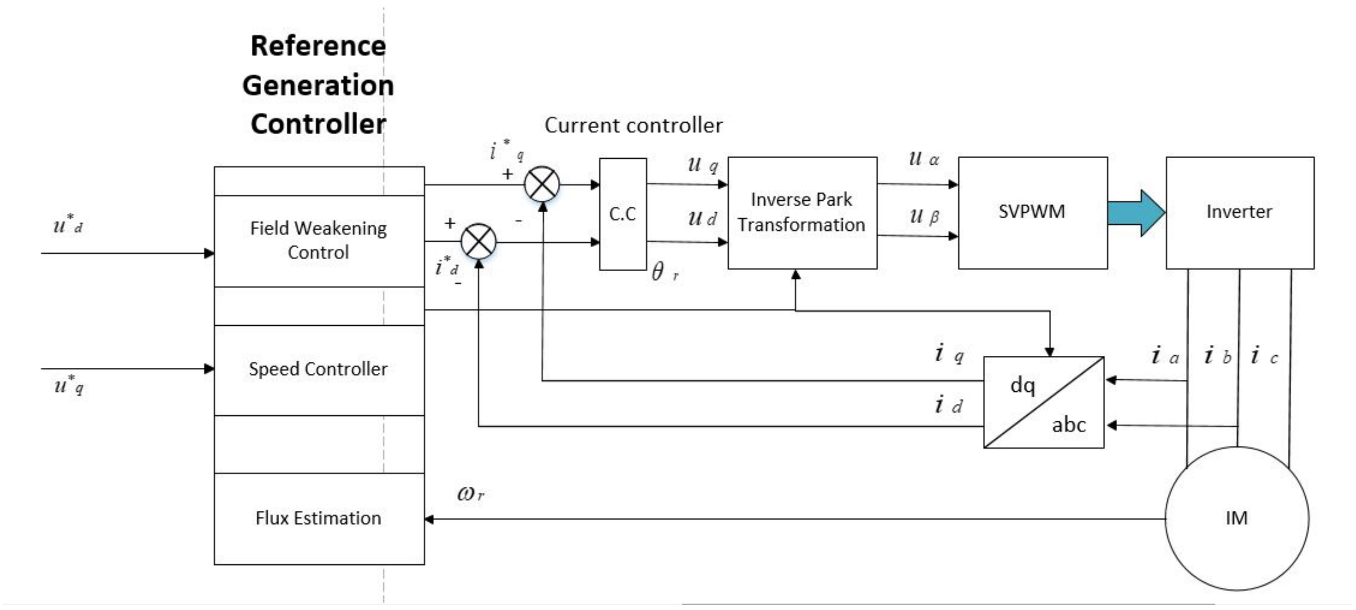


Figure 5: Overall diagram for vector control

4.2.1 D-Q transformation

To construct the synchronously rotating $d - q$ frame the Park-Clarke transformation is performed on the measured currents. This is justified as we can assume there is no zero component current (no ground connection) so the Clarke transform can be applied, representing the three currents as only two components i_α and i_β and the resulting space vector, $\mathbf{i}^s$, where the superscript represents the stationary reference frame (Equation 21). K represents the scaling constant described in Section A

$$\mathbf{i}_s^s = \begin{pmatrix} i_\alpha \\ i_\beta \end{pmatrix} = K \begin{pmatrix} \frac{2}{3} & -\frac{1}{3} & -\frac{1}{3} \\ 0 & \frac{1}{\sqrt{3}} & -\frac{1}{\sqrt{3}} \end{pmatrix} \begin{pmatrix} i_a \\ i_b \\ i_c \end{pmatrix} \tag{21}$$

However, this vector is not stationary as required from Section 4.2 but is rotating with an frequency equal to the angular synchronous frequency, ω_1. The Park transformation can then be applied using the angle corresponding to the synchronous frequency, θ_1 (Equation 22). This creates the stationary current components as required.

$$\begin{pmatrix} i_d \\ i_q \end{pmatrix} = \begin{pmatrix} cos\theta_1 & sin\theta_1 \\ -sin\theta_1 & cos\theta_1 \end{pmatrix} \begin{pmatrix} i_\alpha \\ i_\beta \end{pmatrix} \tag{22}$$

$$\mathbf{i}_s = e^{-j\theta_1}\mathbf{i}^s \tag{23}$$

4.2.2 Perfect field orientation

To apply the Park transformation the synchronous angular frequency is needed. In perfect field orientation, the $d - q$ coordinate system is oriented along the rotor flux of the IM i.e. the dq transformation angle is equal to the rotor flux angle as follows [7]:

$$\theta_1 = arg(\psi_R^s) \tag{24}$$

The $d - q$ reference frame is then be aligned with the rotor flux of the induction motor, so that ψ_R becomes real. Substituting this into the induction motor model in Equation 2, it can be shown that under perfect field orientation, the flux modulus has first order dynamics with input i_d, which is the d-axis component of stator current, i.e., $i_s = i_d + ji_q$ and rotor time constant $T_r = L_M/R_R$ [?]. In steady state,

$$\psi_R = L_M i_d \tag{25}$$

Thus, i_d controls the flux level and plays the role of the field current in dc motor. Therefore, i_d is called the flux-producing component, and should be controlled such that

$$i_d = \frac{\psi_{ref}}{L_M} \tag{26}$$

where ψ_{ref} is the is the flux reference, i.e., the desired flux level. With perfect field orientation, the torque is given by

$$\tau_e = \frac{3n_p}{2K^2} Im(\psi_R^* i_s) = \frac{3n_p}{2K^2}\psi_R i_q \tag{27}$$

The q-axis component of the stator current then controls the production of torque, just like the armature current of a dc motor. For this reason, i_q is called the torque-producing current component.

4.2.3 Reference Generation Controller

The Reference Generator is the outer controller which provides references for inner current controller. The outer controller reacts much slower than the inner controller and is designed based on the principle of cascade control [14], [15]. The inputs to the Reference Generator are the flux command and speed/torque command and it generates d and q-axis current references for the inner current controller. This means the Reference Generator includes three components; the field weakening controller, speed controller and rotor flux estimator. The Field Weakening Controller generates d-axis current reference based on the flux reference and the speed of the induction motor as shown in equation (33). Based on this d-axis current reference, a first order current model based flux estimator is implemented to compute the rotor flux magnitude as shown in equation (34). The magnitude of the rotor flux determines the proportional and integral gains of the speed controller. Based on the d-axis current reference and speed reference, speed controller generates q-axis current reference. Slip angular speed of the rotor is then estimated based on the q-axis current reference as shown in equation (44). Slip angular speed of the rotor along with the actual rotor speed from the speed sensor is used to estimate the position of the rotor flux in space which is utilized in the *d-q* transformation.

Field weakening control is integral control with k_{fw} as the integral gain. The algorithm for modifying the d-direction current reference is given by:

$$i_d^{ref} = k_{fw}\int_{I_{min}}^{I_{nom}} [V_{base}^2 - (v_d^{ref})^2 - (v_q^{ref})^2]\, dt \tag{28}$$

In this case, the nominal value of d-axis current should be selected as given in (26):

$$i_{nom} = \frac{\psi_{ref}}{L_M} \tag{29}$$

whereas the minimum value is selected large enough to prevent complete demagnetization [7] as:

$$i_{min} = 0.1 I_{nom} \tag{30}$$

To get reasonably good dynamic performance, the gain of the field weakening controller is selected appropriately [7], [14], [15].

$$k_{fw} = \frac{R_R}{L_\sigma^2 V_{base} max(|\omega_1|, \omega_{base})} \tag{31}$$

which yields a constant gain below base speed (where field-weakening controller is not used).
Discretization of the equation (28) results in:

$$i_d^{ref}(k) = i_d^{ref}(k-1) + \frac{R_R}{L_\sigma^2 V_{base} max(|\omega_1|, \omega_{base})}[V_{base}^2 - (v_d^{ref}(k-1))^2 - (v_q^{ref}(k-1))^2] \tag{32}$$

$$i_d^{ref}(k) = min[max(i_d^{ref}(k), I_{min}), I_{nom}] \tag{33}$$

With this value of i_d^{ref}, the rotor flux ψ_R can be estimated using the current model as follows:

$$\psi_R(k) = \frac{1}{1 + \frac{T_s R_R}{L_M}}\Big(\psi_R(k-1) + T_s R_R i_d^{ref}(k)\Big) \tag{34}$$

This estimated magnitude of the rotor flux ψ_R is used to compute the gains of the speed controller.
Speed regulator is a PI controller which is designed based on the cascade control principle. Speed controller bandwidth α_s is selected as:

$$\alpha_s = 0.1\alpha_c \tag{35}$$

where α_c is the current controller bandwidth. Recalling the normalized mechanical dynamics of the induction as shown in the equation (15), speed controller parameters can now be computed as:

$$b_a = \frac{(\alpha_s J - b)}{n_p \psi_R} \tag{36}$$

$$k_{ps} = \frac{\alpha_s J}{n_p \psi_R} \tag{37}$$

$$k_{is} = \frac{\alpha_s^2 J}{n_p \psi_R} \tag{38}$$

where, b_a, k_{ps}, and k_{is} is the active viscous damping constant, proportional and integral gain of the speed controller respectively. In the speed controller algorithm, it must be taken into account that limitation is necessary such that $i_d^2 + i_q^2 \leq I_{max}^2$, where I_{max} is the maximum allowed short-term stator current [7]. This yields the speed controller as follows:

$$e_s(k) = \omega_{ref} - \omega_r(k) \tag{39}$$

$$I_s(k) = I_s(k-1) + T_s e_s(k) \tag{40}$$

$$i_{q,nom}^{ref}(k) = k_{ps} e_s(k) + k_{is} I_s(k) - b_a \omega_r(k) \tag{41}$$

$$i_q^{ref}(k) = sat\Big(i_{q,nom}^{ref}(k), \sqrt{I_{max}^2 - (i_d^{ref}(k))^2}\Big) \tag{42}$$

In the event of saturation, i.e., when $i_{q,nom}^{ref}$ and i_q^{ref} are not same, anti-windup action of the integral action is implemented as modifying the integral part as follows:

$$I_s(k) = I_s(k-1) \tag{43}$$

Using this q-axis current reference (i_q^{ref}), the rotor flux position can be estimated by using slip relation as follows:

$$\omega_1(k) = \omega_r(k) + \frac{R_R i_q^{ref}(k)}{\psi_R(k)} \tag{44}$$

$$\theta_1(k) = mod\Big(\theta_1(k-1) + T_s\omega_1(k), 2\pi\Big) \tag{45}$$

Equation (44) is commonly known as slip relation. The rotor flux position θ_1 thus obtained is used as dq transformation angle to convert three phase stator currents into corresponding d-q current components.

4.2.4 Current Control

The inner current controller is a PI controller which produces d and q-axis voltage references based on the current references from the controller [7], [16]. Inner current controller is much faster than outer reference generator. The current controller bandwidth, α_c is thus computed to obtain the desired rise time, t_r (which is in the order of 'ms', [7]) as follows:

$$\alpha_c = \frac{\log_{10} 9}{t_r} \tag{46}$$

Recalling the normalized current dynamics of the induction as shown in the equation (11)-(12), current controller parameters is computed based on the principle of pole-zero cancellation [14], [15] as:

$$R_a = \alpha_c L_n - R_n \tag{47}$$

$$k_{pc} = \alpha_c L_n \tag{48}$$

$$k_{ic} = \alpha_c^2 L_n \tag{49}$$

where, R_a, k_{pc}, and k_{ic} is the active resistance damping constant, proportional and integral gain of the current controller respectively. The current control algorithm is formulated similar to speed controller as follows:

$$\begin{pmatrix} i_d(k) \\ i_q(k) \end{pmatrix} = \begin{pmatrix} cos\theta_1(k) & sin\theta_1(k) \\ -sin\theta_1(k) & cos\theta_1(k) \end{pmatrix} \begin{pmatrix} i_\alpha(k) \\ i_\beta(k) \end{pmatrix} \tag{50}$$

$$\begin{pmatrix} e_d(k) \\ e_q(k) \end{pmatrix} = \begin{pmatrix} i_d^{ref}(k) \\ i_q^{ref}(k) \end{pmatrix} - \begin{pmatrix} i_d(k) \\ i_q(k) \end{pmatrix} \tag{51}$$

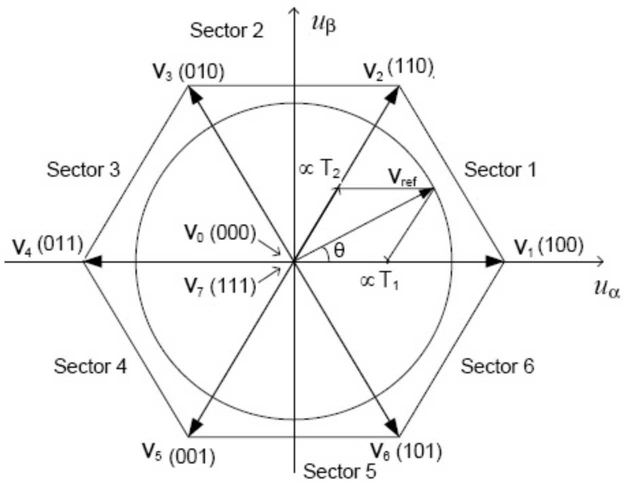
$$\begin{pmatrix} v_d^{ref} \\ v_q^{ref} \end{pmatrix} = K_p \begin{pmatrix} e_d(k) \\ e_q(k) \end{pmatrix} + K_i \begin{pmatrix} I_d(k) \\ I_q(k) \end{pmatrix} - \begin{pmatrix} R_a & \omega_1\hat{L} \\ -\omega_1\hat{L} & R_a \end{pmatrix} \begin{pmatrix} i_d(k) \\ i_q(k) \end{pmatrix} \tag{52}$$

$$\begin{pmatrix} I_d(k) \\ I_q(k) \end{pmatrix} = \begin{pmatrix} I_d(k-1) \\ I_q(k-1) \end{pmatrix} + T_s \begin{pmatrix} e_d(k) \\ e_q(k) \end{pmatrix} \tag{53}$$

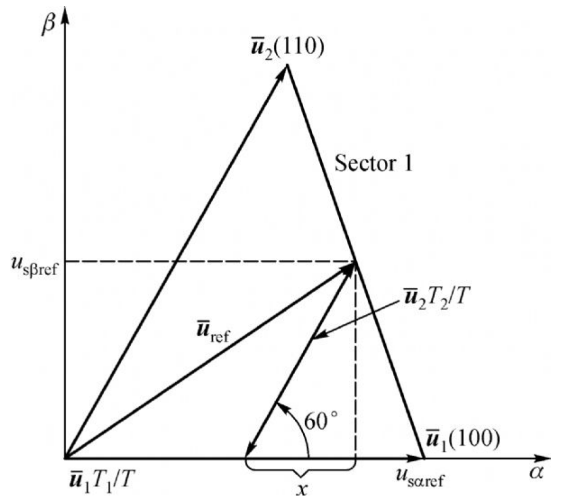
The d and q-axis voltage references, thus generated, are sent to pulse width modulator to generate gate pulses for the switches of power electronic converter.

4.3 PWM Generation

In order to achieve the optimum torque under the entire operating region, and the minimization of the current ripple and the total losses, full utilization of the DC link is required. In order to achieve all three aims, a customized modulation scheme needs to be developed for different regions. The unified modulation scheme is based on the same concepts as the space vector modulation scheme [17] - [18]. There is no power flow within the converters when the voltage difference between the different arm legs of the power electronic switches is zero. This time duration is called the 'effective time' and they are utilized to create the same average voltage and at the same time reduce the current with reduction in harmonics. In the space vector modulation scheme (SVPWM), the possible space vector area of the modulation scheme is divided into six sectors as shown in Figure 6a. After the calculated voltage reference, the sector is identified. The voltage vector is then calculated using the eight fundamental vectors using the effective time relocation algorithm.



(a) Space Vector Distribution [19]



(b) Fundamental Sectors [20]

Figure 6: Space Vector Modulation

However, such complicated procedure of determination of sectors and timing calculation leads to complex algorithms for implementation in the micro-processors. Thus, a new scheme called the "unified modulation scheme" has been investigated which mainly focuses on the effective timing reallocation by introducing the concept of imaginary time. Since, the output voltage of the converter is mainly decided by the effective time, negative time can exist virtually [17]. This value is directly related to the phase voltage reference resulted from the closed loop. Before the implementation of the gating signals, a time displacement operation will be applied to the imaginary times to create an offset. Then the offset is added to all the gating signals to create a symmetric switching pattern. The unified modulation scheme can be adapted to normal SPWM and the SVPWM with little variations. Following are the mathematical equations for the entire algorithm. The terminologies are described in the first chapter.

$$\frac{V_{as}}{V_{dc}} = \frac{T_{as}}{T_s} \tag{54}$$

where V_{as} is the resultant phase A voltage, T_{as} is the ON time of the periodic modulation and T_s is sample time.

Since, the voltage reference V_{as} can be negative as a result of the calculation of the control loops in the vector control system, the switching time in the equation 54 can be negative. Thus, a new term 'effective time' can be defined:

$$T_{eff} = T_{as} - T_{bs} = \frac{T_s}{V_{dc}} * V_{as} - V_{bs} \tag{55}$$

However, to avoid negative numbers in the micro-processors, the actual gating signals can be defined by introducing an offset as per in the following equations.

$$T_{ga} = T_{as} + T_{offset}; \qquad T_{gb} = T_{bs} + T_{offset}; \qquad T_{gc} = T_{cs} + T_{offset} \tag{56}$$

After some mathematical manipulation, the imaginary switching times can be defined using the reference voltage can be defined as :

$$T_{as} = \frac{T_s}{V_{dc}} * V_{as}; \qquad T_{bs} = \frac{T_s}{V_{dc}} * V_{bs}; \qquad T_{cs} = \frac{T_s}{V_{dc}} * V_{cs} \tag{57}$$

Since in a balanced system without a common point connection, the voltage references should add up to zero, the effective time of the switching times can be identified between the extremities of the negative time values as calculated before. The actual switching time can be obtained by finding the difference between the time period and the actual gating times as the following:

$$T_{ga*} = T_s - T_{ga}; \qquad T_{gb*} = T_s - T_{gb}; \qquad T_{gc*} = T_s - T_{gc} \tag{58}$$

The value of the offset can be set for different kinds of modulation schemes. Thus, effectively playing with the zero voltage vectors both the SPWM and SVPWM technique can be implemented. For this project, SVPM is chosen as the final modulation scheme. In that case, the offset switching time value can be defined as the following:

$$T_{offset} = -T_{min} + \frac{T_o}{2} \tag{59}$$

The extensive mathematical proof for equation 59 is included in the appendix. The offset switching times are included in all the three phases and the resultant is applied for symmetric patterns. A flow chart describing the entire algorithm is shown in Appendix G.

4.4 Speed Measurement (Encoder Interface)

This section describes the theoretical basis of design for the encoder interface and how the speed estimate is determined. The speed estimation is required as the feedback input into the speed controller. Typically quadrature incremental encoders are used for speed measurement [21] and this is how the interface is designed for in the DogDrive controller.
Quadrature incremental encoders can be used to determine speed by counting the digital pulses emitted. As the shaft rotates, the encoder issues digital pulses on two channels (A and B) that are mounted 90 degrees to each other. The frequency with which the pulses occurs determines the speed and the order of the pulses between the channels determines the rotation direction. (see Figure 7).

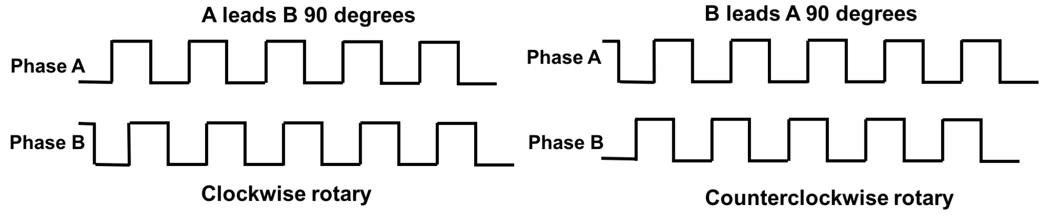


Figure 7: Discrimination of rotary direction

There are two self-evident methods that can be used for the speed estimation; the M-method and T-method. The M-method counts the number of encoder pulses in a given sample period. The T-method counts the number of clock samples between pulses. The M-method has diminished accuracy at low speeds when there are few encoder pulses within the sample period and the T-method has a similar issue at high speeds when there is a smaller number of clock cycles between pulses.

Reference [22] discusses the combined method to address these limitations called the M/T method. The principle is shown in Figure 22, where T_c is the sampling time from the timer, calculated precisely by counting the pulses from the clock. This makes $T_c = M_2/f_0$ where f_0 is the frequency of the clock from the BW. If we let Z be the number of pulses from the incremental encoder in one revolution and M_1 be the number of pulses from the incremental encoder in T_c, the M/T speed calculation is given by [23]:

$$n = \frac{60M_1}{ZT_c} = \frac{60M_1 f_0}{ZM_2} \tag{60}$$

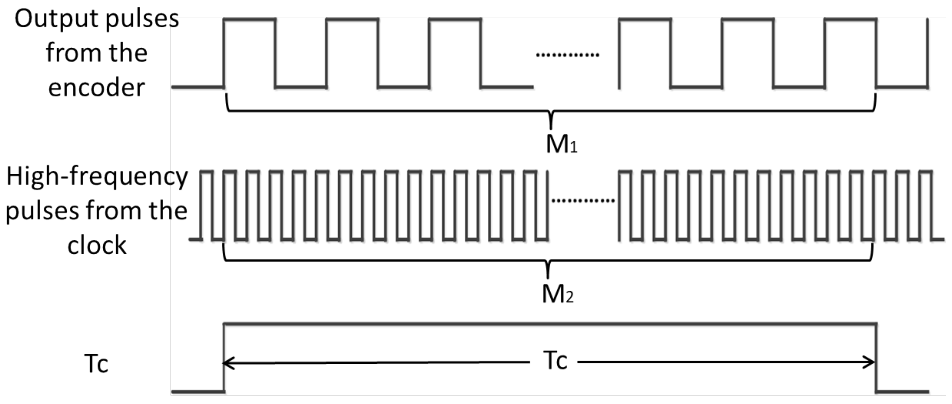

Output pulses from the encoder
M1
High-frequecy pulses from the clock
M2
Tc
Tc

Figure 8: M/T method for speed measurement

The M/T method is the approach adopted for the DogDrive controller. Further, since the BW has external Digilent PMOD ports an interface using a PMOD encoder such as that in [24] will be assumed for the controller interface design.

4.5 Sensor Interface

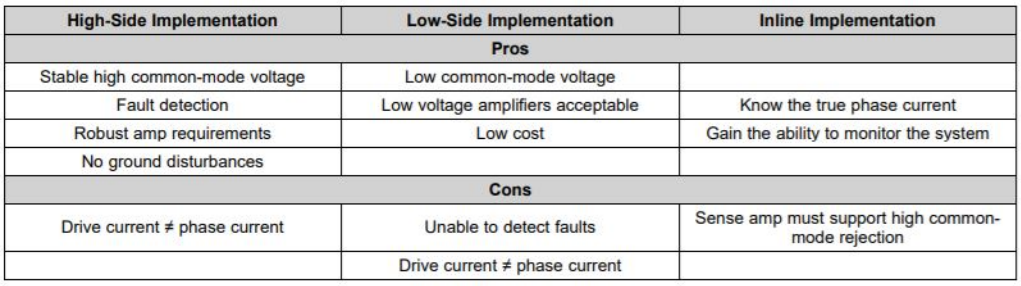
To ensure an appropriate sensor interface is implemented in the controller, the type of sensors need to be provisionally specified. This section will discuss the proposed sensor types with brief reasoning used and the method selected.

Phase current sensors are required for:

1. Fault protection.
2. Input for control algorithm.

There are three ways to implement a current sensor; high side measurements, low side measurement and inline measurements. The "side" refers to the location of the current sensor in relation to the gate driving FETs and is discussed in greater detail in section

The table shown in Figure 9 presents a summary of the different implementation methods. Due to its lower cost, the Low-side implementation is selected as the intended interface for the DogDrive controller.



High-Side Implementation	Low-Side Implementation	Inline Implementation
Pros		
Stable high common-mode voltage	Low common-mode voltage	
Fault detection	Low voltage amplifiers acceptable	Know the true phase current
Robust amp requirements	Low cost	Gain the ability to monitor the system
No ground disturbances		
Cons		
Drive current ≠ phase current	Unable to detect faults	Sense amp must support high common-mode rejection
	Drive current ≠ phase current	

Figure 9: Comparison of Implementation methods from [25]

It is also noted that current sensors are not necessarily required on each phase leg as the third phase may be inferred from the other two legs using Kirchoff's Current Law. However, for the proof-of-concept controller we will assume the use of three phase sensing for computation ease and the ability to detect circulating currents, if needed.

The proposed type of current sensor is a Magnetoresistive type, since it is available in wide measurement ranges and has 100 times the sensitivity of an hall effect sensor [26].

There are variety of voltage sensors based on shunt sensing schemes available in the market. In order to sample the voltage ripples at the DC capacitor at a high resolution that matches the sampling rate of the processors, the ADS8688 from Texas Instruments was selected. Due to availability of multiple channels with simultaneous acquisition of analog measurements, it facilitates a great deal of flexibility at a high rate of 500kSPS (kilo samples per second).

5 Method

This section describes the approach taken with regards to implementing the theory described in Section 4 into the system described in Section 3. This section is in part intended as a manual to the software created. The implementation can be split broadly into three categories: the DSP, FPGA and the interface between them. These will be described sequentially. Figure 21 in Appendix E presents a detailed depiction of the software architecture implemented in the project.

As is discussed, not all of the controller design from Section 3 was implemented and integrated. The HMI, sensor interface, encoder interface and protection function have only partially been implemented. Progress on these sections was halted following the limitations that will be discussed in Section 6. However, the extent of implementation on these software blocks will still be explained for potential re-use by future groups.

5.1 DSP

As shown in Figure 2, the DSP is responsible for the HMI, Vector Control, Induction Motor Model and PWM Generation. Of these functions, the vector control module, induction motor model and PWM generation have all been implemented in full. The algorithms behind these functions are described in Sections 4.1, 4.2 and 4.3. A web-server HMI has been created to provide an example of possible future applications, but does not have full functionality and serves only as a feature demo.

5.1.1 Induction Motor Model, Vector Control, PWM Generation

These three function blocks can be considered as closely related components, as such they are not considered separately for discussion. The software is written in C and, as of now, must be initiated manually on the BBB. The program is started by running the target file currently named: *build*. The source code is separated as follows and is referenced in Appendix B:

- IRQ Handler: controls management of interrupts originating from the FPGA
- IM-Model: models an induction motor, used primarily for testing the algorithm
- RG-Controller: performs reference generator functions
- DQ-Transformation: performs the necessary DQ transformation
- PI-Controller: main control loop functionality
- IDQ-Transformation: performs the necessary inverse DQ transformation
- SV-PWM: performs the UPWM scheme calculations

The DSP initializes an induction motor based on nameplate data that is hard-coded into the program. This data is shown and explained in Appendix A. Internally, the algorithm converts all physical parameters to a per-unit system, thereby providing scalability for the program. All further calculations are based on said converted data, therefore our solution should be applicable for induction motors of various ratings.

The primary principle of operation is the following. The algorithm is running in standby mode waiting for an appropriate interrupt on a suitably configured GPIO pin. When the program detects the interrupt signal, arriving at a fixed frequency, it initiates the motor control algorithm in a sequential order, as per the underlying theory, where the output of the control loop is then forwarded to the PWM generator function. This single interrupt, occurring once per switching period, is the only governing control signal in the entire program. Its occurrence indicates that sensor information from the FPGA is ready to be received and the DSP is now running on a virtual time limit as determined by the switching frequency. This time limit acts as a hard stop, meaning that by the time it is triggered, PWM generation information must be available from the DSP, thus it will have to have finished running the control loop algorithm and completed all necessary calculations for the consequent switching period. Utilizing only the current and speed measurements as input, as well as the voltage level of the DC link, the final output of the DSP algorithm will be two numbers per IGBT switch indicating the ideal switching time instances of the PWM signal. This output, as well as a PWM data available indicator is then communicated via the common GPMC interface to the FPGA.

5.1.2 HMI

A demonstration HMI has been created on the BBB using the Apache webserver that ships with Debian Stretch. When accessing the IP address of the device through a web browser, it will respond with the HTML interface page. This has been created using HTML, CSS, Javascript and Plotly.js. A screenshot is shown in Appendix C and the files can be found in the GitHub repository DogDrive-UI repository.

Currently, the portal only brings up a pre-saved plot of data extracted from the induction motor model. However, this demonstrates the capability using a webserver based approach that may be extended to real-time monitoring.

5.2 Interface

The BeagleWire device was designed to communicate to the BBB over the GPMC which is an embedded section of the AM335x processor. A number of files were provided as part of the BeagleWire software repository (BeagleWire) for this purpose, however, they needed significant reverse engineering and modification to allow them to function as part of the DogDrive. This section describes how they have been utilised and modified in the controller.

The intent of the GPMC is to allow an external memory device to be attached to the BBB and be addressed from the BBB as if it were a part of the BBB memory. By setting up the expected external memory in the FPGA and configuring an appropriate GPMC handler, it allows very close integration between the two devices.

There is another interface between the BW and BBB, the Serial Peripherary Interface (SPI). This is used to configure the BW from the BBB. However, this was fully configured out of the box and will not be discussed further.

5.2.1 GPMC

The GPMC utilises the GPIO pins on the BBB to interface to the BW. This is a 16 bit wide data / address multiplexed bus (each bit represented by GPIO). Figure 10 demonstrates the timing for a single read access (A[27:17] bits are not used). The address is sent first and then subsequently the data is sent using the same pin in/outs.

5.2.2 DSP GPMC Interface

The GPMC is configured in the BBB by applying a Device Tree Overlay (DTO). The DTO is a file which describes the structure of an external device to the Linux kernel. It creates the device in the kernel so that a user space application may interact with it. In the case of the GPMC, since it is integral to the kernel it only requires configuration and does not need a

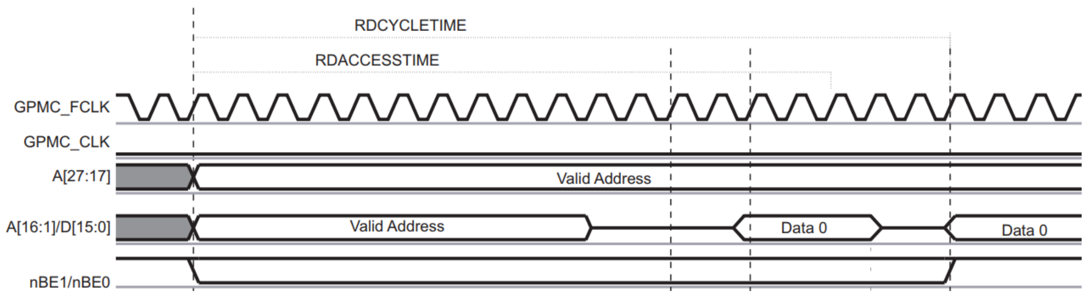


Figure 10: GPMC Read Access Timing Diagram from [28]

separate driver to be created. The interaction from the user space is directly to the memory locations so there is no need for an API.

The DTO is created from a Device Tree Structure (DTS) file before being applied. The DogDrive uses the DTS from the BeagleWire software package, *BW-ICE40Cape-00A0.dts*, with some minor amendments (DTS File). This DTS first sets the pins that the GPMC will use in the 0 fragment. The pins need to be in the correct mode, Mode 0, to be used for GPMC communication. In fragment 2, the actual GPMC is configured. The pingroup previously defined is assigned and the GPMC configuration is set-up using defined Linux variables (see Section 5.2.1). From a usability perspective, the key parameter in this configuration is the register address, set to 0x010000000, this is the start of the "mounted" GPMC memory in the BBB. This overlay then needs to be mounted, the process for doing this is described in Appendix I.

With the GPMC configured and overlaid on the BBB, it is interacted with as if it were an extension of the BBB memory (starting at the defined memory address). There is a lightweight bridge file provided as part of the BW software package, which is used in the DogDrive to simplify this memory interface (*bw_bridge.c* and *bw_bridge.h*). It contains the two user functions *set_fpga_mem* and *get_fpga_mem* that are used in the DSP application to interface to the FPGA, see Figure 21.

This bridge file itself is relatively minimal. It maps (using *mmap*) the memory address (must be the same as that in the overlay) and the memory size into a bridge structure, essentially a collection of pointers to the GPMC element in memory. It opens the memory directly from */dev/mem/* and then manages the function calls to set the memory address to the intended value.

5.2.3 FPGA GPMC Interface

The FPGA is set-up as the external memory device that the GPMC is expecting. That is, it:

- Allocates an appropriately sized memory storage
- Responds to a read command from the GPMC with an address by supplying requested data
- Responds to a write command from the GPMC by writing the supplied data into the given address location
- Executes at the required timing rate with required GPMC bit response

In a similar manner to the DSP, the memory register set-up in the FPGA matches that, which is expected, in terms of data width (16 bit) and size (5 bit). This is defined in the main file described in Section 5.3. The memory is interfaced to the FPGA using the variables *addr* (reg - address pointer), *data_in* (wire - DSP read from FPGA) and *data_out* (reg - DSP write to FPGA). However, the interaction of these variables is set within the separate interface component, *gpmc-sync.v*.

Only 13 of theses 16-bit memory addresses are used by the DogDrive. These registers are described in the table in Appendix F. The first word address holds the configuration bits. The functionality of the addresses are discussed in Section 5.3. All of the switch on and off times reference the upper switching value for each phase.

The main FPGA GPMC component the DogDrive uses, *gpmc-sync.v*, is another driver file supplied with the BW packaged software files. When interpreting this file, it must be remembered that the FPGA is behaving like a slave memory device to the DSP. While the FPGA can read and write to its own memory, it cannot instruct the DSP to read or write to this memory (without interrupt, see Section 5.2.4). A high level description of this sequence is as follows:

- When the address is not valid (from GPMC discrete pins), latch the pin states as an address. This becomes the *addr* value and remains until a new address is sent.
- Always at the positive clock edge, the data currently in the address location specified will be written to a hold variable *gpmc_data_out*.
- If the GPMC is requesting a read through the discrete pin, this hold variable will be written through the package pins bi-directional tri-state buffer (*SB_IO* block) to the data bus.
- If the GPMC is requesting a write to memory, the pin data will be transferred to the hold register *gpmc_data_in*. At the negative clock edge, the data in this register will be set into the *write* variable, which is assigned to the *data_out* variable from *top.v*. If there is no write to memory, this will just be the data that is currently in the memory address.

In addition to the above, all the signals are passed through a dual flip-flop synchronizer to add robustness to the signals. However, this has been omitted from the above description for clarity.

5.2.4 Interrupt Routine

As described in Section 5.2.4, to the DSP the FPGA acts as an external memory device which is controlled via the GPMC. However, as described in Section 5.3, the FPGA is also in control of the timing and sequencing. The FPGA needs to be able to have sequencing control over the DSP, which cannot be done over the slave GPMC interface. This requires an independent interrupt, which is completed over an unassigned GPIO pin between the BW cape and BBB. In the DogDrive, this is Pin 17 (GPIO 49 - P9-23).

To configure this, the DTS discussed in Section 5.2.2 also includes configuration of this pin to the correct state (MODE 7 and input) and assigns it to the correct *bw_irq_pins* pin group. This allows the pin to be “uncovered” as an input for the BW to drive. With the pin uncovered, the data in the memory address for the value of the GPIO pin in the BBB follows the physical value of the pin. This is the interrupt signal.

The interrupt signal is driven by the BW according to the master counter in the FPGA (see Section 5.3). It triggers at the very beginning of the switching frequency and is held for a period of 500ns to ensure it is read by the DSP. Note that it is not a strict requirement that the interrupt occurs at the start of the period, it could occur slightly earlier to increase available DSP processing time, however, this was considered the cleanest implementation.

In the DSP, the interrupt handler is incorporated in the main routine. It is not a true interrupt handler in the sense that it does not interrupt the routine, rather it waits in an infinite while loop for a rising edge detection on the IRQ pin. The pin value is read by directly mapping the memory location of the pin's value to a readable variable. This allows a much higher sensing rate than alternative kernel GPIO driver implementations. When the DSP detects the rising edge interrupt, it initiates the vector control functions detailed in 5.1.

5.3 FPGA

As shown in Figure 2, the FPGA's primary functions are as a state machine, sensor interface, protection function and encoder interface. Of these functions only the state machine (including the PWM output) was entirely implemented. This is the key module as it is the critical to demonstrating whether the DogDrive can meet the timing requirements of a drive controller. The sensor interface is partially complete but not yet integrated whereas the encoder interface is missing implementation. This is primarily due to the timing issues discovered and discussed in Section 6.

The protection function blocks were not implemented, nor designed further than that discussed in Section 3 and will not be discussed further (see Section 7.2).

Figure 21 shows the breakdown of the FPGA code, the three source files are listed and explained below. The source files are written in Verilog and are compiled using the IceStrom toolchain (further description of this can be found in [5]):

- *top.v* - Main file including state machine and PWM outputs
- *gpmc_sync.v* - Interface function discussed in 5.2.3
- *sensor.v* - Reads data from the external sensors and stores in the shared memory when requested by controller state.

5.3.1 State Machine

The state machine implemented in the FPGA executes the following tasks according to it the controller state:

- **DSP control** - it initiates the control calculation and monitors success.
- **PWM control** - directly operates pin outs according to the switch on and off times received from the DSP.
- **Sensor Timing** - reads the sensor data from the external sensor interface module at the required times and stores for DSP read access.

Sequence states are determined solely by timing based on the master counter (*counter*), which increments every clock cycle and resets at the end of the switching period. It is from this counter that all of the DogDrive tasks are sequenced. This task sequencing is depicted in Figure 11 .

DSP Control The DSP is controlled by the IRQ pin discussed in Section 5.2.4. This determines when the DSP starts processing. As shown in Figure 11, this is initiated at time 0 in the period. The DSP will then execute and return data to the FPGA GPMC memory at some point in the period.

Since the FPGA has no control over when the DSP will return the data, before commencing with the next period a check is completed to ensure that valid data has been received. This check is completed using the *data_valid* single bit register which acts similar to a heartbeart message. Each period the DSP must increment the data valid bit in the FPGA GPMC memory (from 0 to 1 or 1 to 0). At 500ns before the end of the period, the data valid bit is checked

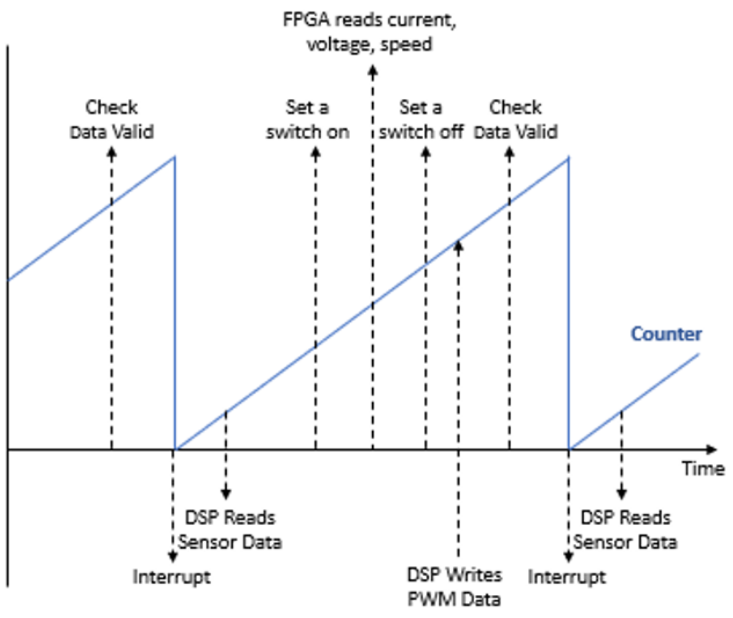


Figure 11: State Machine timing and tasks

by the FPGA to determine if it has changed. If not, the data from the DSP will be assumed invalid, a zero vector will be switched on the PWM outputs and the PWM will be disabled.

PWM Control PWM control is managed through the switch times written by the DSP to the GPMC memory on completion of the control calculation. The FPGA requires these values prior to the end of the switching frequency to allow addition of a deadband and ensure the minimum pulse width between the requested times. These values are a function of the physical requirements of the gate drivers and switches.

Once the actual switch times are determined, they are stored in the *sw_on_next* and *sw_off_next* registers. The current switch registers are set to the next registers at the end of the period.

The outputs use the PMOD1 port on the BW. Each phase is written to a different output pin (i.e. phase a to pin 1, b to 2 and c to 3). The outputs are set by comparing the master counter to the current switch register values. In a completed drive these ports would interface directly the IGBT gate driver modules.

Sensor Timing Although the sensor timing is shown on Figure 11, it was not implemented and will not be discussed further.

5.3.2 Sensor Interface

The sensor interface is yet to be integrated into the overall controller due to the lack of sensors available. However, the current sensor and voltage sensor blocks have been implemented using a simulation tool mimicking the parameters of their respective real-world counterparts, and is detailed in the section below.

Current and Voltage Sensor Interface It is not uncommon that commercially available sensors have a signal conditioning system that converts the measurement value to equivalent analog voltage ranges. Since the BW does not have an inbuilt ADC support, an external ADC can be implemented to convert the analog voltage ranges to digital pulses from which the corresponding phase currents can be computed based on the current sensor's conversion ratio. The analog voltage measurement from the sensors are sent to the ADC which converts it into corresponding digital pulses for the FPGA. This part of the implementation has been simulated and the timing diagram has been verified.

The 12 bits provide sufficient resolution for the measurements, and the 4 channels can incorporate the current measurements on three phases as well as the DC-link voltage. Due to the limited availability of PMOD ports (the BW has 4), this is the preferred solution.

The challenge in implementing an external sensor comes from the fact that the sensor operates at a different clock frequency than that of the external FPGA clock. Therefore, data sensing has to match both the system clock and the ADC clock. The ADC interface operates at 1 MHz and the data is read serially on all 4 channels, which includes the three current sensor and a voltage sensor data.

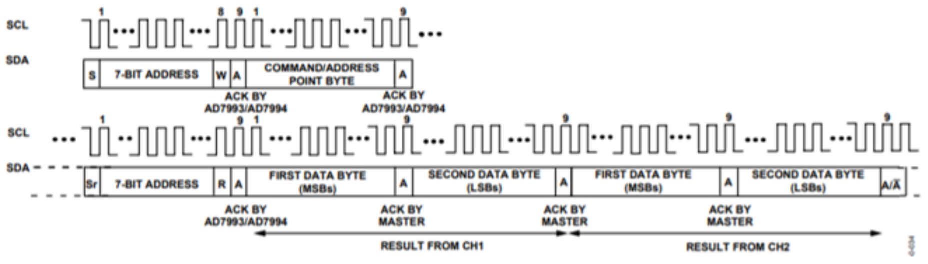


Figure 12: ADC Timing Diagram from the AD7993/AD7994 datasheet [27]

The ADC interface is a state machine that consists of 27 states which are executed in a loop to read all the four channels. This follows the serial I2C protocol. The address bit 0010 0000 is first sent, followed by the write bit and acknowledgment. The write command is sent to select the particular mode and channel, followed by acknowledgement. The data reading process begins with a start bit followed by sending the address again along with the read bit. The data in all four channels are read as two 8-bit data separated by acknowledgements.The reading process stops with a stop bit where the data line is pulled to high when the clock is high. Once the data is read, the corresponding actual value is computed by using the conversion ratio of the sensor. The value is then stored in a register which can be read by the FPGA controller algorithm.

The code for ADC interface is tested on EDAPayground tool using a testbench code and the timing diagram is shown in Figure 13, demonstrating the correct timing along with the different states of the state machine.

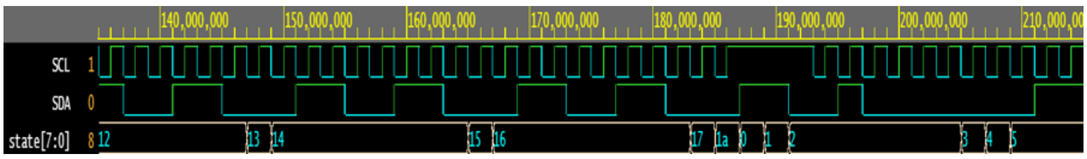


Figure 13: ADC Interface Simulated Timing Diagram

6 Application

The project, if considered successful, could have great implications on the future market of electrical motor drives. As existing drives are typically expensive and complex devices with proprietary software and hardware, a relatively cheap and open-source solution could provide a feasible alternative in the long run. Undoubtedly, there are numerous improvements that need to be made in the software, and rigorous testing needs to be done in order for such a solution to be commercially viable, however as a proof-of-concept work, it has its merits in proving that it is a concept worthwhile to investigate in the future. The system is highly compact by design. Ideally, the user would need to configure the system parameters based on the motor and sensors used and the rest is handled by the DogDrive.

The project is currently in its testing phase, however some significant obstacles have been encountered. Potential solutions have been investigated, however at this point it is not yet possible to evaluate the validity of the proposed solution. That being said, there are many conclusions and results available, and the basic functionality is completed. Goals that have been met, functions that are currently being tested and tools that have been postponed for future development are listed in the following sections. All actions marked as *SUCCESSFUL* have been fully implemented and tested, and those that are marked with *PENDING* have been implemented externally to the main functions but are missing integration and testing or otherwise need further development before testing can begin.

6.1 Primary Functions

This section considers all the primary goals originally considered in the project plan.

- Implement a functioning interface between DSP and FPGA – **SUCCESSFUL**

  Communication between the DSP and FPGA has been successfully implemented and tested. The GPMC functionality that is set up to provide a communication link between the DSP and FPGA has been mapped and the user-space functions necessary to control it have been identified.

  As part of this communication, the interrupt handler between the DSP and FPGA has also been implemented and tested to be working. The original implementation consisted driving the GPIO pin directly from the embedded Linux kernel, however this resulted in a significant performance reduction on the BeagleBone, due to the Linux kernel handling the edge detection. This involved calling the kernel module on every interrupt happening at switching frequency which greatly limited the system performance. This resulted in a hard limit of 4kHz for the drive switching frequency, as going above that caused the Linux kernel to crash. This value is typically a minimum available setting in existing drive control applications. Furthermore, the performance hit suffered by the BeagleBone was assumed to conflict with the motor control algorithm, potentially causing problems regarding the strict timing constraints. As such, it was deemed necessary to find an alternate solution to the problem. Significant improvement was achieved after successfully mapping the GPIO pin to an internal memory segment of the BeagleBone, thus instead of invoking the kernel module at every interrupt, a simple memory read was all that was necessary to trigger the DSP interrupt. This has increased the speed at which the GPIO pin can be handled by approximately a thousandfold, which more than exceeds typical motor applications. Thus in conclusion, a highly optimized and efficient solution was found to accommodate the interrupt signal.

- Implement vector control algorithms in DSP – **SUCCESSFUL**

The vector control algorithms detailed in section 4 have all been implemented on the DSP. Their evaluation is completed to ensure sufficient accuracy and speed, see section 6.3. In order to do this, an induction machine model has been implemented both on the BeagleBone, as well as in Simulink. The concept is, that by providing the same operating point for both models, the output of the vector control algorithm and the Simulink model can be directly compared. Therefore, the implementation on the DSP can be tested for both functionality and the actual performance of the control loop.

- Implement PWM interface on FPGA – **SUCCESSFUL**

  The PWM module is a fairly simple segment of the FPGA code. Two versions were considered and tested. First, the input parameters considered were the period length and duty cycle, however these were later modified to move the PWM signal formatting to the DSP and consider only the ideal switching times as this would later simplify adopting the FPGA code for various protection algorithms. The final version has been tested and the configurable PWM signal driven through a PMOD output of the FPGA has been measured using an oscilloscope.

- Implement external sensor interface on the FPGA – **PENDING**

  This feature has yet to be fully integrated to the existing source code. Although current sensor reading is tested for the FPGA, it has yet to be merged with the rest of the source code. Implementation of the encoder interface has not been initiated. It was deemed to be of lower priority as multiple higher level features are still in testing phase and this segment is only applicable once the drive control is advancing to a lab testing phase on an actual motor.

- Implement protection functions on the FPGA – **PENDING**

  This feature is a fairly small extension of the main FPGA code, however the implications are highly important. These protection functions include over-voltage and current control, the PWM dead-band control and checking for minimum pulse width based on the physical properties of the IGBT controllers. As such, rigorous testing criteria must be met, and a sufficiently detailed test procedure, to take into account all possible scenarios where the protection functions are critical, is currently being developed. In addition, gate drivers are being investigated as some variants offer some of these protection functions in-built, which would mean that additional integration and testing on the FPGA is unnecessary.

6.2 Secondary Functions

This section details progress on the secondary goals originally considered in the project plan.

- Implement a functioning real-time HMI – **PENDING**

  Due to time limitations, this functionality has only been planned and a simple demo version has been prepared to showcase the possibilites. The proposal is to implement a web-server, as the BeagleBone provides ample solutions to accommodate this method, where the user can view the DogDrive input and output parameters in real-time and configure the DogDrive before the start-up sequence.

- Integrated test of drive controller to physical sensors and gate drivers – **PENDING**

  Due to time constraints, the software was not migrated to a laboratory setup, however, thanks to the induction machine model implemented on the BBB, we were able to evaluate the performance of the drive control algorithm without a physical setup. The results are detailed and analyzed in the following section. To summarize, while running the algorithm at lower than switching speeds for the purpose of testing, the functionality is effective and provides the expected outcome. However, when increasing the speed to typical switching frequencies, issues arise with the performance of the BBB. For this reason, integrated testing was put on hold, until hard real-time performance enhancing methods are explored and evaluated for the BBB Linux system.

6.3 Results

In this section, the results of the performance of the vector control of the emulated induction motor and the integration of the DSP and FPGA are presented.

6.3.1 Performance of the Vector Control of the Induction Motor

In this work, start-up of an induction motor at full load torque is simulated. The data for the applied stator voltage to the emulated induction motor, and the stator current, rotor flux, torque and speed of the induction motor are saved in csv format and analyzed in MATLAB. The corresponding quantities are plotted in Figures 14 and 15 respectively. As seen in Figures 14 and 15, rotor flux, developed electromagnetic torque and the mechanical rotor speed are converging to the nominal p.u. values in the Table 2 as expected. During the start-up of the induction motor, rotor speed is going to negative due to the application of the load torque at the starting.

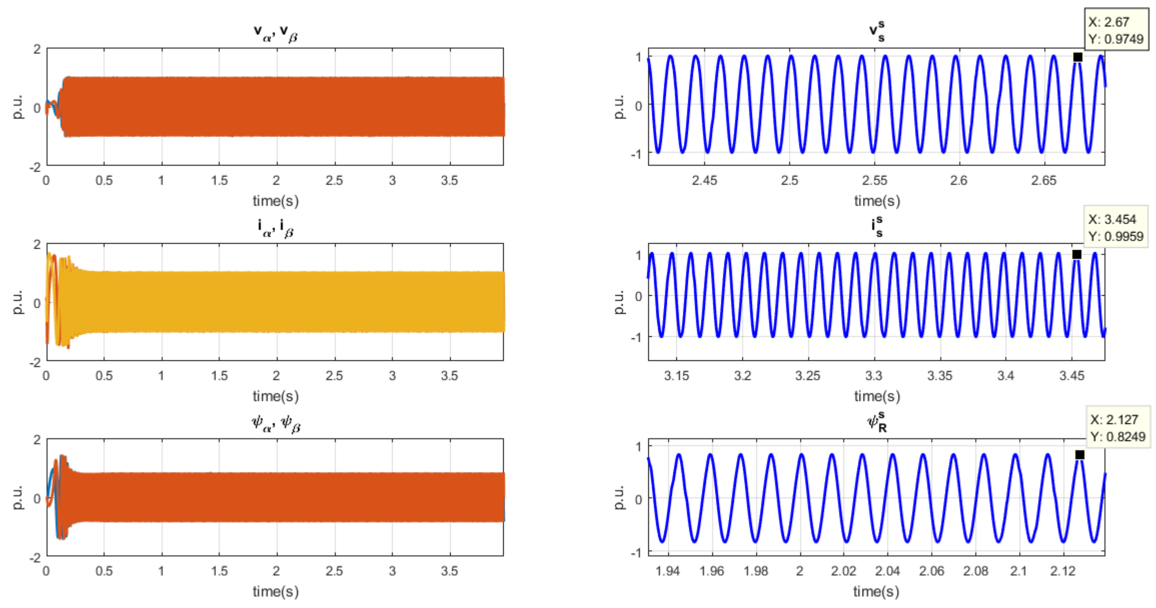


Figure 14: Stator voltage, stator current and rotor flux of the emulated induction motor

6.3.2 Integration of DSP and FPGA

The following sub-section shows the results of the integration of the DSP and FPGA, in particular the interrupt handling and the consecutive PWM generation driven by the DSP and output by the FPGA. In the following figures, the yellow signal indicates the 10kHz interrupt originating from the FPGA. In these figures we can follow the basic operating principle of the motor drive by considering that the time between two interrupts is one PWM cycle. The DSP is generating the switching times of the PWM output on the FPGA PMOD interface and

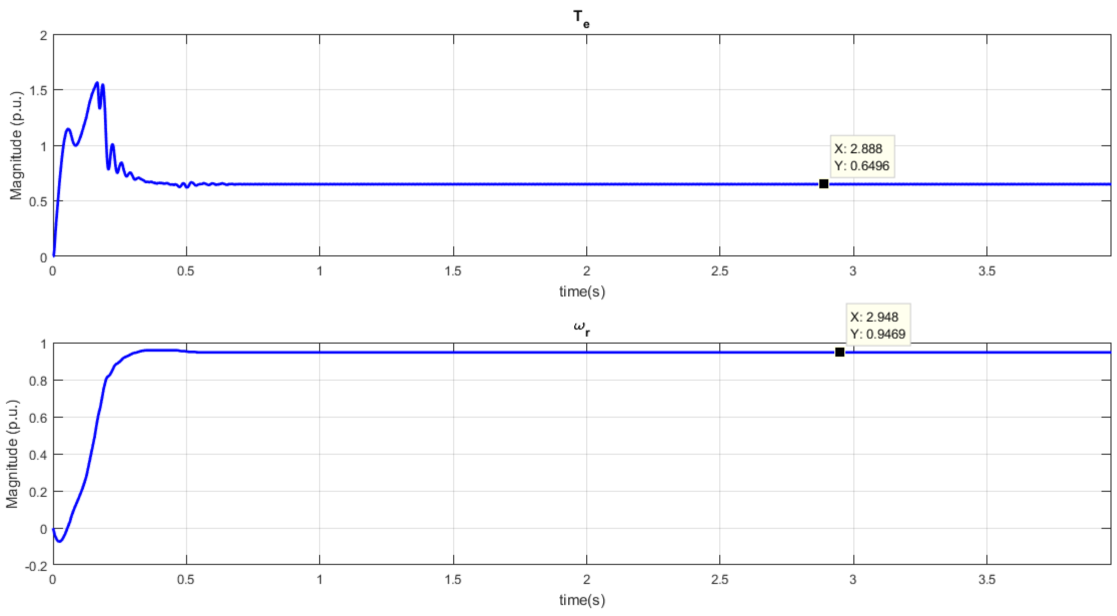


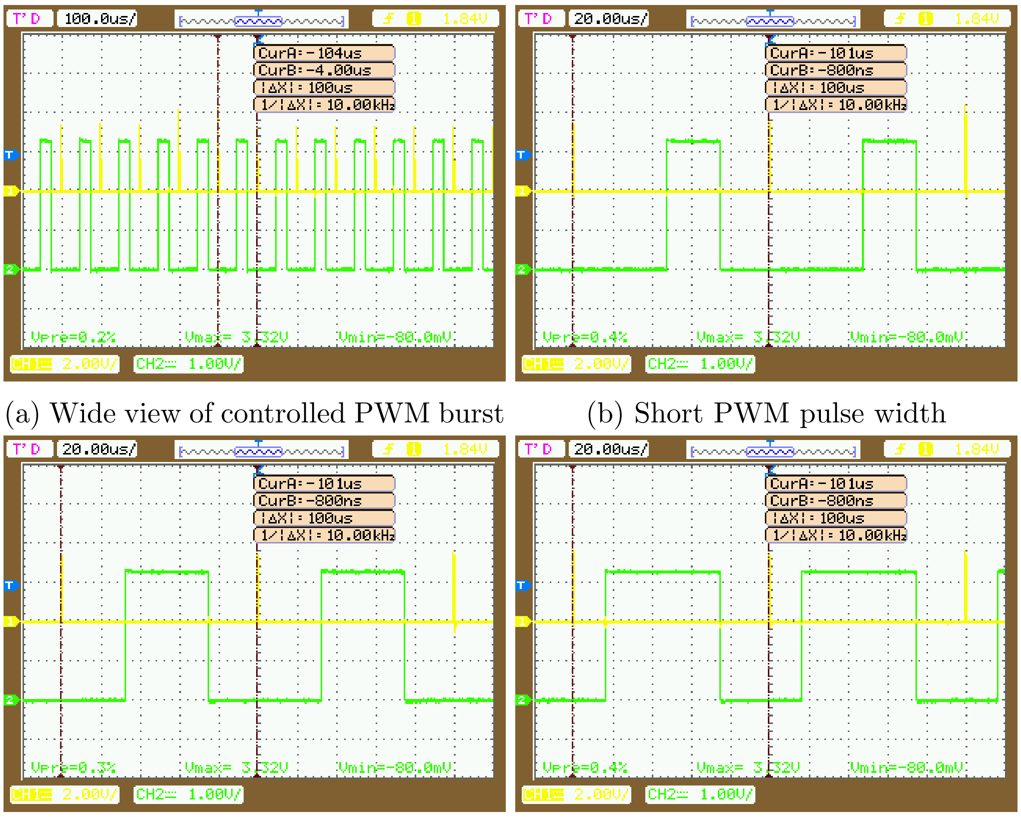
Figure 15: Torque and speed response of the emulated induction motor

communicating them towards the FPGA via the GPMC bus. This behavior has been tested by alternating the PWM length at every interrupt cycle, gradually increasing the PWM pulse width. Figure 16a depicts the PWM generation and interrupt sequence over a longer period of time, whereas figures 16b, 16c and 16d show increasing pulse widths. These were generated by a continuously running user-space program on the DSP, and the change in PWM pulse width can be observed in real-time.

While the oscilloscope measurements are meant to showcase the functionality of the program, the performance of the control loop algorithm was also evaluated. In order to accomplish this, the DSP is measuring the number of clock cycles it takes to complete all the control loop calculations within the program itself. The DSP was running for approximately one and a half minutes, which as can be seen is enough to provide highly indicative results. This is deemed a sufficient solution to give an overview of the execution time of the control algorithm. The measurements are converted to appropriately scaled time values that measure how long the processor was performing the control algorithm. The data was exported and plotted as can be seen on Figure 17.

The y-axis indicates the algorithm execution time in microseconds (50-500) whereas the x-axis is the program runtime in seconds (0-100). The horizontal, red lines indicate various switching frequencies, starting from 4kHz with the uppermost line, 8kHz for the middle and 16kHz for the lowermost line. As apparent from the figure, the algorithm fails to perform the intended calculations within the strict timing constraints. This can be divided into two primary issues.

First, at lower switching frequencies, due to the fact that the embedded Linux running on the BBB is inherently not suitable as a hard real-time operating system, various system calls and interrupts that are higher priority than the running program are interfering, and occasionally disrupting the program for a long enough time, that the timing criteria, as preordained by the FPGA, is not met. Fortunately, numerous articles can be found that involve modifying the Linux kernel to lower the latency and thus help resolve the issue by removing or at least lowering the value of the spikes seen in the execution time. This involves further investigation, but could potentially improve the system performance enough to comfortably accommodate up to 8kHz switching frequencies.



(a) Wide view of controlled PWM burst (b) Short PWM pulse width

(c) Increasing PWM pulse width (d) Wide PWM Pulse Width

Figure 16: Demonstration of changing pulse widths

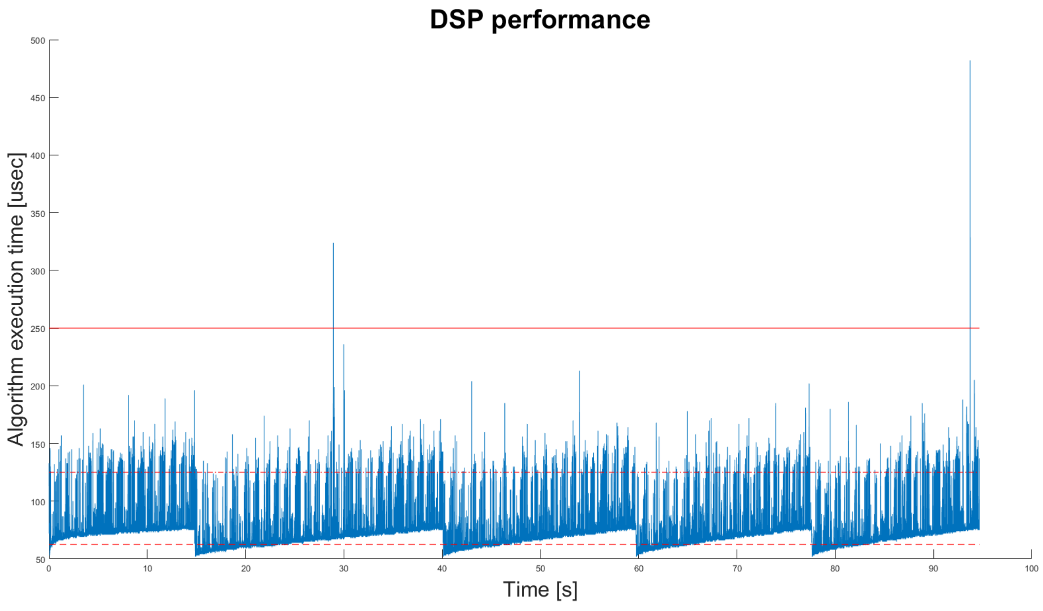


Figure 17: Wide PWM Pulse Width

Second, we can see that the base speed of the algorithm is well above the 16kHz marker. This has to do with the algorithm implementation itself. Currently, although it is by no means an optimized algorithm, it is difficult to say whether the current performance of the BBB processor is sufficient to perform the control loop functions under 62.5 microseconds, which corresponds to the 16kHz switching frequency.

7 Closure

The project is sufficiently tested to provide an evaluation in terms of the course and it is clear what further steps need to be taken with the development.

7.1 Conclusions

- A mathematical model of the Induction machine has been implemented in MATLAB/Simulink. The model is successfully tested by comparing the output characteristics to the theoretical calculations.
- The entire model of the Induction machine has been successfully implemented on the Beaglebone by using characteristic parameters based on the aforementioned simulations. Speed and Current control loops have also been successfully executed. The response time of the machine has been analyzed and deemed insufficient, improvements to the BBB kernel need to be investigated and implemented.
- After the voltage references are created as a result of the executed control loops, a Unified PWM algorithm has been developed over the Beaglebone to create timing signals for the FPGA. The FPGA, in turn, synthesizes the modulation signals for the gate drivers.
- An algorithm has been created to read sensor data at the FPGA interface. This data is then transferred to the DSP for the application of speed and current loops. Integration to the primary system is pending.
- A successful communication channel has been established with appropriate interrupt routines. This channel is used to transfer the timing signal data to the FPGA. The same channel is also used to transfer sensor data to the DSP and PWM regulation signals to the FPGA. Thus, a bidirectional channel is functioning.
- The modulating signals to the gate drivers are created at the FPGA by using the timing data transferred over the channel from the DSP.
- The current/voltage fault clearing functions are being developed over the FPGA for take appropriate measures during fault scenarios.

7.2 Ideas for Future Work

- Solutions for improving the Linux kernel to accommodate a closer to real-time solution for the algorithm execution needs to be investigated and implemented.
- Currently, Only the FPGA has been calibrated for reading the Sensor data using the Analog digital Converters. However, in order to create a complete system, Market Research about the available sensors will need to be done and eventually selected for our project. Consequently, The sensors shall be calibrated and the control loop finalized by fully integrating the sensor functions.
- A human machine interface (HMI) could be fully developed to monitor the DSP and FPGA system in real time. Given the success of the webserver demo application, this is the recommended pathway for this development.
- Once the software has been tested and found suitable in terms of function and performance, integration with an electrical motor in a laboratory setup is the final step to evaluate the project.

References

[1] https://roskill.com/market-report/lithium/

[2] http://www.ti.com/tool/BEAGLEBK

[3] Angelo D'Aversa, Bob Hughes, and Subhash Patel, Challenges and Solutions of Protecting Variable Speed Drive Motors, Schweitzer Engineering Laboratories, 6th Annual Conference for Protective Relay Engineers, College Station, Texas, 2013

[4] Mohan N. (2012). Electric Machines and Drives, Wiley, ISBN 9781118074817

[5] "BeagleWire" RPi Hub - eLinux.org. [online] Available at: https://elinux.org/BeagleBoard/BeagleWire [Accessed 7 Dec. 2018]

[6] Texas Instruments, Three-Phase Inverter Reference Design Using Gate Driver With Built-in Dead Time Insertion, 2017

[7] EJ2230 Control in Electrical Energy Conversion — KTH. [online] Available at: https://www.kth.se/student/kurser/kurs/EJ2230?l=en [Accessed 7 Jan. 2019].

[8] Mohan N. (2012). Electric Machines and Drives, Wiley, ISBN 9781118074817

[9] K. P. Kovacs, Transient Phenomena in Electrical Machines, Elsevier, Amsterdam, 1984.

[10] Web.mit.edu. (2019). Forward and Backward Euler Methods. [online] Available at: http://web.mit.edu/10.001/Web/Course Notes/Differential Equations Notes/node3.html [Accessed 6 Jan. 2019].

[11] L. Harnefors, "Design and analysis of general rotor-flux-oriented vector control systems," IEEE Trans. Ind. Electron., Vol. 48, No. 2, Apr. 2001, pp. 383–390.

[12] S.-H. Kim and S.-K. Sul, "Maximum torque control of an induction machine in the field weakening region," IEEE Trans. Ind. Appl., Vol. 31, No. 2, July/Aug. 1995, pp. 787–794.

[13] S.-H. Kim and S.-K. Sul, "Voltage control strategy for maximum torque operation of an induction machine in the field weakening region," IEEE Trans. Ind. Electron., Vol. 44, No. 2, Aug. 1997, pp. 512–518.

[14] K. J. Astrom and B. Wittenmark, Computer-Controlled Systems: Theory and Design, 2nd Ed., Prentice Hall, Englewood Cliffs, NJ, 1990.

[15] K. J. Astrom and B. Wittenmark, Adaptive Control, 2nd Ed., Addison-Wesley, Reading, MA, 1995.

[16] L. Harnefors, "Model-based current control of ac machine using the internal model control method", IEEE Trans. Ind. Appl., vol. 34, no. 1, pp. 133-141, 1998.

[17] D.W. Chung, S.K. Sul, J.S. Kim, "Unified PWM technique for real time power conversion", Power Conversion Conference, vol. 1, pp. 265-270, 1997

[18] Chung Dae-Woong, Kim Joohn-Sheok, Sul Seung-Ki, "Unified Voltage Modulation Technique for Real-Time Three-Phase Power Conversion", IEEE Trans. on Ind. Applicat., vol. 34, no. 2, pp. 374-380, March/April 1998.

[19] Santoshkumar M. Rangani, Hiren H. Patel, Control Scheme for Ultra Sparse Matrix Converter, Sarvajanik College of Engineering and Technology, 2015

[20] Benchabane, Fateh, A. Titaouine, O. Bennis, K. Yahia and Djamel Taibi. "Direct field oriented control scheme for space vector modulated AC / DC / AC converter fed induction motor." (2012)

[21] Hace, A. and Čurkovič, M. (2018). Accurate FPGA-Based Velocity Measurement with an Incremental Encoder by a Fast Generalized Divisionless MT-Type Algorithm. Sensors, 18(10), p.3250.

[22] Hace, A. and Curkovic, M. (2018). A Novel Divisionless MT-Type Velocity Estimation Algorithm for Efficient FPGA Implementation, IEEE Access, 6, pp.48074-48087.

[23] Boggarpu, N. (2019). Improved kinematic sensing for motion control applications. [online] Cora.ucc.ie. Available at: https://cora.ucc.ie/handle/10468/2145?show=full [Accessed 6 Jan. 2019].

[24] Reference.digilentinc.com. (2018). [online] Available at: https://store.digilentinc.com/pmod-enc-rotary-encoder/ [Accessed 10 Nov. 2018].

[25] "Current Sensing for Inline Motor-Control Applications^{TM} (2016) Texas Instruments - Application Report. [online] Available at: http://www.ti.com/lit/an/sboa172/sboa172.pdf [Accessed 26 Nov. 2018].

[26] "Magnetoresistive Sensors", Honeywell
https://sensing.honeywell.com/index.php?ci_id=50272la_id=1

[27] Analog Devices, Inc. AD7993/AD7994 4-Channel, 10- and 12-Bit ADCs with I2C-Compatible Interface in 16-Lead TSSOP Data Sheet

[28] "AM335x and AMIC110 Sitara^{TM} Processors Technical Reference Manual" (2011) Texas Instruments. [online] Available at: https://www.ti.com/lit/ug/spruh73p/spruh73p.pdf [Accessed 7 Dec. 2018].

A Determination of motor parameters and normalization

In order to determine the induction motor parameters, no-load test and blocked rotor test are performed in the laboratory. These estimated parameters are used to compute proportional and integral gains of the speed and current regulator respectively. There parameters are also used to emulate an motor model in DSP to verify the performance of implemented control algorithm. The ratings and the name plate data of the induction motor under consideration are as follows:

3-phase, **4** pole, **1.8** kW output power, **1400** rpm
Stator: Delta connected, **220** V (line to line), **7.8** A, the resistance of each winding is **2.3** Ω
Rotor: Y-connected, **180** V (line to line), resistance in each winding **0.67** Ω

From name plate data, following important nominal parameters are summarized as these will be used in normalization of the space vectors:

- Number of pole pairs, $n_p = 2$
- RMS phase voltage for Y connection, $V_N = \frac{220}{\sqrt{3}} = 127\ V$
- RMS phase current for Y connection, $I_N = 7.8\ A$
- Stator frequency, $f_N = 50\ Hz$

The experimental data from no-load test and blocked-rotor tests are analyzed in MATLAB to determine parameters R_s, R_R, L_M, and L_σ using the mathematical derivations in [7]. The estimated motor parameters are summarized in the Table 1:

Table 1: Estimated induction motor parameters from no-load and blocked rotor test

Parameters	**Estimated values**
Stator resistance, R_s	0.77 Ω
Transformed rotor resistance, R_R	1.03 Ω
Magnetizing inductance, L_M	66 mH
Leakage inductance, L_σ	9.5 mH

The space vector scaling constant K can be chosen arbitrarily. Depending on the situation, one choice may be more convenient than another.

Peak-value scaling: $K = 1$
RMS-value scaling: $K = \sqrt{1/2}$
Power-invariant scaling: $K = \sqrt{3/2}$

Throughout this project, RMS value scaling, $K = \sqrt{1/2}$ is assumed. For space vectors normalization, the selection of base values should include the space vector scaling constant K as follows:

Voltage: $V_{base} = \sqrt{2}KV_N$
Current: $I_{base} = \sqrt{2}KI_N$
Frequency: $\omega_{base} = \omega_N = 2\pi f_N$
Power: $S_{base} = P_{base} = Q_{base} = 3V_N I_N = \frac{3}{2K^2} V_{base} I_{base}$
Impedance: $Z_{base} = V_N/I_N = V_{base}/I_{base}$
Inductance: $L_{base} = Z_{base}/\omega_{base}$
Flux: $\psi_{base} = U_{base}/\omega_{base}$
Torque: $\tau_{base} = n_p P_{base}/\omega_{base}$
Inertia: $J_{base} = n_p \tau_{base}/\omega_{base}^2$
Viscous active damping: $b_{base} = n_p \tau_{base}/\omega_{base}$
Time: $t_{base} = 1/\omega_{base}$

With these base values, nominal values and corresponding p.u. values of important parameters are summarized in the Table 2.

Table 2: Nominal values and corresponding p.u. values of important parameters under consideration

Parameter	**Nominal Value**	**Nominal Value in p.u.**
Stator Voltage, V_s	127 V	1 p.u.
Stator Current, i_s	7.8 A	1 p.u.
Rotor Flux, ψ_R	0.35 Wb	0.83 p.u.
Torque, T_l	12.28 N-m	0.65 p.u.
Mechanical Rotor Speed, ω_m	1400 rpm	0.94 p.u.

B Source Code

Rather than reproducing the code here (and in line with the open-source philosophy of the project) all code can be found in the following public GitHub repository:

`https://github.com/beagledrive/dogdrive`

The code for the user interface is stored in a separate respoitory:

`https://github.com/beagledrive/dogdrive_ui`

C Demonstration HMI

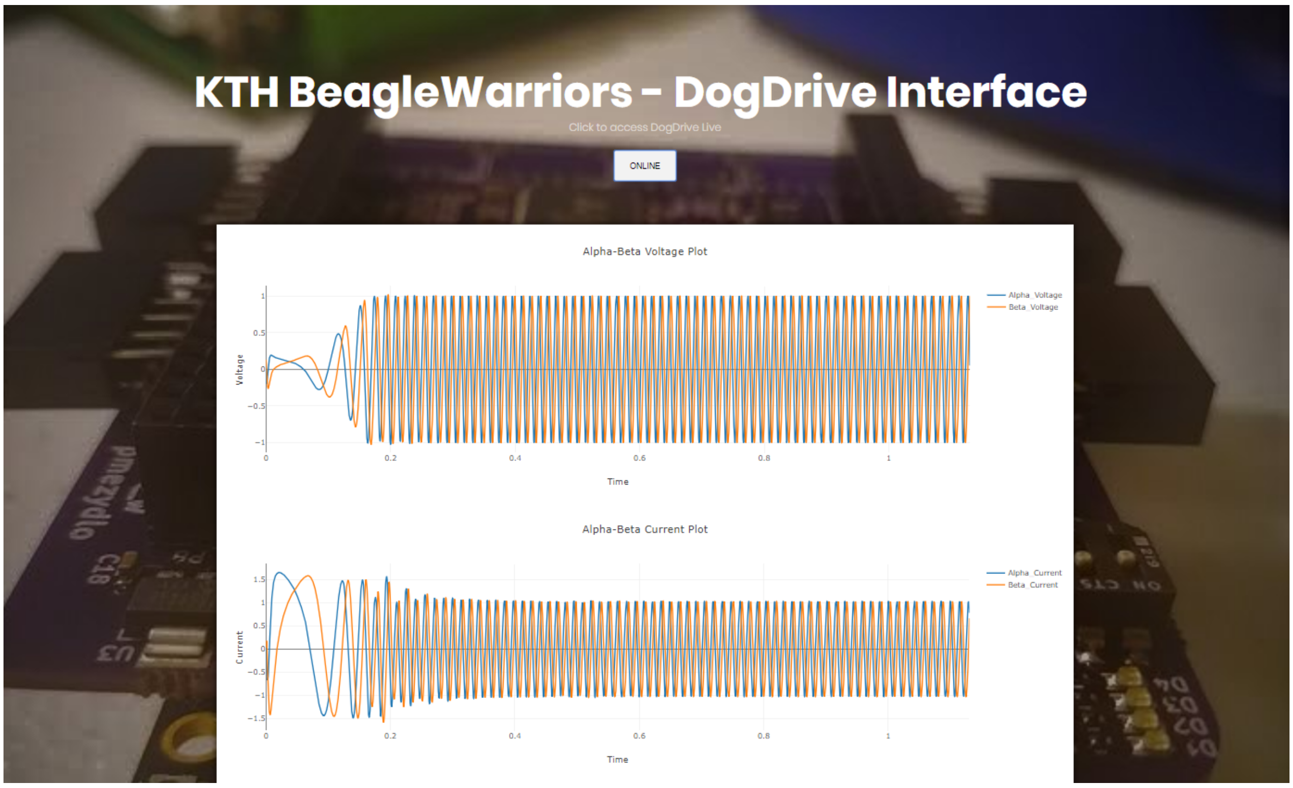


Figure 18: DogDrive HMI Screenshot

D Simulink Comparison Model

In this section, modelling of induction motor in MATLAB Simulink is presented. To verify the performance of the induction motor inside the DSP, an induction motor with same motor parameters are implemented in Simulink as shown in the Figure 19. Equations (11)-(15) are used to model the dynamics of induction motor in Simulink. Both the machines are operated at $u_\alpha = 1p.u.$, $u_\beta = 1p.u.$ and $T_{load} = 0$ and the corresponding stator current and rotor flux of both the machines are plotted as shown in the Figure 20. As seen in the Figure 20, stator current and flux response of the emulated machine inside DSP closely match to those of the simulated machine in Simulink.

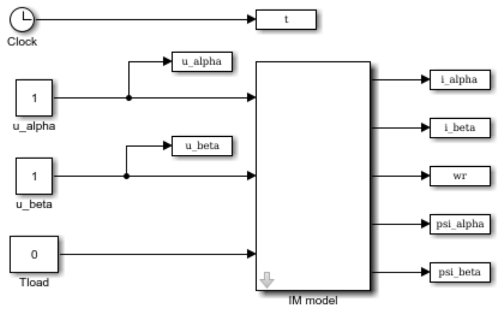


Figure 19: Induction Motor Simulink Model

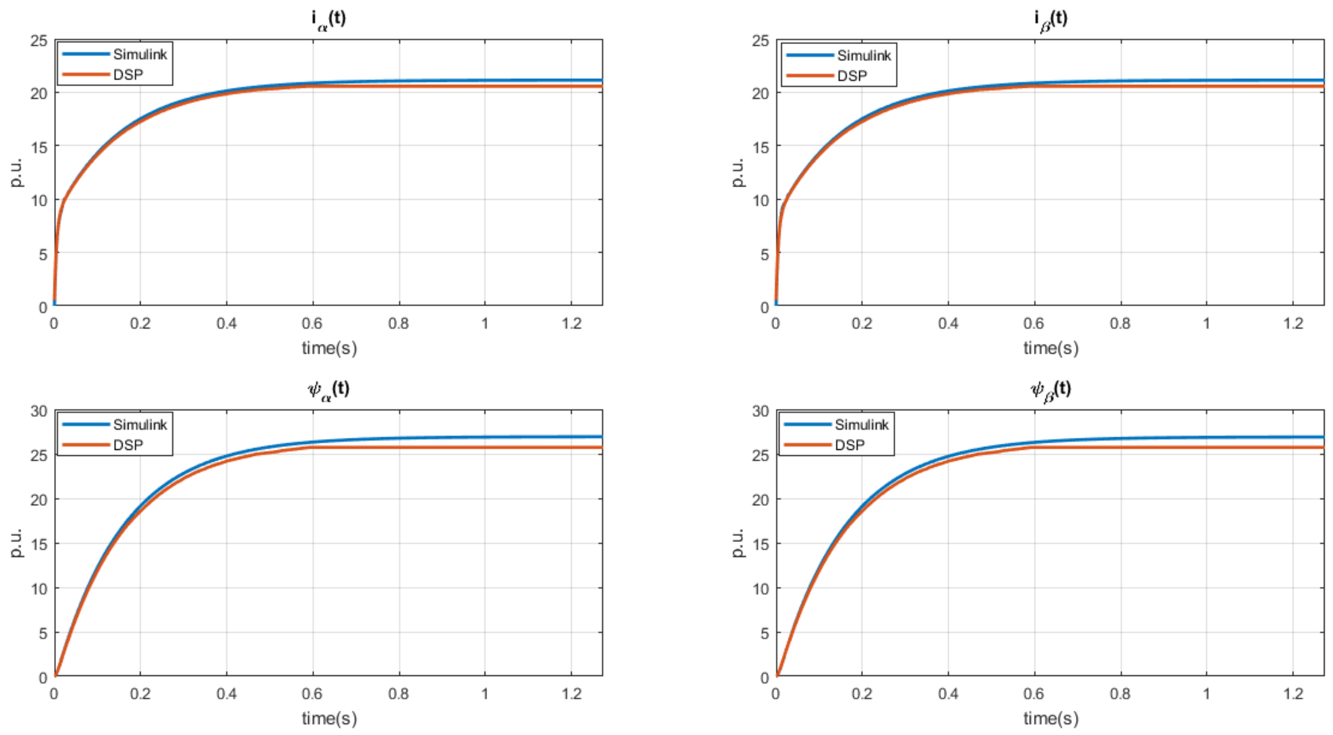


Figure 20: Performance comparison of simulated induction motor in DSP and in Simulink respectively

E Software Architecture Diagram

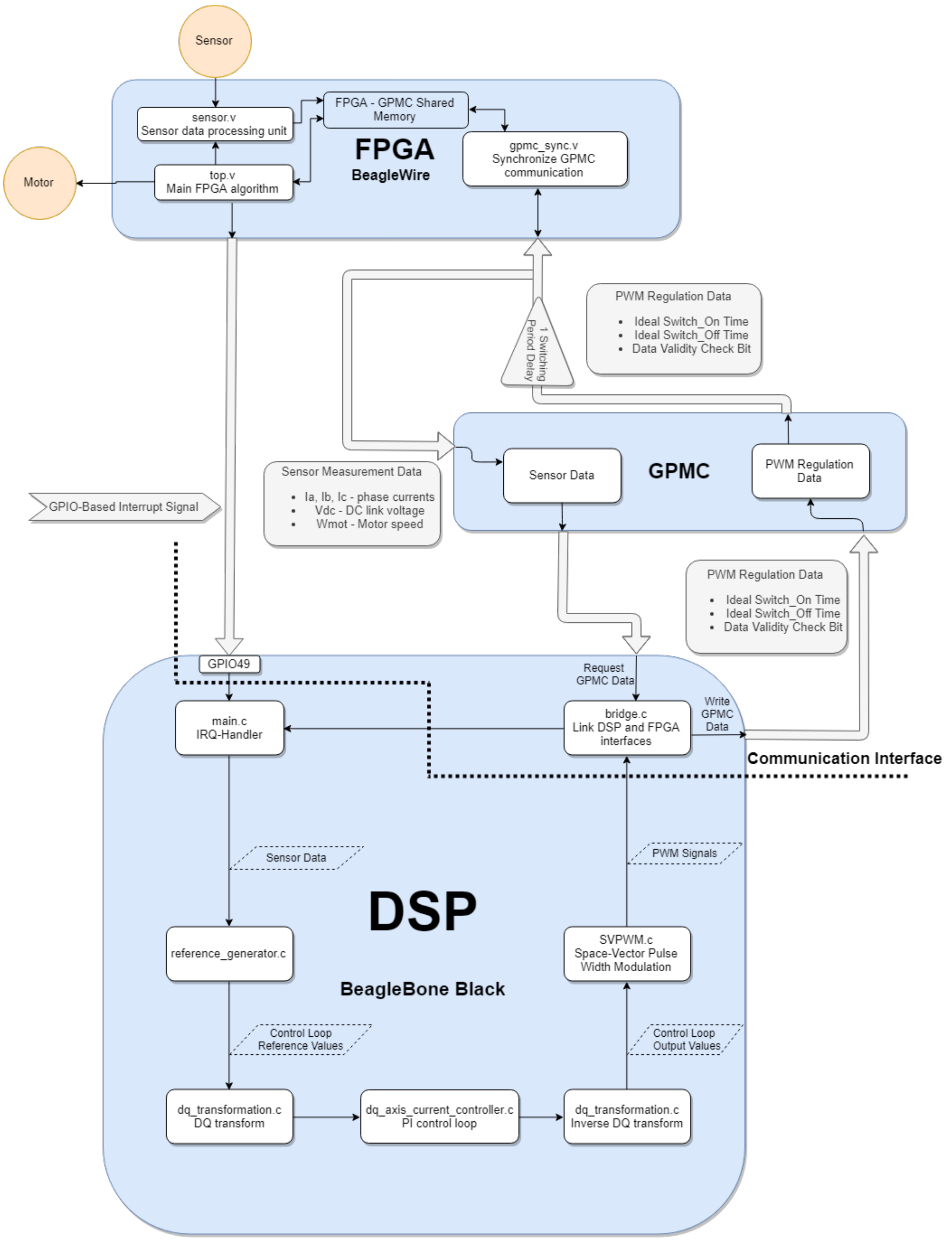


Figure 21: Software Architecture High Level Diagram

F FPGA GPMC Memory Locations

In this section, the table containing the memory locations where the FPGA GPMC messages are mapped is included.

Table 3: FPGA GPMC Memory Locations

Address	**Function**
0x00	Reset Bit
0x01	Enable Bit
0x02	Polarity Bit
0x03	Data Valid Bit
0x16	Phase A Switch On Lower Word
0x32	Phase A Switch On Upper Word
0x48	Phase A Switch Off Lower Word
0x64	Phase A Switch Off Upper Word
0x16	Phase B Switch On Lower Word
0x32	Phase B Switch On Upper Word
0x48	Phase B Switch Off Lower Word
0x64	Phase B Switch Off Upper Word
0x16	Phase C Switch On Lower Word
0x32	Phase C Switch On Upper Word
0x48	Phase C Switch Off Lower Word
0x64	Phase C Switch Off Upper Word

G UPWM Flow Diagram

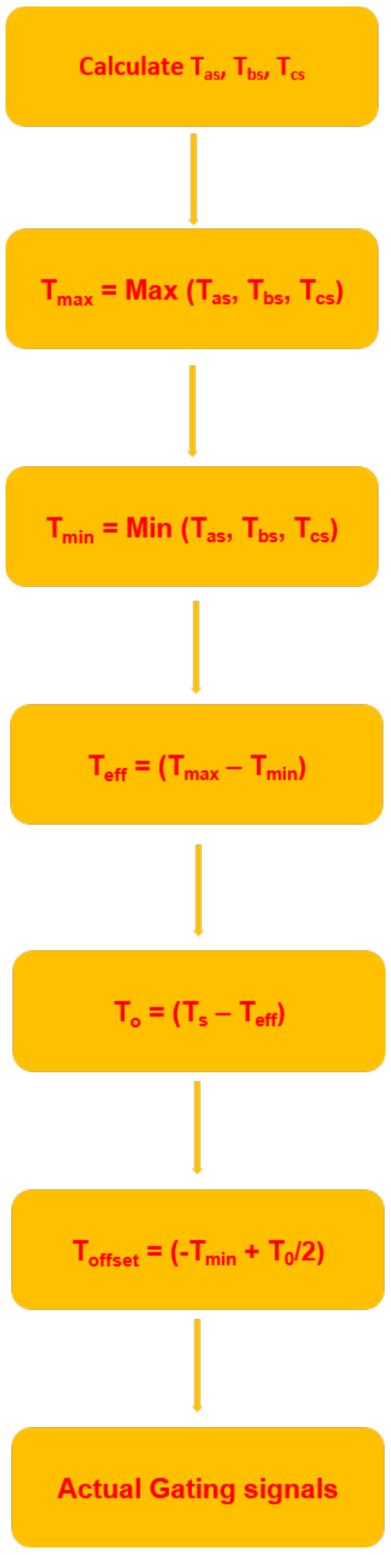

Calculate T_{as}, T_{bs}, T_{cs}
T_{max} = Max (T_{as}, T_{bs}, T_{cs})
T_{min} = Min (T_{as}, T_{bs}, T_{cs})
T_{eff} = (T_{max} – T_{min})
T_o = (T_s – T_{eff})
T_{offset} = (-T_{min} + T_0/2)
Actual Gating signals

Figure 22: Flow Chart for UPWM implementation

H Current Sensing Hardware Options

This section provides a detailed description of the available current sensing hardware options. This analysis is based on the description from the TI application note in [25].

H.1 Low-Side Implementation

In low-side current sensing, the sensor is implemented on the low side of the gate-driving FETs, in line with each switching leg, as shown in Figure 23(a).

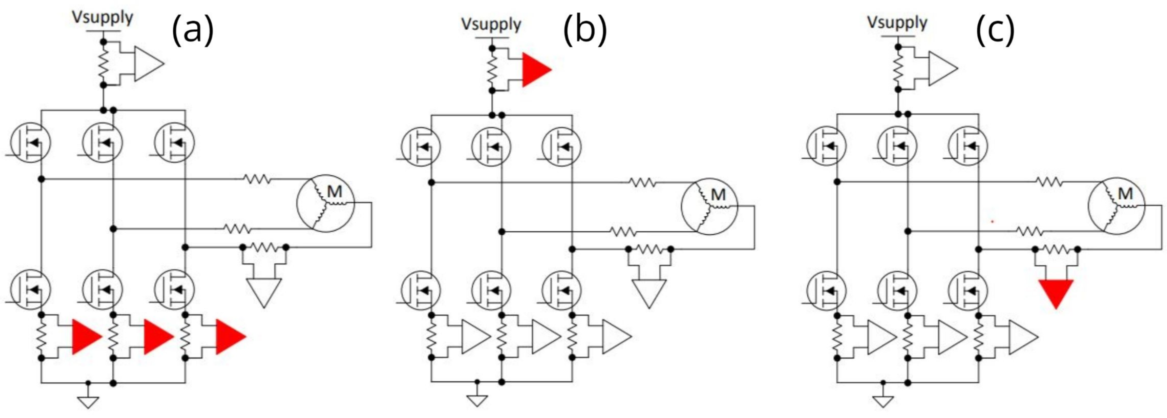


Figure 23: Different current sensor implementation methods from [25]

There are three different topologies of implementing current sensors in the Low-side implementation method.

1. Three-leg Current Sensing.
2. Dual-leg Current Sensing.
3. Single-leg Current Sensing.

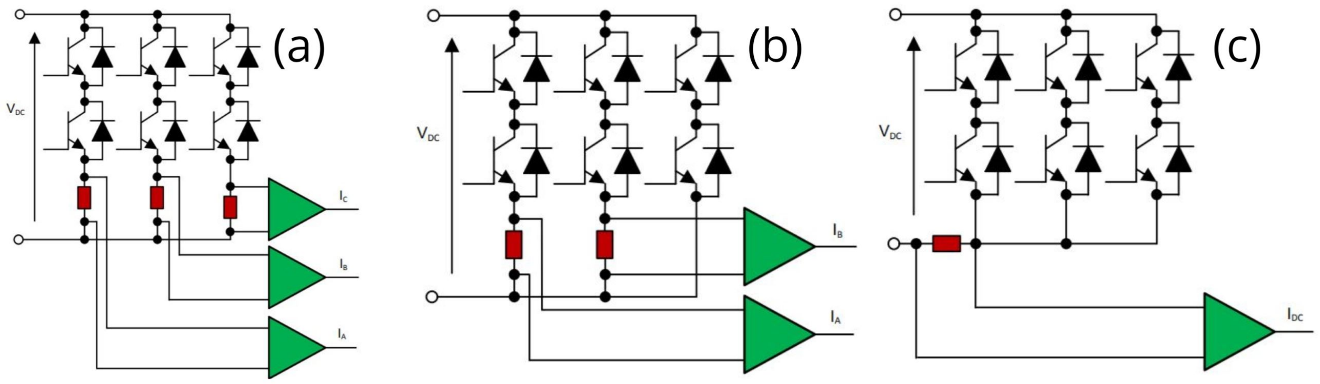


Figure 24: Different current sensing methods in low-side implementation from [25]

Three-leg Current Sensing The three-leg current sensing topology is shown in the Figure 24(a). This is the easiest topology in terms of computational ease, since the three phase currents are measured directly.

Dual-leg Current Sensing The two-leg current sensing method measures phase currents only in two legs of the inverter circuit, Kirchhoff's current law (KCL)is used to calculate the third phase current. The implementation is shown in the Figure 24(b).

Single-leg Current Sensing The single-leg current topology basically measures the power supply current and recreates each of the three-phase currents based on the current switching state. Figure 24(c) shows the single-leg implementation in the inverter.

H.1.1 High-Side Implementation

In high-side current sensing, the sensor is placed inline immediately following the supply voltage of the gate bridge, as shown in Figure 23(b).

H.1.2 Inline Implementation

In inline current measurement method, the sensor is implemented as shown in Figure 23 (c).

H.1.3 Commercial Sensors

There are three popular types of commercial current sensors available in the market.

1. Shunt resistor.
2. Hall-effect sensors.
3. Current transformers.

Shunt resistors are usually preferred because they provide an accurate measurement at a low cost. Hall effect current sensors are most commonly used as they are non-intrusive usually come with a small package with signal-conditioning circuit. Current-sensing transformers have low accuracy and are used in high-current or AC line-monitoring.

I Loading the Device Tree Overlay on BBB

This section describes the method for mounting the device tree overlay for the BW cape on the BBB.

To mount the overlay, the DTS needs to be compiled into a DTOB (DTO blob) and loaded into the kernel. To load the overlay, the DTOB must be copied to the firmware folder */lib/firmware/* and the universal boot loader (U-Boot) configured to load at boot. This is a matter of editing the */boot/uEnv.txt* file which is called on boot to load the overlay by adding the overlay to one of the four available slots with the additional line shown in the below code snippet.

```
enable_uboot_overlays=1
uname_r=4.14.41-bone14
uboot_overlay_addr0=/lib/firmware/BW-ICE40Cape-00A0.dtbo
disable_uboot_overlay_video=1
disable_uboot_overlay_emmc=1
```

Listing 1: Contents of uEnv.txt