THE DESIGN & SYNTHESIS OF A GRAPHICAL SYSTEM FOR THE VISUAL REPRESENTATION OF AUTOMOTIVE DATA

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF ENGINEERING TECHNOLOGY
OF WATERFORD INSTITUTE OF TECHNOLOGY
IN COMPLETE FULFILMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTERS OF ENGINEERING

By:
Dominick P. O' Brien

Supervised By:
Mr. Gavin Walsh

June 2007

Dedicated To:

My Mother: Aisling O’ Brien

And

My Grandparents: Patrick and Elizabeth O’ Brien

Declaration

I hereby declare that the material presented in this document is entirely my own work and has not been submitted previously as an exercise or degree at this or any other establishment of higher education. I, the author alone, have undertaken the work except where otherwise stated.

Signed: ________________________

Date: __________________

Acknowledgements

I hereby acknowledge the contributions to my work and offer my thanks to people who have helped and supported me during my work over the past two years.

My Supervisor:

> Mr. Gavin Walsh: I would like to thank Gavin for his constant encouragement, invaluable guidance and excellent supervision during the last two years.

My Family:

> I would like to thank my family for their support, encouragement and understanding throughout all of my studies.

The AAEC (Advanced Automotive Electronic Control) Research Group:

> I would like to take this opportunity to thank all members, both past and present, of the research group whose assistance, knowledge and support has been first-rate. I would like to pay a particular thanks to both Henry Acheson and Niall Murphy for their additional support throughout the project.

Mr. Denis O' Shea:

> I would like to sincerely acknowledge the financial support of Denis; without Denis' funding this project would not have materialised.

My Friends:

> A special word of thanks goes to all my friends who started out with me back in 2001 but left at different stations. Their humour and encouragement will never be forgotten.

There are also many more people who have contributed in countless others way and deserve my thanks also – Thank you!

Abstract

THE DESIGN & SYNTHESIS OF A GRAPHICAL SYSTEM FOR THE VISUAL REPRESENTATION OF AUTOMOTIVE DATA

By

Dominick P. O' Brien

Master of Engineering

Waterford Institute of Technology

This report investigates the design and implementation of a system that visually represents automotive data upon a connected graphical display. The devised system obtained vehicle data from numerous CAN nodes that were constructed to formulate an automotive network. The data was transmitted on this network and was interpreted by an intelligent-device. The intelligent-device manipulated the CAN data into an appropriate digital video stream. This stream was then converted into analogue format for display upon a monitor's screen. This report details all aspects of the design, testing and synthesis of this automotive application.

Table of Contents

Table of Figures

List of Tables

List of Abbreviations

A/V	Audio/Video
AC	Alternating Current
ACK	Acknowledge
A-D	Analogue-to-Digital
API	Application Program Interface
BRP	Baud Rate Prescalar
CAN	Controller Area Network
CANH	CAN High
CANL	CAN Low
CCLK	Processor Core Clock
CEC	Core Event Controller
CRC	Cyclic Redundancy Check
CRT	Cathode Ray Tube
CSMA	Carrier Sense Multiple Access
CSMA/CD+AMP	Carrier Sense Multiple Access with Collision Detection and Arbitration on Message Priority
DCB	Deferred Callback
DLC	Data Length Code
DMA	Direct Memory Access
DSP	Digital Signal Processor
EAV	End of Active Video
EMI	Electromagnetic Interference
ESD	Electrostatic Discharge
FPGA	Field Programmable Gate Array
fps	Frames per Second
GPS	Global Positioning System
GUI	Graphic User Interface
HDTV	High Definition Television

I^2C	Inter-Integrated Circuit
IC	Integrated Circuit
IDE	Identifier Extension
IP	Intellectual Property
ISO	*International Standardisation Organisation*
ISR	Interrupt Service Routine
ITU	*International Telecommunications Union*
IVG	Interrupt Vector Group
LCD	Liquid Crystal Display
LED	Light Emitting Diode
LLC	Logic Link Control
LSB	Least Significant Bit
MAC	Medium Access Control
MIPS	Millions of Instructions per Second
MSB	Most Significant Bit
NBR	Nominal Bit Rate
NBT	Nominal Bit Time
NRZ	Non-Return to Zero
NTSC	National Television System Committee
OSI	Open Systems Interconnection
PAL	Phase Alternating Line
PCI	Peripheral Component Interface
PLL	Phase Lock Loop
PPI	Parallel Peripheral Interface
QAM	Quadrature Amplitude Modulation
REC	Receive Error Counter
RGB	Red Green Blue
RX/TX	Receive/Transmit
SAE	*Society of Automotive Engineers*
SAV	Start of Active Video
SCLK	Processor System Clock (Peripheral Clock)

SIC	System Interrupt Controller
SJW	Synchronisation Jump Width
SNR	Signal-to-Noise Ratio
SPI	Serial Peripheral Interface
TEC	Transmit Error Counter
TWI	Two Wire Interface
USB	Universal Serial Bus

Chapter 1 - Introduction

1.1 Introduction

Since the advent of vehicular instrumentation, dash-panel displays have traditionally been implemented with analogue and mechanical dials and gauges. For example, fuel and temperature gauges were analogue electrical devices; whereas speedometers and tachometers were mechanically driven. From this dash-panel displays have today evolved to incorporate both analogue and digital dials and gauges. For instance, some cars currently utilise analogue electrical speedometer devices while they also include a tachometer that is comprised of a LCD (Liquid Crystal Display) [1]. The focus now is on developing dash-panel displays that dynamically represent vehicular data utilising a complete graphical approach.

This research project investigates the design and synthesis of an application which visually represents simulated automobile data. This information is obtained via a CAN (Controller Area Network) network and is displayed upon a connected monitor. Modern day vehicles contain many sensors that measure various performance parameters; e.g. automobile speed, oil temperature etc. Therefore this project necessitates a suitable method for mimicking the operation of such sensors in order to replicate authentic vehicle data. Once obtained, this information is transmitted over a CAN network and it is essential to appropriately manipulate it with the aim of representing it proportionally upon a display-device. Consequently, the goal of this research project is to utilise a suitable intelligent-device and additional resources to process CAN information and configure the data for visual representation. For this system a television screen is sufficient to act as a monitor utilised to illustrate vehicle data.

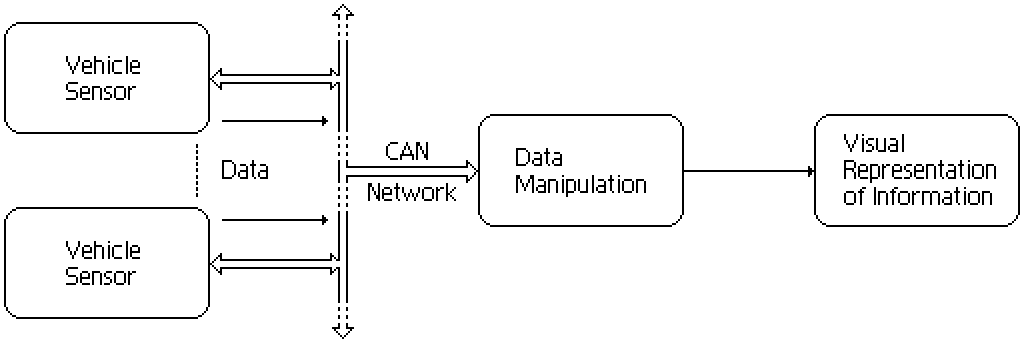


Figure 1: Application Overview

1.2 Thesis Organisation

The material and information presented in this thesis is compiled into two main sections. The first section, *Literature Review*, gives an overview of the protocols, technologies and components researched in order to formulate a suitable methodology for this application. This section is comprised of *Chapters 2*, *3* and *4*.

The second section, *System Synthesis*, details the implementation of the system design with respect to the findings of the *Literature Review* section. This particular section encompasses *Chapters 5*, *6* and *7*. Finally, there are conclusions drawn by the author based on the outcomes of the research and system implementation.

The work presented in this thesis is laid out as follows:

Chapter 2:	*Chapter 2* discusses the CAN protocol used for in-vehicle networking; detailing the exact composition of a CAN message and the physical make-up of an automotive network.
Chapter 3:	*Chapter 3* introduces the fundamentals of video processing to the reader. It describes the basics relating to video data and discusses in detail a particular digital video standard.
Chapter 4:	*Chapter 4* details the selection of a suitable processor for utilisation within this system. This chapter describes how the author evaluated numerous processors under several key considerations in order to establish the most fitting component for the system's development.
Chapter 5:	*Chapter 5* outlines the implementation of the CAN network employed in this system. It describes to the reader how the selected components were coordinated, both in terms of hardware and software, to instigate the network. An account of the test algorithms used to establish correct functionality of the network is also included.
Chapter 6:	*Chapter 6* discusses the measures taken by the author to implement a video display incorporating the device chosen in *Chapter 4*. It describes how encountered errors were overcome in order to devise a correct video module strategy.
Chapter 7:	*Chapter 7* describes how the strategies devised in *Chapters 5* and *6* were combined to formulate the system's synthesis. A discussion is included outlining how encountered problems were surmounted in order to visually represent vehicle data upon a display device.
Chapter 8:	*Chapter 8* outlines the conclusions derived by the author based on the research and system implementation conducted. A discussion regarding further possibilities for research based on findings from this particular study is also provided.

Table 1: Table of Chapters

SECTION I – TECHNICAL & LITERATURE REVIEW

Chapter 2 - CAN Bus Protocol

2.1 Introduction

This chapter details and outlines the CAN bus protocol. The information given in this chapter is partitioned into the following sections:

- An overview describing the history and fundamental ideas behind the introduction of the CAN protocol for utilisation as a vehicle networking standard.

- An account detailing how the CAN protocol is physically implemented including a discussion on bit rates and timing.

- A look at what exactly constitutes a CAN message and how devices connected to the network achieve synchronisation with each another.

- A synopsis portraying arbitration and error confinement within the CAN bus protocol.

2.2 CAN Bus Protocol - An Overview

The CAN protocol is an advanced asynchronous serial-bus system that efficiently supports distributed control systems. It was initially developed for use in automobiles by *Bosch* in the late 1980s [2]. The CAN protocol is internationally standardised by the *ISO* (*International Standardisation Organisation*) and the *SAE* (*Society of Automotive Engineers*) [3]. *ISO11898* is the international standard for high-speed CAN communications in automobiles. CAN is presently being employed as the standard for vehicle communications within Europe by automobile manufacturers. Meanwhile, it is gaining more mainstream acceptance within the United States [4].

CAN is similar in principle to other serial communication protocols such as *SPI* (Serial Peripheral Interface) [5]; however it is more complex. It is a "message-based" protocol as opposed to an "address-based" network system such as I^2C (Inter-Integrated Circuit) [6]. This essentially means that devices connected to a CAN network do not have unique addresses, but rather the message(s) that a device sends out onto the network possesses a unique ID number [7]. As a result, each device on the network listens to every message transmitted on the bus and determines what action, if any, it needs to take. For that reason, this implies that a CAN network may contain multiple masters.

The development of CAN began as a result of the increasing quantity of electronic components and control systems being incorporated into modern-day motor vehicles [3], [8]. Examples of such components/systems include engine management systems, transmission control and central locking. The integration of these electronic components/control systems result in additional safety and comfort features for the driver; thus enhancing the vehicle as a whole. To further these improvements it was necessary for the different control systems to exchange information [3], [7]. Previously, this was carried out using discrete interconnection of the different systems, i.e. point-to-point wiring. The requirement for data exchange has since grown to such an extent that a cable network with a length of up to several kilometres, with many connectors, would be required if point-to-point wiring was employed.

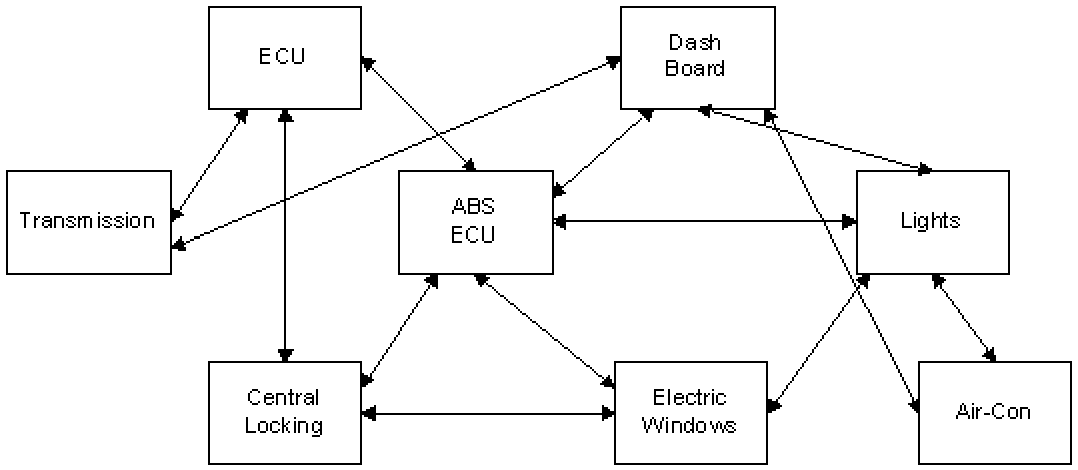


Figure 2: Automobile Systems Interconnected using Point-to-Point Wiring

Subsequently a solution to this problem was realised with the design and introduction of the CAN bus protocol. Within the CAN protocol point-to-point wiring was replaced by a single serial-bus connecting all control systems and electronic devices on the network [2], [4].

The design of the CAN protocol had to take into consideration some special requirements due to its employment within a vehicle. Examples of such special requirements include durability and reliability. This is accomplished in the design by adding some CAN specific hardware to each control unit that provides the “rules” of the protocol for transmitting and receiving information via the bus. The combination of CAN specific hardware and a particular control system/electronic device leads to the formulation of a CAN node. Each of the nodes on a particular network has a solitary interface to the serial-bus network thus allowing communication between attached nodes. Due to the fact that each of the nodes on a CAN network connects to the same serial-bus there is a considerable reduction in cable length requirements.

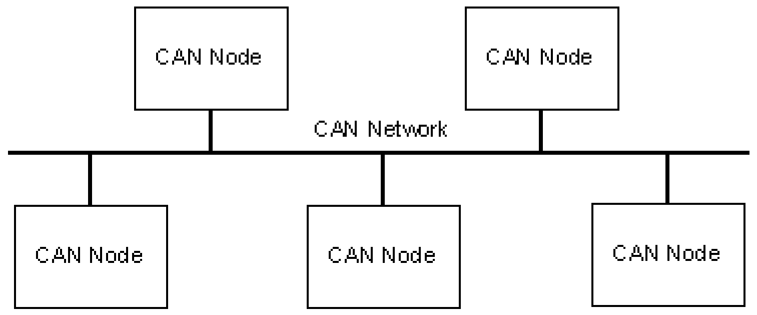


Figure 3: CAN Serial-Bus Interface Leading to Reduced Cable-Lengths

As CAN is an asynchronous multi-master message-based protocol, the designer can implement a degree of flexibility into how and when nodes communicate over the network. For instance, a particular node may only transmit a message every twenty milliseconds, while another node may only transmit data if, for instance, a temperature rises above a pre-determined value [9]. Therefore it is easy to see that the use of CAN within an automobile introduces adaptability and practicality to a designer for each individual network.

2.3 CAN & the OSI Model

The CAN protocol, like many other network topologies, can be illustrated using the seven-layer *OSI* (Open Systems Interconnection) model [4], [10], [11]. This layered approach is intended to achieve interoperability between standard components from different manufacturers. With reference to this model the CAN protocol defines the functions and services of the *Data Link Layer* and also the bit-timing and synchronisation components of the *Physical Layer* [12]. The remaining elements of the *Physical Layer* and the five additional layers are purposely not defined within the CAN protocol [10]. The implementation of these additional layers is completely at the hand of the system-designer so that specific system requirements can be met.

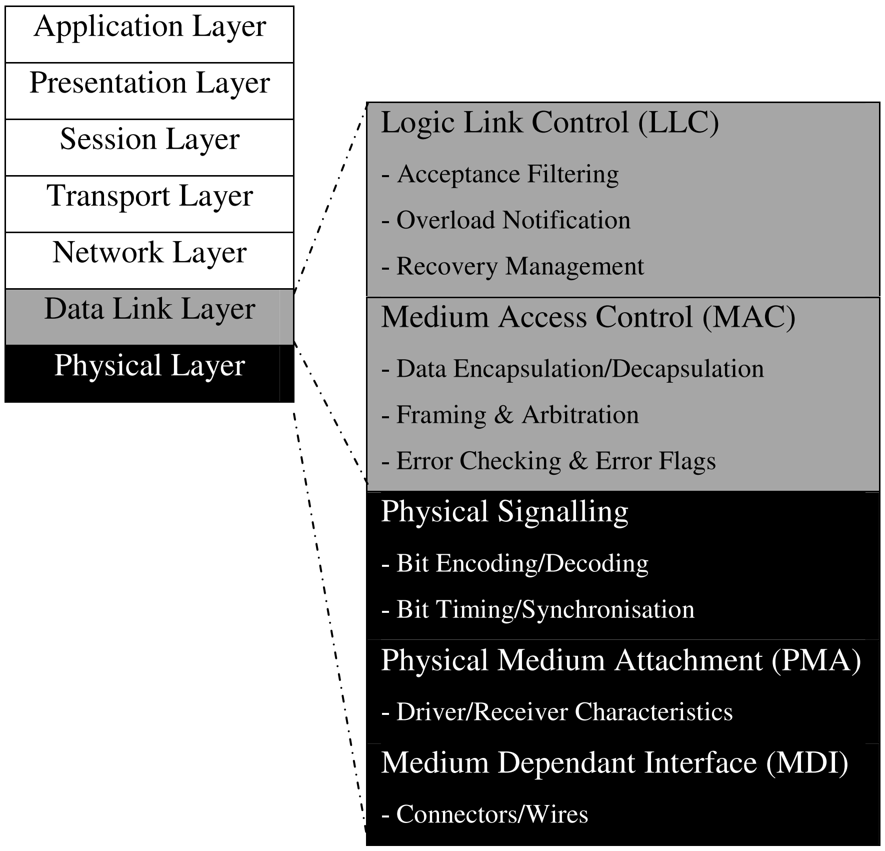


Figure 4: CAN Protocol with Reference to the *OSI* Model

The additional five layers of the *OSI* model are typically implemented by a system-designer using a number of hardware/software components which complete the formation of a CAN node. The components that classically comprise a node are as listed below:

- **Application Software** that controls a particular function e.g. Measure water temperature. Application software executing within a particular CAN node may perform a singular function, or a number of functions depending on the situation.

- **Microcontroller** (or a corresponding intelligent embedded device) upon which the application software executes. This device also transmits/receives relevant information to/from a CAN Controller at typical digital logic levels.

- **CAN Controller** is used to read data from the microcontroller and write it to a CAN Transceiver. Conversely, a CAN Controller may receive data from a CAN Transceiver, via the network, and transmit it to the microcontroller. The CAN Controller generally interfaces with a microcontroller (or an equivalent intelligent embedded device) via a *SPI* link. This device typically contains components which allow for the filtering of unwanted messages transmitted over the network resulting in the reduction of the microcontroller's overhead.

- **CAN Transceiver** exchanges information with a CAN Controller and broadcasts it over the asynchronous network. Additionally this device converts the digital signals supplied to it by a CAN controller to signals suitable for transmission over the bus cabling. A CAN Transceiver also provides a buffer between the CAN Controller and the high-voltage spikes that can be generated on the CAN bus by outside sources (EMI, ESD, electrical transients, etc.).

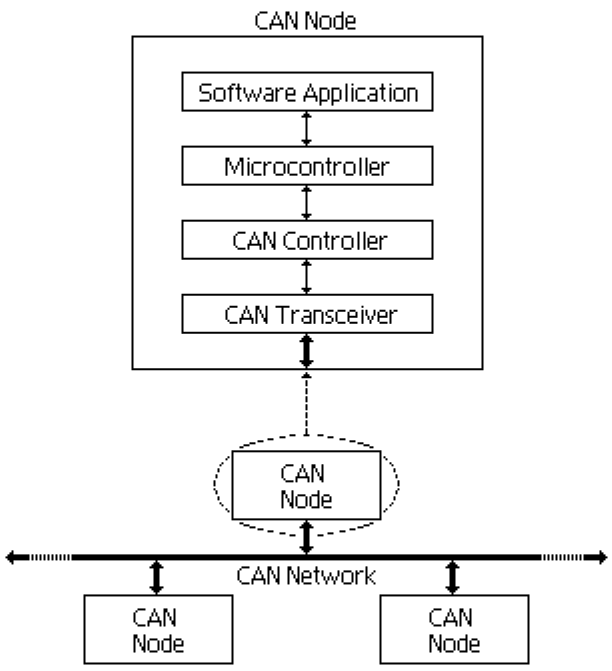


Figure 5: Typical CAN Node

The CAN Controller and Transceiver are the hardware units, mentioned earlier, that help to meet numerous requirements like durability and reliability. With the popularity of CAN increasing, not just within the automotive industry [9], but also within other sectors, many IC (Integrated Circuit) manufacturers have taken the step of integrating CAN Controller modules into their microcontrollers. This consequently eradicates the *SPI* link previously needed between a microcontroller and a peripheral CAN Controller.

2.3.1 CAN's Physical Layer

As mentioned earlier, the CAN protocol only defines the bit-timing and encoding portions of the *Physical Layer*. The *Physical Layer* essentially defines how the raw-data is actually transmitted over the network [7].

2.3.1.1 Bit Encoding

Fundamentally the CAN bus protocol uses *NRZ* (Non-Return to Zero) bit-encoding to represent data [13]. *NRZ* encoding represents data with Logic 1 or 0 levels during the entire bit time. If two or more Logic 1s (or Logic 0s) occur in succession, the waveform does not return to Logic 0 (Logic 1) level until Logic 0 (Logic 1) actually occurs.

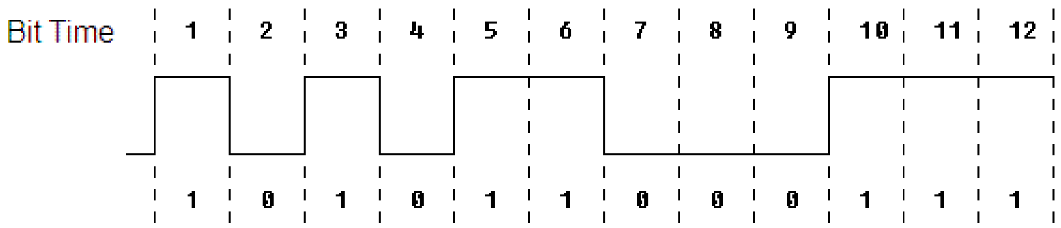


Figure 6: An Example of a *NRZ* Waveform

The CAN protocol specifies two logical states - *dominant* (Logic 0) and *recessive* (Logic 1). *ISO11898* defines a differential voltage, V_{DIFF}, to represent these two logic states.

Typically, a twisted-wire pair is used to transfer data over the network using. Data can also be transferred over the network using other physical-phenomena e.g. light pulses. The wires are twisted together to prevent electromagnetic interference from other electrical devices internal or external to the vehicle. One of the wires is given the label *CANH* (CAN High), while the other is given the label *CANL* (CAN Low) [3], [7]. The differential signal between the voltages carried in each wire defines the bus state.

$$V_{DIFF} = V_{CANH} - V_{CANL} \qquad \textit{Eq. 2.1}$$

, where V_{DIFF} is the differential voltage (Volts),

V_{CANH} and V_{CANL} are the *CANH* and *CANL* voltages respectively (Volts).

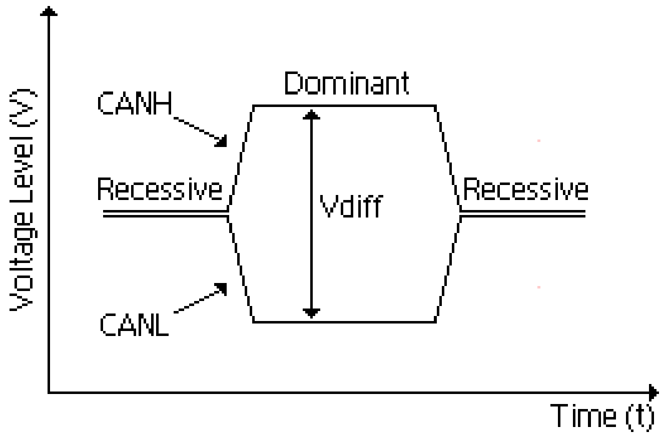


Figure 7: Differential Bus Signalling

In the *recessive* state, the differential voltage between the two signals is less than a minimum threshold. Conversely, in the *dominant* state the differential voltage between *CANH* and *CANL* is greater than a minimum threshold. A *dominant* bit will always have precedence over a *recessive* bit as CAN uses the *Wired-AND* mechanism [3]. Under this system if any node transmits a *dominant* bit the bus resides in the *dominant* state; the CAN bus only exists in the *recessive* state when all nodes on the network transmit *recessive* bits.

Node A	**Node B**	**Node C**	**Bus State**
Dominant	*Dominant*	*Dominant*	*Dominant*
Dominant	*Dominant*	*Recessive*	*Dominant*
Dominant	*Recessive*	*Dominant*	*Dominant*
Dominant	*Recessive*	*Recessive*	*Dominant*
Recessive	*Dominant*	*Dominant*	*Dominant*
Recessive	*Dominant*	*Recessive*	*Dominant*
Recessive	*Recessive*	*Dominant*	*Dominant*
Recessive	*Recessive*	*Recessive*	*Recessive*

Table 2: Truth-Table for *Wired-AND* Mechanism

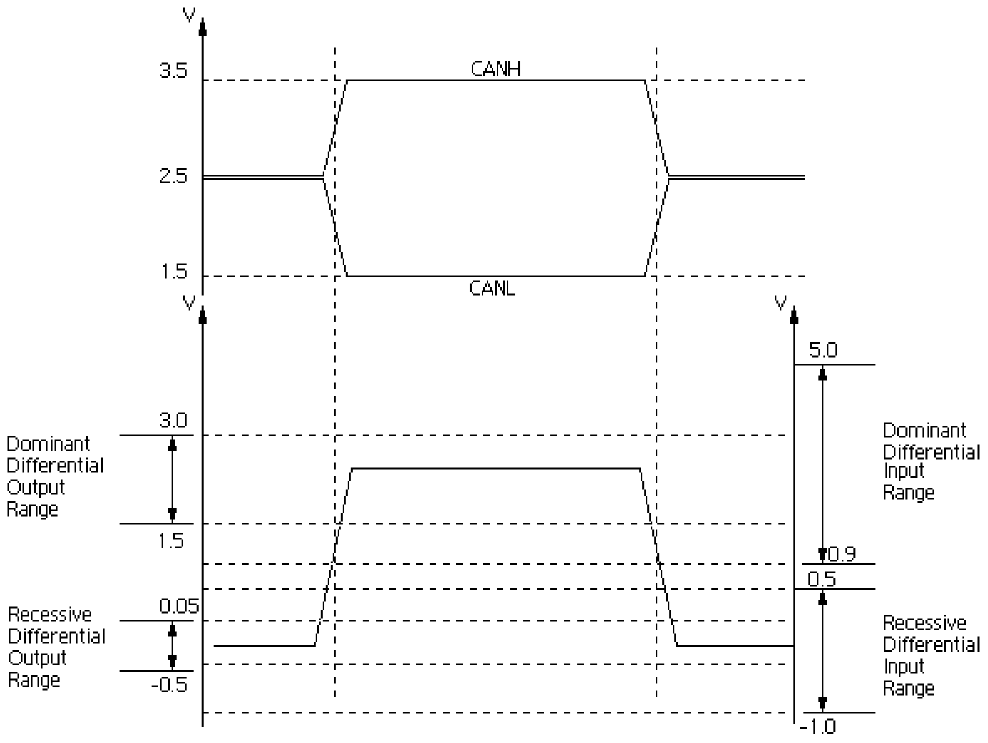

V
3.5
CANH
2.5
1.5
CANL
V
V
5.0
3.0
Dominant Differential Output Range
1.5
Dominant Differential Input Range
0.9
0.5
0.05
Recessive Differential Output Range
-0.5
Recessive Differential Input Range
-1.0

Figure 8: *ISO11898* Nominal Bus Levels

2.3.1.2 Transmission Medium and Connectors

With reference to the *OSI* model, even though the CAN protocol itself does not define the *PMA* and *DMA* sub-layers of the *Physical Layer*; *ISO-11898-2* makes recommendations for the *PMA* and *DMA* sub-layers. *ISO-11898-2*, however, does not define the mechanical wires and connectors to be used; but on the other hand stipulates numerous electrical specifications for the mechanical connectors and wires. The specification requires that each end of the CAN network is terminated using 120Ω resistors [10]. The terminating resistors prevent data on the network from being reflected back when the signal reaches the end of the system. If the signal was reflected it could cause errors on the CAN network.

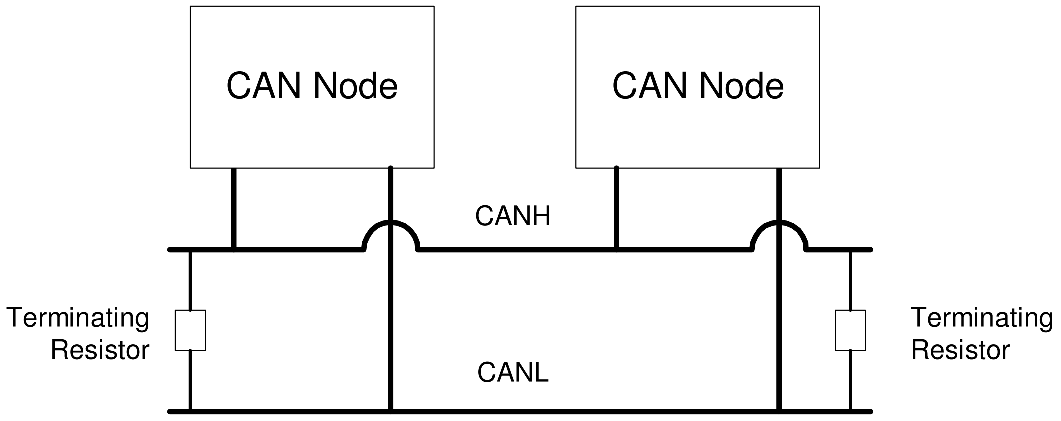


Figure 9: CAN Network with Terminating Resistors

2.3.1.3 Bit Rates & Timing

The CAN protocol can achieve data rates of up to 1MBit/s. In today's terms this is considered to be moderately slow when compared to other networks. Nevertheless, CAN's transfer speed is more than adequately equipped to deal with the transmission of data inside in an automobile. One of the appealing aspects of CAN for network designers is that it's bit rate, bit sample point and the number of samples in a bit period are user programmable. Modern high-speed CAN networks use crystal oscillators to derive their bit timing. Each node has its own timing reference but it is not necessary for all nodes on a particular network to use the same oscillator frequency [12].

A CAN message is made of numerous bits. Each of these bits has a specific period, t_{bit}. This parameter, t_{bit}, is itself made up of a number of non-overlapping portions.

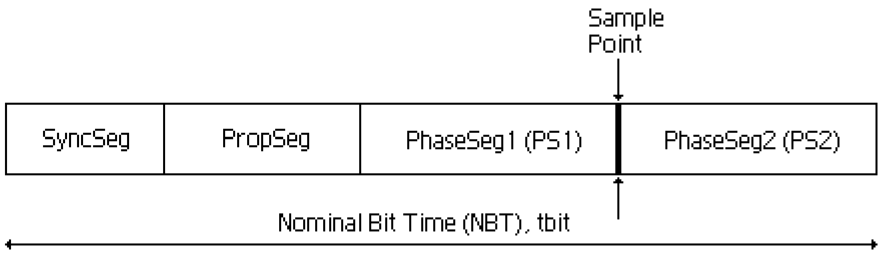


Figure 10: CAN Bit Time Segments

These non-overlapping segments are made up from an integer number of units called time quantum, t_q. The *NBR* (Nominal Bit Rate) is defined within the CAN specification to be the number of bits per second transmitted by an ideal transmitter with no resynchronisation and can be described using the following [14]:

$$NBR = f_{bit} = \frac{1}{t_{bit}} \qquad \textit{Eq. 2.2}$$

, where *NBR* is the nominal bit rate (Seconds),
f_{bit} is the frequency of a bit (Hertz),
t_{bit} is the bit period (Seconds).

From the preceding diagram it can be seen that the *NBT* (Nominal Bit Time), or t_{bit}, can be expressed as a summation as follows:

$$t_{bit} = t_{SyncSeg} + t_{PropSeg} + t_{PS1} + t_{PS2} \qquad \textit{Eq. 2.3}$$

, where t_{bit} is the bit period (Seconds),
$t_{SyncSeg}$ is the synchronisation segment period (Seconds),
$t_{PropSeg}$ is the propagation segment period (Seconds),
t_{PS1} and t_{PS2} are the periods (Seconds) for phase segment 1 and 2 respectively.

The first portion of the *NBT*, the synchronisation segment (*SyncSeg*), is used to synchronise nodes connected to the bus. The duration of this segment is always one t_q. Bit edges are expected to occur during this portion. The propagation segment (*PropSeg*) is user programmable and is used to compensate for propagation delays between communicating nodes. The system-designer can program the duration of the propagation segment to be from one to eight t_q in duration. Phase segments 1 and 2 (*PhaseSeg1* and *PhaseSeg2*) are used to compensate for any edge error that occurs around the sample point. The sample point is the instance in the bit time where the logic level is read. This is typically read at the end of *PhaseSeg1*. However, the system-designer has the option to sample the logic level three times during the *NBT*. If so, two additional samples are taken at half t_q intervals prior to the end of *PhaseSeg1*. The durations of *PhaseSeg1* and *PhaseSeg2* are also user defined; *PhaseSeg1* can be lengthened, or conversely, *PhaseSeg2* can be shortened. *PhaseSeg1* is programmable from one to eight t_q and *PhaseSeg2* is programmable from two to eight t_q [14].

The duration of a time quantum, t_q, is derived from the period of the oscillator, t_{osc}, employed within an individual node. The base t_q is equal to twice t_{osc} and is also equal to one t_q clock period, t_{brpclk}. The figure for t_q can be modified by the system-designer from its base value using a programmable prescalar called the *BRP* (Baud Rate Prescalar) in order to change the period of t_{brpclk} [14]. The relationship between these parameters is mathematically illustrated below:

$$t_q = 2 \times BRP \times t_{osc} \qquad \textit{Eq. 2.4}$$

$$\Rightarrow t_q = \frac{2 \times BRP}{f_{osc}} \qquad \textit{Eq. 2.5}$$

, where t_q is the time quantum (Seconds),

BRP is a user-configurable prescalar integer unit,

t_{osc} is the period of an oscillator used within a node (Seconds),

f_{osc} is the frequency of an oscillator used within a node (Hertz).

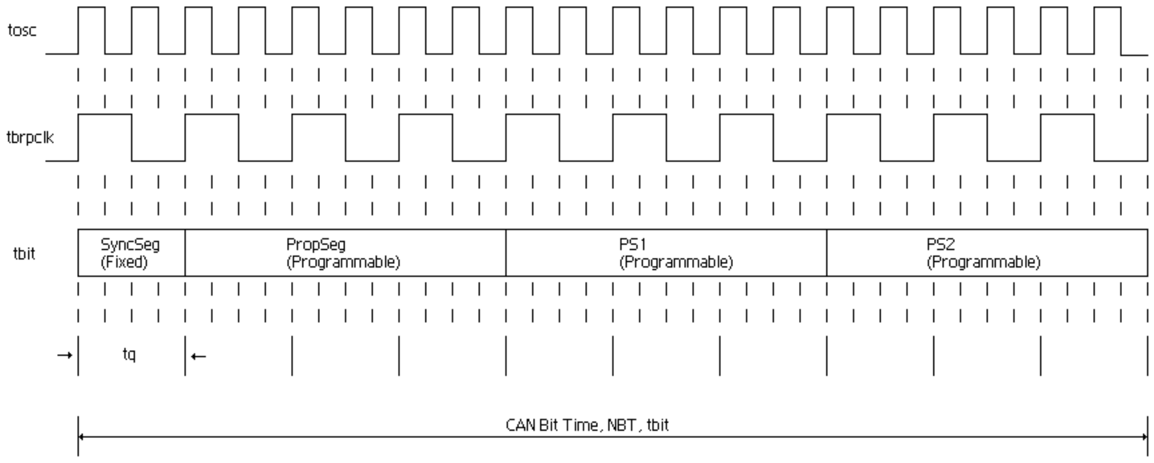


Figure 11: Time Quantum, t_q, & the Bit Period, t_{bit}

2.3.1.4 Bus Lengths & Synchronisation

ISO11898 states that a CAN Transceiver must be able to drive a bus length of approximately forty-metres at a data rate of 1MBit/s [10]. A longer bus length can be realised by implementing a slower data-rate on the network.

Bit Rate (kBits/s)	**Bus Length (m)**
1000	30
500	100
250	250
125	500
62.5	1000

Table 3: CAN Bit Rate vs. Bus Length [14]

Within CAN, relationships exist between the bit timing parameters and the oscillator tolerances; and as a result physical bus propagation delays. For a CAN network the propagation delay, t_{prop}, is calculated as being a signal's round trip time on the physical bus, t_{bus}, plus the output driver delay, t_{drv}, plus the input comparator delay, t_{cmp}. Assuming all devices on a CAN bus have similar component delay-times the propagation delay of a CAN network can be expressed mathematically as follows:

$$t_{prop} = 2 \times (t_{bus} + t_{drv} + t_{cmp}) \qquad \textit{Eq. 2.6}$$

, where t_{prop} is the network propagation delay (Seconds),
t_{bus} is the time duration of a signal's round-trip (Seconds),
t_{drv} is the delay of the output driver (Seconds),
t_{cmp} is the input comparator delay (Seconds).

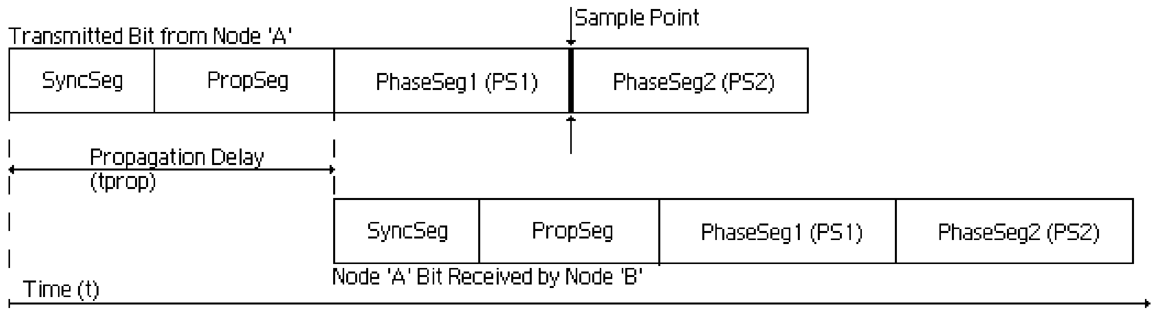


Figure 12: Propagation Delay between a Transmitting & Receiving Node

The bit timing parameters, the oscillator tolerances, and the propagation delays of a CAN network are interrelated due to the fact that the later the sample point in the bit period is taken, the more tolerance the system has to propagation delay. This means greater bus lengths can be installed. Conversely a sample taken closer to the midpoint of the bit period achieves greater oscillator tolerance levels. Therefore it is easy to see that a system-designer is left with a trade-off; greater bus length vs. large oscillator tolerance [12].

Earlier, it was mentioned that each CAN node has its own timing reference and that it is not necessary for all nodes on a particular network to use the same oscillator frequency. However all devices connected to a CAN bus must operate at the same *NBR*. This is achieved by the system-designer by varying the *BRP* of each node to ensure a consistency in nominal bit rate between all devices connected to the network. Factors such as noise, phase shifts, and oscillator drift lead to situations where the ideal bit rate does not equal the actual bit rate in a real system. Therefore, the nodes must have a method for achieving and maintaining synchronisation with messages transmitted on the bus [14].

As discussed previously a *dominant* bit will always have precedence over a *recessive* bit. With this style of arbitration in place each node involved with arbitration must be able to sample each bit level with the same bit time otherwise invalid arbitration may occur. For the CAN protocol there are two categories of synchronisation which guarantee suitable decoding of messages despite hindrances like phase errors etc. [12]; the two categories are as follows: *Hard Synchronisation* and *Resynchronisation*.

Hard Synchronisation occurs on the first *recessive*-to-*dominant* (Logic 1 to Logic 0) edge during an idle period on the network which indicates a *Start-of-Frame* condition. Every CAN Controller on the network now initialises its current bit period timing at this first *recessive*-to-*dominant* transition with *SyncSeg* [12], [14]. At this point, all of the receiving nodes will be synchronised to the transmitting device. *Hard Synchronisation* occurs only once during a message.

Resynchronisation is carried out once for each *recessive*-to-*dominant* transition throughout the remainder of the received message. It is implemented to uphold the preliminary synchronisation carried out on the first *recessive*-to-*dominant* transition using *Hard Synchronisation*. If *Resynchronisation* is not employed receiving nodes could loose synchronisation due to factors such as oscillator drift and noise. *Resynchronisation* is typically implemented using a *PLL* (Phase Lock Loop) which compares and eradicates any variations existing between the actual *recessive*-to-*dominant* transition and the expected (during *SyncSeg*) *recessive*-to-*dominant* transition [2], [14]. *Resynchronisation* compensates for any phase error by as much as the user defined parameter *SJW* (Synchronisation Jump Width). *SJW* is not a segment within the bit period, t_{bit}; it is a value which defines the maximum number of t_q by which a bit period can be lengthened/shortened in the event of resynchronisation [12]. The user can program the value of *SJW* to be in the range of one to four t_q.

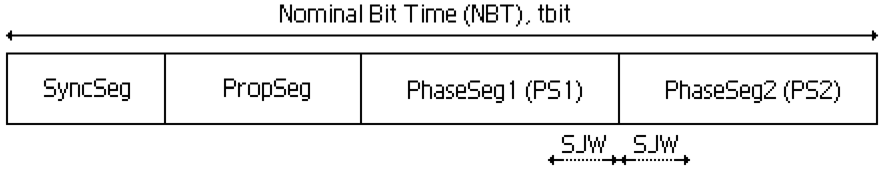


Figure 13: *SJW* used for Resynchronisation

The stipulations seen in the table below must be adhered to by a system-designer in order to comply with the synchronisation standards outlined in *ISO11898* [2], [14].

1.	Only a single synchronisation within a particular bit period, t_{bit}, is allowed.
2.	Only *recessive*-to-*dominant* transmissions are to be used for synchronisation purposes.
3.	*Hard Synchronisation* is only performed whenever there is a *recessive*-to-*dominant* transition during an idle-bus condition.
4.	All other *recessive*-to-*dominant* transitions will be used for resynchronisation purposes.
5.	$SJW \leq PhaseSeg2 \leq PhaseSeg1$

Table 4: Important CAN Bit Timing & Synchronisation Rules

2.3.2 CAN's Data Link Layer

The *Data Link Layer* is primarily responsible for assembling the encoded data produced in the *Physical Layer* into an *ISO11898* structured frame. This layer, with reference to the *OSI* model, is also required to perform arbitration and error confinement [8], [9]. For this discussion, with reference to the *OSI* model, it is only necessary to describe the *MAC* (Medium Access Control) section of the *Data Link Layer*. The *LLC* (Logic Link Control) section is outside the scope of this discussion.

2.3.2.1 Message Framing

As outlined above the raw-data encoded in the *Physical Layer* has to be bundled into a predefined structure called a *frame* as outlined in *ISO11898*. The CAN protocol defines four different types of frames [9], [15]. A brief description of the various frame types is described in the table below.

Data Frame:	Data is sent by a transmitting node to one or more receiving nodes. This is the most common type of CAN message.
Remote Frame:	A Remote Frame is used when one node requests the transfer of information from another device connected to the CAN bus.
Error Frame:	This type of CAN message is generated by a node when it detects a particular protocol error defined within the *ISO11898* standard.
Overload Frame:	This is used within the CAN protocol to request additional time needed by a node to process received information.

Table 5: Four Categories of CAN Messages

From above it can be seen that each of the message frames serves its own particular function. Each of the frame types differ somewhat in their structure; although substantial similarities exist between all of them.

Data Frame

The *Data Frame* will be discussed in greater detail than the other categories because it is the most commonly employed frame type.

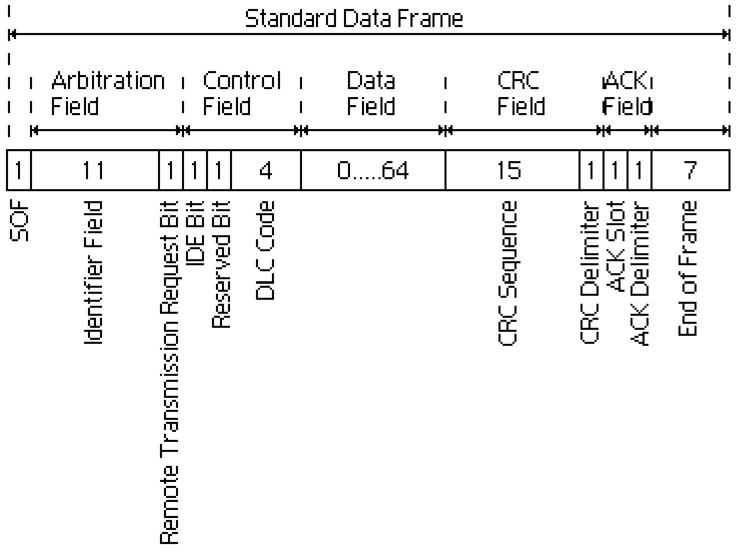


Figure 14: Standard CAN *Data Frame*

The diagram above illustrates the composition of a standard CAN *Data Frame*. The frame consists of a number of *fields*; this is true for the remaining three frame types. A *field* within a frame is compromised of a number of bits. The composition of a *Data Frame* is as described below [7], [15], [16]:

- **Start-of-Frame Field:** The *Start-of-Frame* field is always one bit in length and is represented by a *recessive*-to-*dominant* transition. It is used to indicate the start of a new message. Also, as discussed previously, the *Start-of-Frame* field is also used for *Hard Synchronisation* purposes.

- **Arbitration Field:** This field is comprised of twelve bits and is used to prioritise messages transmitted on a CAN network. The first eleven bits of this field consist of the *Identifier Field* portion. These eleven bits contain the ID which is used to identify a particular CAN message. The *Identifier Field* portion is therefore used by network-nodes to establish if a received message is relevant to their own specific function; if not, nodes will just ignore the message. This field is also used for arbitration purposes which will be discussed in greater detail later. The

Remote Transmission Request bit is used to distinguish between a *Data Frame* and a *Remote Frame*. If this bit is *recessive* it indicates that the message is a *Remote Frame*, otherwise the frame is a *Data Frame*.

- **Control Field:** The *Control Field* is composed of six bits, the first of which is labelled the *IDE* (Identifier Extension) bit. In its *dominant* state it specifies that the message is a *Standard Data Frame*. Otherwise, this bit indicates that the message is an *Extended Data Frame*. A discussion outlining the principal differences between a *Standard Data Frame* and an *Extended Data Frame* will be described later. The following bit in this field is reserved and is defined to be *dominant*. The additional four bits that make up the *Control Field* are the *DLC* (Data Length Code) bits. The *DLC* is used to indicate the number of bytes of data (0 - 8) contained within the following *Data Field* of the message.

- **Data Field:** This field contains the actual information data; e.g. oil pressure, vehicle speed etc. The length of the *Data Field* is controlled by the contents of the *DLC*. The *Data Field* can contain anything from zero to sixty-four bits (0 to 8 bytes).

- **CRC Field:** The *CRC Field* (Cyclic Redundancy Check) is used to detect any possible transmission errors and contains a fifteen bit check sequence and a *CRC Delimiter* bit. A receiving node compares the *CRC* it has computed from the received frame to the information contained within the received message to establish if any errors have occurred.

- **ACK Field:** The *ACK Field* (Acknowledge) contains two bits. During the *ACK slot* bit a transmitting CAN node sends out a *recessive* bit. Any node on the network that has received the transmitted message without any errors acknowledges the correct message reception by sending a *dominant* bit back to the transmitting node. The other bit within the *ACK Field*, the *ACK delimiter* bit, must be *recessive* at all times and cannot be overwritten by a *dominant* bit.

- **End-of-Frame Field:** This field is used to signify the end of the CAN message. It consists of seven consecutive *recessive* bits.

The *Data Frame* described above is a *Standard Data Frame* as outlined by *Bosch* [2]. The CAN protocol describing a *Standard Data Frame* is entitled *CAN2.0A*. *CAN2.0B* is a subsequent protocol release which describes an *Extended Data Frame*. The fundamental difference between *standard* and *extended* frames is that an *extended* frame has the capacity to support a twenty-nine bit *Identifier Field* as opposed to a *standard* frame's eleven bit *Identifier Field*. Thus the *extended* frame format possesses a greater ID-range and relieves the system-designer from compromises with respect to defining well-structured identification schemes [2], [3]. Overall, the *extended* format is similar to the *standard* CAN frame. As discussed earlier, the two frames are distinguishable by the *IDE* bit within the *Control Field*. Within an *extended* frame the *Identifier Field* is separated into eleven and eighteen bit portions respectively. *CAN2.0B* is capable of receiving *CAN2.0A* messages; however this situation is not reciprocal, *CAN2.0A* does not support the reception of *CAN2.0B* messages.

CAN2.0A is used within the vast majority of automobile applications because an eleven bit *Identifier Field* more than adequately realises typical system requirements. Another reason for its employment within the majority of applications is the fact that it also requires less overhead and silicon space than *CAN2.0B* implementations. It has been established that the implementation of *CAN2.0B* is not always necessary and its employment is only necessary under certain circumstances [4].

Remote Frame

ISO11898 specifies that any node on a CAN network can send a *Remote Frame* which essentially is a request for information from another attached node. The transmission of a *Remote Frame* is analogous to asking a question. The node that has the "answer" will transmit a message containing the requested information to the node that sent the *Remote*

Frame [15]. The composition of a *Remote Frame* is identical to a *Standard Data Frame*; the only exception is that the *Remote Transmission Request* bit is transmitted in a *recessive* sate. In addition, the *DLC* portion of the *Control Field* contains zero to indicate that no data will be contained within the *Data Field*.

Error Frame

The CAN protocol allows any node on the network that detects a bus error to generate an *Error Frame*. An *Error Frame* is comprised of two fields; an *Error Flag Field* and an *Error Delimiter Field*. The content of the *Error Flag Field* depends on the error-status of the node that has detected the error. The *Error Delimiter Field* consists of eight *recessive* bits.

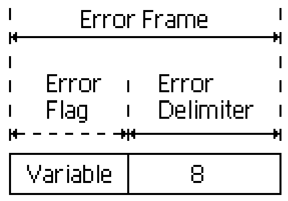


Figure 15: CAN Error Frame

Once an *Error Frame* is formed bus activity returns to normal and the node in which the error occurs attempts to re-transmit the aborted message.

Overload Frame

An *Overload Frame* is defined within the CAN protocol to allow a node to tell the network that it is occupied and is not yet ready to receive any further messages. It is comprised of an *Overload Flag* and an *Error Delimiter*.

2.3.2.2 Arbitration

ISO11898 allows simultaneous bus access from different nodes; this is known as *CSMA* (Carrier Sense Multiple Access). A node can proceed to transmit a message over the network if it detects that the bus is currently residing in an idle state i.e. no other node is currently transmitting [15]. The situation can arise however where two nodes attempt to transmit a message over the network simultaneously. Consequently a method of arbitration is employed within the *Data Link Layer* to establish which node may continue its transmission.

Many techniques exist within network topologies to implement arbitration [11], however the CAN protocol stipulates that the *CSMA/CD+AMP* (Carrier Sense Multiple Access with Collision Detection and Arbitration on Message Priority) technique be used [3], [16]. This arbitration methodology involves determining the priority of messages to establish which node may proceed with transmission. A message with a low binary value in its *Identifier Field* will have a high priority based on the *Wired-AND* logic (a *dominant* bit overwrites a *recessive* bit) discussed previously.

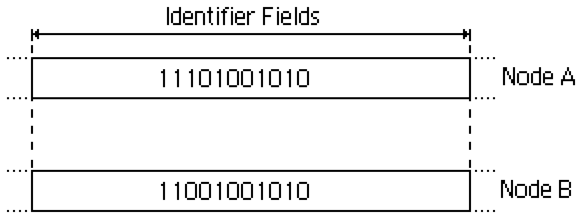


Figure 16: Arbitration Based on the *Identifier Fields* of Two Nodes

For the diagram above *Node B* has a higher priority over *Node A* because it's *Identifier Field* has a lower binary value than that of *Node A*. Subsequently if *Node B* and *Node A* both attempt to transmit a message concurrently *Node A* will loose arbitration and *Node B* can proceed with its message transmission.

2.3.2.3 Error Confinement

As CAN was initially developed for use within the automotive environment the protocol had to employ a methodology to efficiently process errors in order to acquire a wholesome share within the marketplace [9]. As described previously, *ISO11898* allows all CAN nodes to generate *Error Frames* upon detection of an error. The nodes are intelligent enough to assess whether an error is of a permanent or temporary nature. Each CAN node has three error states in which it can reside upon detection of a fault. The three states are as follows:

1. *Error Active*
2. *Error Passive*
3. *Bus Off*

Both *CAN2.0A* and the subsequent *CAN2.0B* stipulate that each CAN node should contain both a *TEC* (Transmit Error Counter) and a *REC* (Receive Error Counter) register in order to implement error confinement. The contents of these respective counters are incremented by a certain value each time the node transmits/receives an erroneous frame. Successful transmission and reception of message frames decrement the contents of the two counter registers [3].

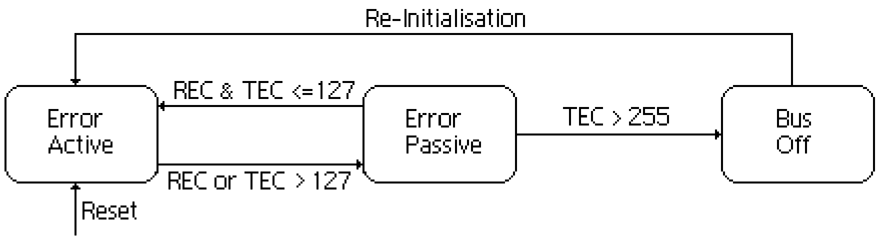


Figure 17: Error State Diagram for a CAN Node

The *Error Active* state is the typical state in which a network-node resides in after a reset condition. When the *TEC* and *REC* counters for a particular node contain a values less than 128 the node also resides in this *Error Active* state. In this condition the node is allowed to transmit and receive messages and is also allowed to transmit error frames (containing *Error Active* flags) without constraint.

If either the *TEC* or the *REC* register for an individual node contain values between 128 and 255 the node resides in the *Error Passive* state. In this state a node is free to transmit/receive message frames, although as soon as an *Error Passive* node transmits an erroneous frame further communication of messages is suspended and an *Error Passive* flag is sent out onto the CAN bus.

One aspect of the CAN protocol is that faulty nodes can withdraw themselves from the network automatically. The *Bus Off* state is entered into by a node when the contents of the *TEC* register are greater than 255; at this point all bus activity for this node terminates. To return to the *Error Active* state, and to reset the error counter values, the CAN node must be reinitialised.

2.4 Summary

This chapter examined and described various aspects of the CAN bus protocol. The main points to embrace are as follows:

- The CAN protocol considerably reduces cable length requirements within a system due to the fact that is a serial-based network topology.

- CAN is robust and reliable, therefore it is ideally suited for use within an automotive environment. It is used as the standard for vehicle communications by European automobile manufacturers; it is currently gaining mainstream acceptance in the United States.

- *ISO11898* only specifies the bottom two layers of the *OSI* model for CAN thus allowing a system-designer the freedom to customise a network to meet specific requirements.

- An appealing aspect of the CAN protocol is that it’s physical parameters, such as bit timing etc., are user programmable thus offering a designer control over bus lengths and timing.

- The CAN protocol defines its own highly efficient method for arbitrating between conflicting nodes in order to avoid transmission conflictions.

- The error handling capability of CAN allows a damaged node to withdraw itself from a system; thus damage to an individual node does not hamper the operation of the overall network.

Chapter 3 - Video Processing

3.1 Introduction

As video processing is required to visually represent automobile data within this application this chapter details the fundamentals relating to video data and its associated standards. The information given in this chapter is separated into the following main sections:

- An overview of the constitution of a generic video signal.
- A discussion outlining the most popular video standards.
- A summary detailing how chrominance is represented in a video signal and what steps are taken to efficiently utilise bandwidth.
- A synopsis of the *ITU-R BT.601* & *ITU-R BT656* digital video protocols.
- A brief explanation of how the *ITU-R BT.656* protocol is implemented using hardware.

3.2 Composition of a Video Signal

In its fundamental existence a video signal is comprised of a two-dimensional array of luminance (intensity) and chrominance (colour) data. The video signal is updated at a regular frame rate to ensure that perception of motion is conveyed to the human eye. The intensity information for each line of video is represented within the signal by a low-voltage waveform. In conjunction with this, timing information is embedded in the analogue signal to ensure that display-devices remains synchronised with the video signal [17], [18].

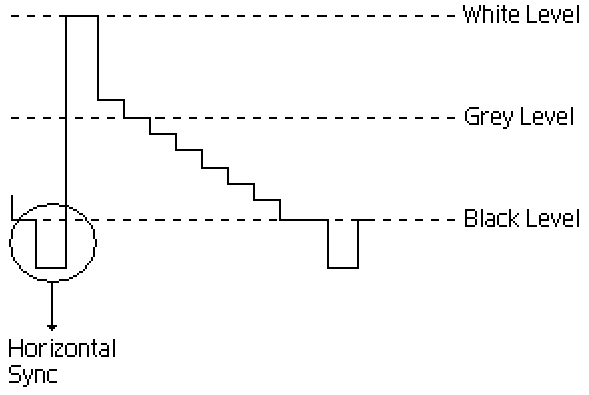


Figure 18: Luminance Component of a Elementary Analogue Video Signal [18]

For example, in a standard CRT (Cathode Ray Tube) television an analogue video signal modulates an electron-beam which results in the illumination of phosphorus on the screen. This practise is carried out in a left-to-right, top-to-bottom manner. As a result, it can be envisaged that a single video frame is comprised of multiple rows of data, which in turn are formed one-by-one on the screen [19].

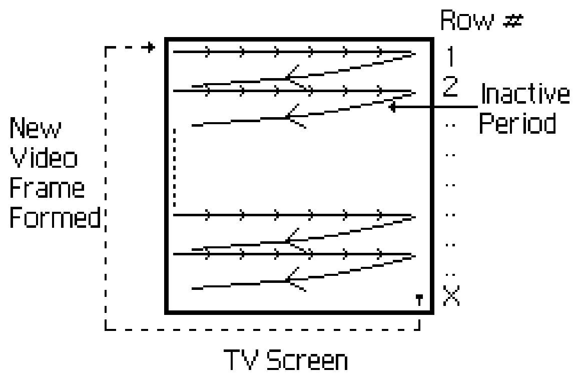


Figure 19: Numerous Rows of Data form a Single Video Frame

The embedded timing signals dictate when the electron-beam is active or inactive. The previous diagram illustrates that during the inactive period the electron-beam is allowed to retrace from right to left. This is so that it can begin to illuminate phosphorus on the next row, or move from the bottom right-corner to the top-left corner of the screen in order to begin formulation of the next video frame.

The synchronisation data embedded within a video signal and the timing relationships between them are shown in the following diagram.

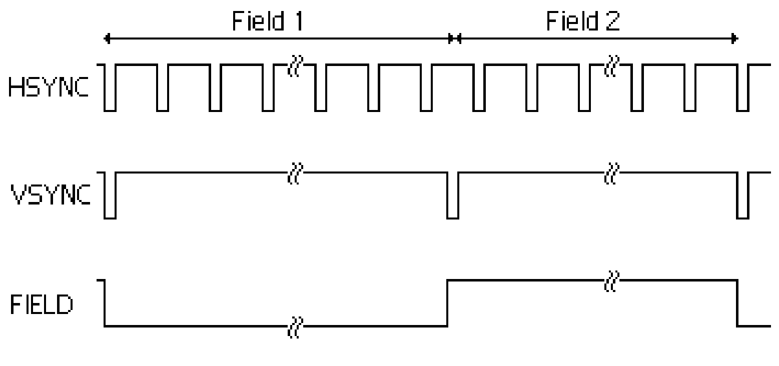


Figure 20: Synchronisation Signals Embedded within a Video Signal

The *HSYNC* waveform is the horizontal synchronisation signal and it is used to indicate the start of active video on each row of a video frame. *Horizontal blanking* is the inactive period during which the electron-beam retraces from the right side of the screen back over to the next row on the left side. *VSYNC* is the vertical synchronisation signal. It demarcates the start of a new video frame. *Vertical blanking* is the inactive period during which the electron-beam retraces from the bottom right-corner to the top-left corner of the display-screen in order to begin formulation of the next video frame [20].

The *FIELD* signal indicates, for an *interlaced* video scan, whether the field being displayed is “odd” or “even”. The *FIELD* synchronisation signal is not applicable to *progressive* scan video systems.

3.2.1 Interlaced and Progressive Scanning

What exactly constitutes an *interlaced* and *progressive* scan, and what is the difference between the two? In early analogue television systems bandwidth was a major restriction, i.e. systems only had the capacity to transmit so many lines of video per second. However, in order to seamlessly convey the perception of movement the video frames needed to be updated at an appropriate frequency ($\approx$ 50/60Hz).

A solution to this problem was realised by introducing the concept of *interlaced* video. Within this concept each video frame is split into two *fields*; one consisting of *odd* numbered row lines and the other composed of *even* numbered row lines. For an *interlaced* system the television displays the *odd-field* (*even-field*) first and then displays the *even-field* (*odd-field*). To the human eye, because of latencies, it appears that the entire frame (made up from the two *fields*) is being displayed simultaneously. This solution ensures that fluid motion is conveyed to the onlooker, while at the same time ensuring bandwidth restrictions are not violated [18], [19].

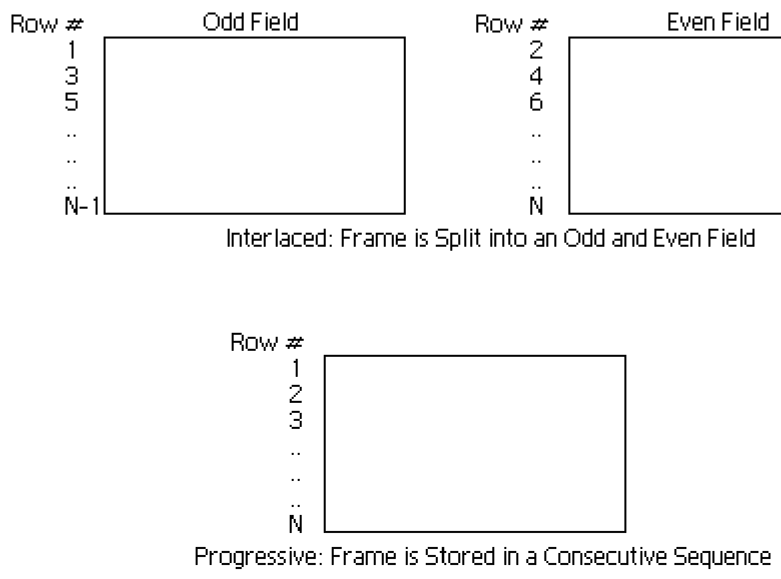

Row #
Odd Field
1
3
5
N-1
Row #
Even Field
2
4
6
N
Interlaced: Frame is Split into an Odd and Even Field
Row #
1
2
3
N
Progressive: Frame is Stored in a Consecutive Sequence

Figure 21: Interlaced vs. Progressive Video Scan [18]

In recent times, due to the advancements in television and video technologies, *progressive* scan video has become more widespread. From the previous diagram it can be seen that a *progressive* video frame is comprised of rows stored in a successive manner. The concept of *odd* and *even* fields does not apply to *progressive* scan systems as an individual frame is not split in two. *Interlaced* systems are still utilised, however because of the exceptional capabilities of modern television and video technologies *progressive* scan is increasingly prevalent, particularly in Western Europe [18], [20].

3.2.2 Video - Standards and Resolution

Numerous analogue video standards are employed worldwide. The primary difference between the various standards is found in the manner in which they encode luminance (intensity) and chrominance (colour) data. Universally speaking, two standards dominate - *NTSC* (National Television System Committee) and *PAL* (Phase Alternating Line).

NTSC is predominantly employed in North America and Asia, while *PAL* on the other hand is mainly utilised in Europe and South America. *PAL* is an enhancement on its older *NTSC* counterpart, improving on colour distortion prevalent with *NTSC*. *HDTV* (High

Definition Television) is the latest addition to the video standard realm. It is actually a digital video standard, as opposed to the other analogue standards previously mentioned, and it is forecasted to be the dominant standard in the future [20].

The fundamentals of *NTSC* and *PAL* are relatively similar; a QAM (Quadrature Amplitude Modulation) [21] sub-carrier relaying the chrominance (colour) data is added to a luminance (intensity) signal to form a composite video baseband signal. *NTSC* is typically implemented using *interlaced* scanning. It has a frame rate of approximately 30fps (Frames per Second); therefore *fields* are updated at 60fps. *PAL* is equilaterally utilised as an *interlaced* or *progressive* scan system. It has a frame refresh rate of approximately 50fps. Notice that the frame rates of *NTSC* (60fps) and *PAL* (50fps) coincide with the 60Hz and 50Hz frequencies of AC (Alternating Current) power in the United States and Europe respectively. This is no coincidence; this is a deliberate design ploy implemented to avoid visible interference upon a display-monitor [18], [20], [22].

The resolution of a video frame is measured in pixels and is defined as the product of the horizontal and vertical resolution. The horizontal resolution indicates the number of pixels on each row of a video frame, while the vertical resolution specifies how many horizontal lines are displayed to create the entire video frame.

Video Standard	**Horizontal Resolution (Pixels)**	**Vertical Resolution (Pixels)**	**Frame Resolution (Pixels)**
NTSC	720	480	345,600
PAL	720	576	414,720

Table 6: Frame Resolution - NTSC vs. PAL [22]

The preceding table illustrates that both *NTSC* and *PAL* possess equal horizontal resolutions. Yet *PAL* has a higher frame resolution than *NTSC* due to its superior vertical resolution. As a result a *PAL* frame represents a video frame with finer detail than *NTSC*. However the colour resolution of *NTSC* is greater than that of *PAL*.

3.2.3 Chrominance Representation

Numerous methodologies exist for representing chrominance within the video environment. Each individual system is suited to a particular application. For instance, some are designed for application with television systems, whilst others are used with computer-graphics displays. The most fundamental methodology for chrominance representation is the *RGB* (Red Green Blue) colour space system. The three primary colours are red, green and blue. When summed together in equal proportions they manifest white light.

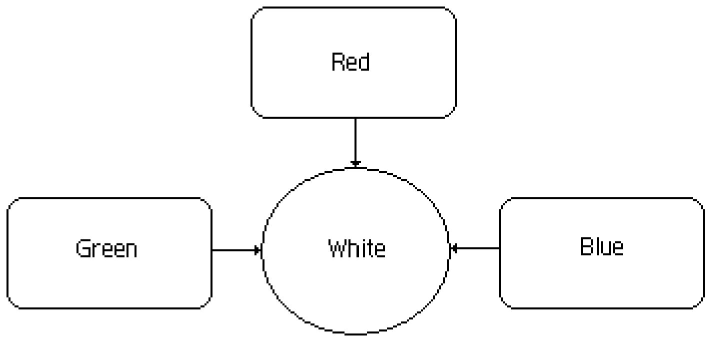


Figure 22: Formation of White Light using the Three Primary Colours

The *RGB* system combines various quantities of the three primary colours to formulate any colour in the visible spectrum. Due to its relative simplicity the *RGB* scheme is the preferred methodology used for chrominance representation in computer-graphics systems [23], [24].

Luminance (intensity) is perceived in a non-linear fashion by the human eye. In addition, display-devices such as CRTs also display luminance in a non-linear manner. Coincidentally the eye's perception of luminance sensitivity is approximately converse to standard display-devices' output characteristics. For that reason video devices and algorithms pre-distort their *RGB* output stream. This is to counteract display-devices' non-linear luminance representation and to create a realistic model of how the eye perceives a video image in reality. Pre-distorted *RGB* values are referenced as *R'G'B'* [18].

Even though *RGB* is the natural technique for colour representation it is not appropriate for image-processing because each it's three components are highly correlated with one another. Consequently other chrominance schemes that are more efficient and highly-uncorrelated have evolved; an example of which is the *YCbCr* system. The *YCbCr* system contains a single luminance value and two chrominance components. The separation of luminance and chrominance data results in more efficient use of image-processing bandwidth. The luminance, *Y*, and chrominance components, *Cb* and *Cr*, are mathematically derived from *R'G'B'* values as seen below [25]:

$$Y = (0.299)R' + (0.587)G' + (0.114)B' \qquad \textit{Eq. 3.1}$$

$$Cb = -(0.168)R' - (0.33)G' + (0.498)B' + 128 \qquad \textit{Eq. 3.2}$$

$$Cr = (0.498)R' - (0.417)G' - (0.081)B' + 128 \qquad \textit{Eq. 3.3}$$

, where *R'*, *G'* and *B'* are pre-distorted *RGB* values,
Y is the luminance component,
Cb and *Cr* are chrominance components.

3.2.3.1 Chrominance Sub-Sampling

The human eye is more sensitive to luminance variation than it is to chrominance difference. *YCbCr* takes advantage of this as it pays more attention to luminance (*Y* component) than chrominance (*Cb* and *Cr* components). Thus chrominance values can be sub-sampled resulting in considerable bandwidth savings.

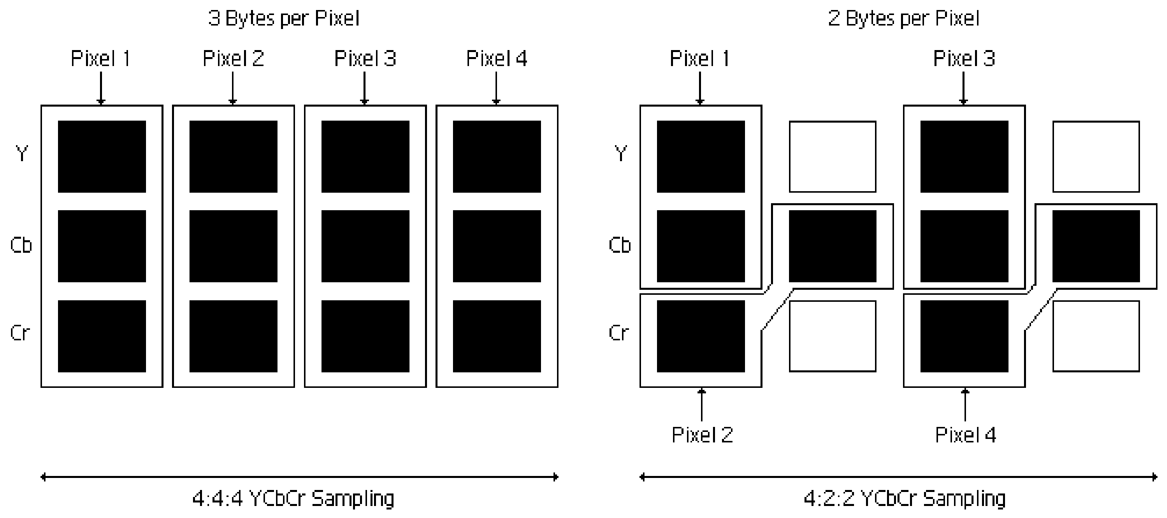


Figure 23: Chrominance Sub-Sampling

From the preceding diagram a full-bandwidth pixel-stream is represented by the *4:4:4 YCbCr* signal. The first number is always "4" and corresponds to the relationship between the sampling frequency of the luminance component and the particular analogue standard (i.e. *NTSC* or *PAL*) sub-carrier frequency. The second number represents the ratio of luminance to chrominance in a given horizontal row; in this case all chrominance components are sampled fully hence this number is "4". The last number illustrates the vertical luminance/chrominance relationship; if no sub-sampling takes place this number is also "4". If the chrominance components of the full-bandwidth signal are sub-sampled by a factor of two horizontally a *4:2:2 YCbCr* signal is obtained. This means that there are four luminance components for every two chrominance values on a particular video row [23]. The acquisition of a *4:2:2* signal results in only a minute distortion to the quality of a video image when compared to a *4:4:4* signal, yet a bandwidth saving of 33% is yielded. Hence sub-sampling is extremely efficient.

3.3 Digital Video

So far only analogue video has been discussed. Since the mid-1990s digital video has become prevalent, primarily due to mass improvements in internet infrastructure. This in turn has lead to an increase in consumers' demands for media-streaming. Digital video holds numerous advantages over its analogue counterpart. For instance, the SNR (Signal-to-Noise Ration) achievable with digital streams is much greater than that of analogue video. In addition, digital video utilises bandwidth more efficiently as several digital channels are compressible into a single analogue channel.

Fundamentally speaking, the construction of a digital video stream involves the sampling and quantisation of existing analogue video. The sampling process involves dividing an analogue image into a grid-like structure and assigning relative amplitude values to each grid-portion based on the intensities of colour-space components in each grid-region. The quantisation process involves determining the discrete amplitude values to assign during the sampling process. 8-bit video is common for consumer applications; a value of 0 is assigned to the darkest portions (black), while a value of 255 is assigned to white portions.

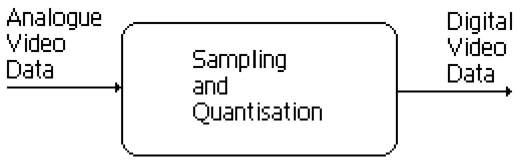


Figure 24: Digitisation of Analogue Video Data

To some degree the introduction of digital video has lead to standardisation between the *NTSC* and *PAL* architectures. The *ITU* (International Telecommunications Union) has defined digital video standards, *ITU-R BT.601* and *ITU-R BT.656*, which are focused towards achieving a large degree of cohesion between *NTSC* and *PAL* so that they can both share the same coding formats [18], [26].

3.3.1 ITU-R BT.601 & ITU-R BT.656

ITU-R BT.601 and *ITU-R BT.656* together define a practice that allows different video system components and standards to interoperate. The *ITU-R BT.601* standard describes the fundamentals of the video digitisation process, while *ITU-R BT.656* defines how *ITU-R BT.601* is actually physically implemented.

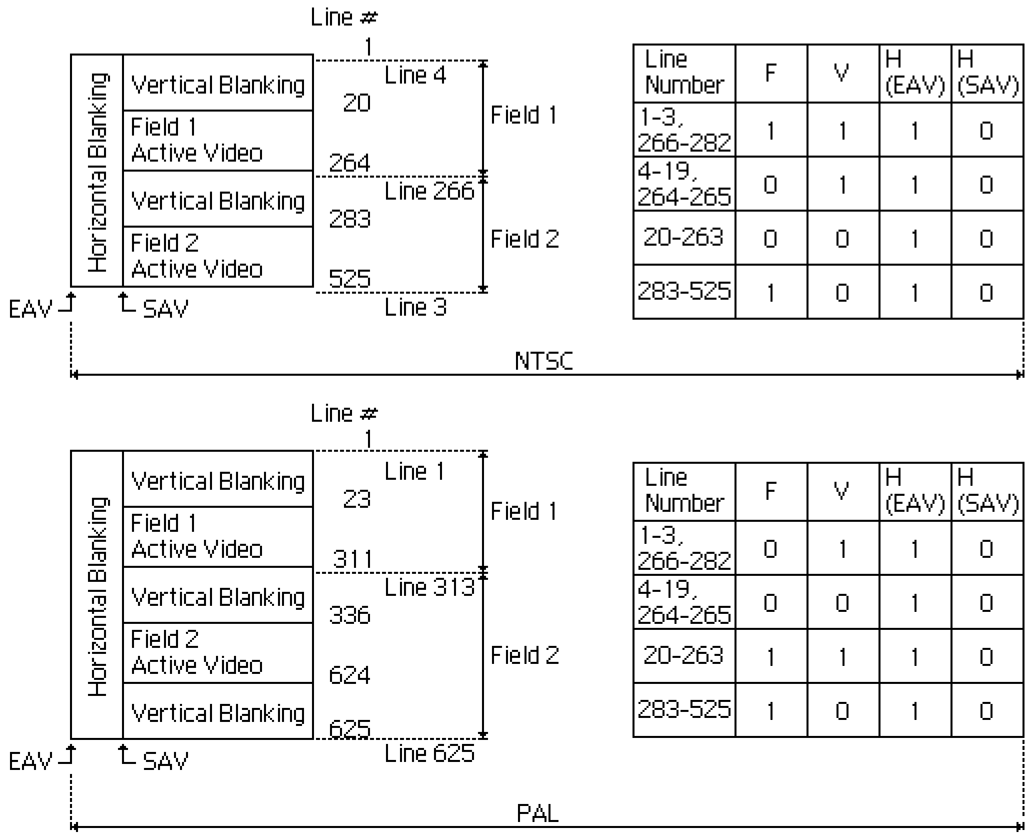
ITU-R BT.601 specifies that *4:2:2 YCbCr* colour-spacing is employed to achieve bandwidth efficiencies as outlined earlier. The protocol also stipulates that standard synchronisation signals (*HSYNC*, *VSYNC,* and *FIELD*) be used to demarcate the boundaries of active video regions. Within this standard each pixel component (*Y*, *Cb*, or *Cr*) is quantised to either 8 or 10-bits. 8-bit quantisation is more practical for implementation purposes as processors can efficiently handle octal multiples.

ITU-R BT.601 specifies that both *NTSC* and *PAL* comprise the same horizontal resolution (i.e. 720 pixels of active video per line). On the other hand, a difference exists in terms of vertical resolution. A 30fps *NTSC* video stream has a vertical resolution of 525 lines; this is in comparison to 625 lines for a 25fps *PAL* frame [23], [24], [27].

As mentioned above *ITU-R BT.656* identifies the physical interfaces and data streams needed to implement the *ITU-R BT.601* standard. One of the main advantages realised from the use of the *ITU-R BT.656* digital protocol is that all timing signals are embedded in the data stream. This therefore means that no additional hardware lines are required for synchronisation purposes. *ITU-R BT.656* defines both a bit-serial and bit-parallel mode. The implementation of the bit-serial mode can be rather complex and is not realisable on many systems. For that reason, this discussion will only refer to the bit-parallel mode [23], [28].

3.3.1.1 ITU-R BT.656 - Frame Partitioning & Data Stream Characteristics

The *ITU-R BT.656* frame partitioning requirements for both *NTSC* and *PAL* are seen below.



Line Number	F	V	H (EAV)	H (SAV)
1-3, 266-282	1	1	1	0
4-19, 264-265	0	1	1	0
20-263	0	0	1	0
283-525	1	0	1	0

Line Number	F	V	H (EAV)	H (SAV)
1-3, 266-282	0	1	1	0
4-19, 264-265	0	0	1	0
20-263	1	1	1	0
283-525	1	0	1	0

Figure 25: ITU-R BT.656 Frame Partitioning

As mentioned earlier, the *HYSNC* (*H*), *VSYNC* (*V*) and *FIELD* (*F*) synchronisation signals are sent as an embedded portion of the video stream. This data is transmitted as a series of bytes that form a control word. The *SAV* (Start of Active Video) and *EAV* (End of Active Video) respectively demarcate the beginning and end of relevant video data for every line/row; thus *horizontal* blanking occurs during this period. *SAV* occurs on a 1-to-0 logic level transition of *HSYNC*, while *EAV* occurs on a 0-to-1 transition of *HSYNC*. *Vertical* blanking occurs when $V = 1$. A *field* of video begins on a logic transition of the *F* bit. An *odd-field* is represented with $F = 0$, while an *even-field* is denoted by $F = 1$. If *progressive* video scanning is employed no distinction is made between *fields*. Thus it is seen that an entire field of video is comprised of active video, *horizontal* blanking, and *vertical* blanking [18], [28].

The *SAV* and *EAV* codes are shown in greater detail below:

	8-bit Data (D_7 = MSB, D_0 = LSB)							
	D_7	**D_6**	**D_5**	**D_4**	**D_3**	**D_2**	**D_1**	**D_0**
Preamble	1	1	1	1	1	1	1	1
	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0
Control Byte	1	F	V	H	P_3	P_2	P_1	P_0

Table 7: 8-bit SAV & EAV Preamble Codes

From the preceding table it is seen that a defined preamble, consisting of 3-bytes (0xFF, 0x00 and 0x00), is followed by a control-byte. This control-byte contains four bits (P_3, P_2, P_1, and P_0) for error detection and correction in addition to the *H*, *V*, and *F* bits. The bit definitions are as follows [18], [26]:

F = 0 for Field 1	*V* = 1 during Vertical Blanking	*H* = 0 at *SAV*	P_3 = *V* XOR *H*	P_1 = *F* XOR *V*
F = 1 for Field 2	*V* = 0 when not in Vertical Blanking	*H* = 1 at *EAV*	P_2 = *F* XOR *H*	P_3 = *F* XOR *V* XOR *H*

Table 8: Bit Definitions for ITU-R BT.656 Preamble

The following diagram illustrates the composition of an *ITU-R BT.656* bit stream for a single line/row of video data. The *SAV* encompassing the defined preamble (0xFF, 0x00, and 0x00) along with the control byte (containing *H*, *V*, *F* and error detection/correction bits) indicates the beginning of a new line/row of video data. The active video information then follows in *4:2:2 YCbCr* format. Recall that *ITU-R BT.601* specifies that both *NTSC* and *PAL* contain 720 pixels of active video per line. As the *4:2:2* format is employed there are twice as many luminance components than chrominance values resulting in a total of 1440 bytes of active video data in a given line/row. The occurrence of an *EAV* completes the formation of a current line/row allowing the construction of the next line/row to begin [20].

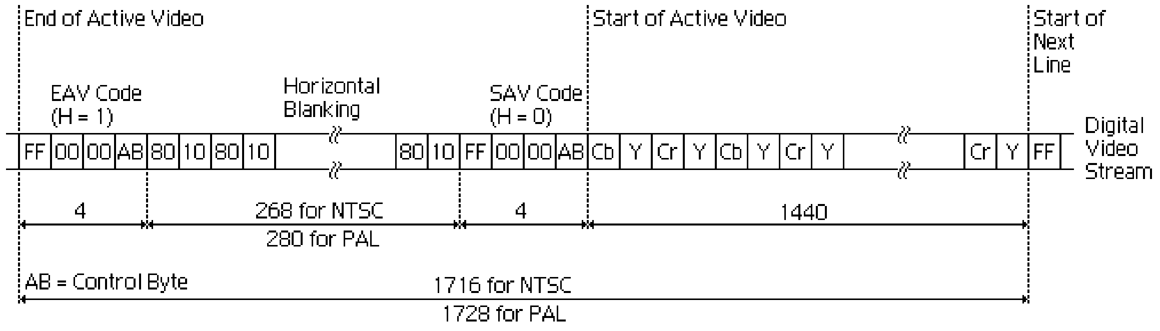


Figure 26: ITU-R BT.656 Video Data Stream [29]

3.3.1.2 ITU-R BT.656 Implementation

ITU-BT.656 is in essence a standard that is implemented in hardware through software initialisation. If video data is being transmitted a video-encoder IC is used to convert the digital *ITU-R BT.656* stream into an analogue signal for display upon a CRT (or other display device). It fundamentally acts as a digital-to-analogue converter converting the input digitised stream into standard analogue video standards like *NTSC* or *PAL*. Conversely speaking, if analogue video in the form of *NTSC* or *PAL* is being received a video-decoder IC is used to convert the input signal to an *ITU-R BT.656* video stream.

Video-encoder and decoder ICs interface to a processor giving a system-designer the control to program the ICs to meet whatever specific requirements a project may demand. The majority of video-encoder and decoder ICs support both *NTSC* and *PAL*. For this application the choice between *NTSC* and *PAL* is irrelevant; the main focus is on displaying vehicle data regardless of the analogue video format utilised.

3.4 Summary

This chapter discussed the fundamentals of video data and outlined standards prevalent in the video environment. The key points to note are as follows:

- Any video signal primarily consists of luminance and chrominance data.
- Timing information is embedded within the video data to ensure that display-devices remain synchronised with the input signal.
- *NTSC* and *PAL* are the predominantly used analogue video standards.
- Chrominance sub-sampling is used to efficiently utilise bandwidth.
- *ITU-R BT.601* and *ITU-R BT.656* are digital video standards that are designed to allow interoperability between video components and standards.
- The *ITU-R BT.656* protocol is a standard implemented using video-encoder and decoder ICs and is software configurable.

Chapter 4 - Selection of a Processor

4.1 Introduction

Now that a review of CAN networking and video processing has been carried out the next step is to select a suitable processor for utilisation within this system. This chapter discusses the selection of an adequate processor. The information given is divided into numerous sections as outlined below:

- An outline of the main factors taken into consideration when selecting an appropriate intelligent-device for this system design, with particular attention being paid to video processing and CAN capabilities.

- A comparison of a number of different processors is discussed under each of the main factors taken into consideration.

- A summary outlining all of the components and the selection of a particular processor for use within this system.

4.2 Key Considerations

When choosing an appropriate processor for operation in this project sizeable consideration must be given to a number of key factors. Bearing in mind that this is a system designed for operation within an automotive setting, a suitable device has to be able to operate sufficiently inside such a harsh environment. The correct processor for this particular design must also possess CAN and video capabilities in order to meet the system requirements. The main deliberations for selection of a fitting device are outlined below [18], [30]:

- Automotive Environment Specifications
- Video Processing Capabilities
- CAN Handling Ability
- Clock Rates & Power Consumption
- DMA - Direct Memory Access
- Programming Environment

The development boards below contain suitable processors for completion of this project. They are evaluated under the headings outlined above to establish which is the most suitable for this system's synthesis.

- Freescale MPC5200 Lite5200 Evaluation Board [31], [32], [33], [34]
- Infineon TriBoard TC1796 [35], [36]
- Xilinx Spartan-3E Starter Kit [37], [38]
- Microchip dsPICDEM 1.1 Plus Development Board [39], [40]
- Analog Devices Blackfin ADSP-BF537 EZ Kit Lite [41], [42]

Each of these components is designed by their respective manufacturers for use in the automotive industry.

4.2.1 Automotive Environment Specifications

An automotive environment contains many hindrances such as EMI, humidity, noise, temperature extremes and vibrations [43]. These factors can have a detrimental effect on signals and devices inside a vehicle. As a result ICs and processor used within an automotive environment have to be able to withstand these factors. For instance, the typical temperature-range for automotive ICs is -40ºC to +125ºC [43] because components may have to operate under severe temperature extremes. Consequently, the processor chosen for use in this system must comply with standard automotive IC provisions.

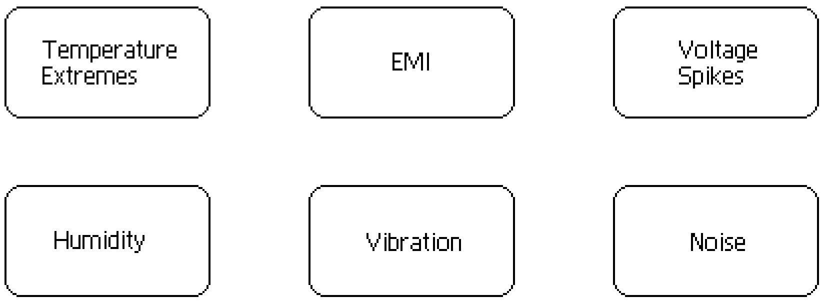


Figure 27: Harsh Factors Experienced inside an Automotive Environment

The following table outlines the ambient operating temperature-ranges for the five intelligent-devices chosen for evaluation.

Processor	Temperature Range (ºC)
Freescale MPC5200	-40 to +85
Infineon TC1796	-40 to +125
Xilinx Spartan-3E	-40 to +100
Microchip dsPIC30F6014A	-40 to +125
Blackfin ADSP-BF537	-40 to +85

Table 9: Ambient Temperature Ranges of Components under Evaluation

As it can be seen, all of the components comply with typical automotive temperatures; however only the *Infineon TC1796* and *Microchip dsPIC30F6014A* processors operate over the full automotive temperature-range (-40 to +125ºC).

4.2.2 Video Processing Capabilities

The computational power of a processor employed with this system is crucial. A raw video signal for instance could be comprised of data operating in the region of tens of MBytes/s [18]. Consequently it is easy to envisage that an appropriate intelligent-device must possess the capabilities to handle such high rates of data throughput. 16 and 32-bit processors should hold enough power to fulfil this role.

In addition, a suitable device connects, with minimum hardware and software effort, to standard video-encoder ICs that support the *ITU-R BT.656* protocol in order to simplify interfacing requirements. The standard hardware component of a processor conventionally used to facilitate such an interface is a *PPI* (Parallel Peripheral Interface) port. This is due to the fact that the transfer of video in *ITU-R BT.656* parallel mode is more efficient than a serial transfer. Thus, a suitable device preferably contains a *PPI* port for transfer efficiencies.

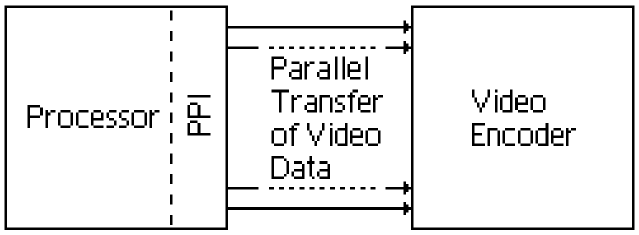


Figure 28: Video Data Transferred in Parallel between Processor & Video-Encoder

To simplify interfacing requirements even further, video encoding ICs should be located upon the development apparatus, or form part of a compatible A/V (Audio/Video) daughter board.

4.2.2.1 Freescale MPC5200 Lite5200 Evaluation Board

The 32-bit *Freescale MPC5200* processor is extremely powerful and is more than capable of adequately handling video data. It can perform 760MIPS (Millions of Instructions per Second) at a 400MHz clock frequency; which gives an indication of its processing power [33].

The *MPC5200* contains a *PCI* (Peripheral Component Interconnect) interface which allows for the connection of different varieties of peripherals to the development board. The *PCI* is in essence a 32-bit configurable address/data bus suited for high data-rate transfers [32]. Consequently, the *PCI* is configurable as a *PPI* port and thus it can be used to interface with an attuned video-encoder IC. However it would take a great deal of effort, both in terms of hardware and software, to interface these components.

A much simpler solution would be realised if the *Lite5200* kit had a compatible A/V daughter board that supports *ITU-R BT.656*. Nonetheless, no such A/V extension board is available for the *Lite5200*.

4.2.2.2 Infineon TriBoard TC1796

This development board incorporates the 32-bit *TriCore TC1796* processor. This device can, with relative ease, support the processing of video data. For instance, an illustration of its power can be seen in the fact that it can operate at a 150MHz clock frequency over its entire temperature range [35].

The *TC1796* contains a 16-bit *PPI* port which would facilitate in the transfer of *ITU-R BT.656* parallel data [35]. However, like the *Freescale Lite5200* kit, there is no A/V extension board available for this particular device. Once more, as a result of this, it would take a large endeavour, both in terms of hardware and software, to interface the *TriBoard TC1796* development board with suitable hardware components that offer *ITU-R BT.656* support.

4.2.2.3 Xilinx Spartan-3E Starter Kit

FPGAs (Field Programmable Gate Array) are highly configurable hardware devices. They consist of a vast array of logic-gates and modules which can be configured to meet any specification required by a designer [44]. Their basis of operation involves the concept of "parallel-processing"; which essentially means that multiple data blocks can be processed concurrently. Conversely, standard processors can only process data sequentially.

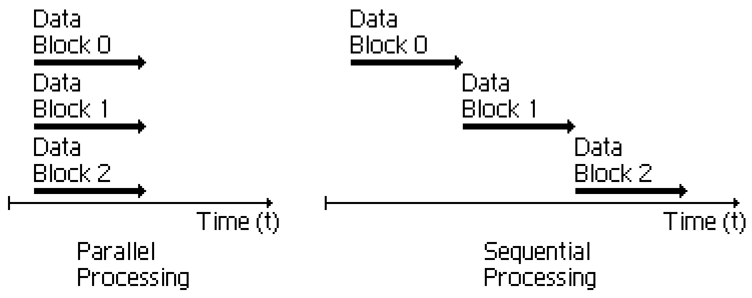


Figure 29: Parallel vs. Sequential Processing

Even though the *Xilinx Spartan-3E Starter Kit* uses a relatively modest 50MHz clock-signal to derive it's timing the concept of parallel-processing results in this FPGA being an extremely fast device [37], [38]. Therefore this component could more than adequately handle a video stream.

However, due to the fact that FPGAs are comprised from an array of configurable hardware blocks designers have to develop all hardware components from first principles. For example, for this particular project a *PPI* port is desirable to facilitate the efficient transfer of video data. This means that a designer would have to construct a *PPI* port from gate-level up. Therefore it is easy to envisage that the development time for a certain application designed to run on a FPGA could be relatively longer than that of a standard processor.

4.2.2.4 Microchip dsPICDEM 1.1 Plus Development Board

The 16-bit *dsPIC30F6014A* device incorporated onto the *dsPICDEM 1.1 Plus Development Board* can operate at a maximum of 30 MIPS [39]. This is relatively slow when compared to the other processors. The device can utilise a PLL to increase the clocking frequency. In spite of this, the maximum clock rate achievable using the PLL is not adequate to competently support video data processing.

The *dsPIC30F6014A* processor contains a *PPI* port, which again facilitates the efficient transfer of video data. Yet, like the other components discussed so far, it does not have an A/V daughter board to simplify interfacing requirements.

4.2.2.5 Analog Devices Blackfin ADSP-BF537 EZ Kit Lite

The *Blackfin ADSP-BF537* is an example of a *convergent* processor. It combines a 16-bit DSP (Digital Signal Processor) and a 32-bit microcontroller onto a single IC. It amalgamates the best qualities of a DSP and a microcontroller making it an extremely powerful device; thus it is sufficiently equipped to deal with video data.

This processor contains a *PPI* port which again is advantageous in efficient data transfers. In fact the *PPI* port of the *Blackfin* has been designed with video processing in mind. In addition, the *Blackfin ADSP-BF537 EZ Kit Lite* development board has a compatible A/V daughter board. This daughter board contains video-encoder ICs and sockets for interfacing with display-devices. Obviously, use of the *Blackfin EZ Kit Lite* and its daughter board would minimise the interfacing efforts required for this project.

4.2.2.6 Video Processing Capabilities - A Summary

Development Board	**Video Processing Speed Capability**	**Parallel Interfacing Capability**	**Compatible A/V Daughter Board**
Freescale MPC5200 Lite5200	Sufficient	Achievable through *PCI* configuration	None Available
Infineon TriBoard TC1796	Sufficient	Yes	None Available
Xilinx Spartan-3E Starter Kit	Sufficient (due to Parallel Processing feature of FPGAs)	Must be developed by Designer	None Available
Microchip dsPICDEM 1.1 Plus Development Board	Inadequate	Yes	None Available
Analog Devices Blackfin ADSP-BF537 EZ Kit Lite	Sufficient	Yes	Yes

Table 10: Summary of Video Processing Capabilities of Reviewed Devices

From the preceding table it can be seen that the *MPC5200*, *TriBoard TC1796* and *Blackfin ADSP-BF537* are sufficiently equipped to process video data. The *Blackfin* is however the processor of choice, in terms of video processing capabilities, due to the fact that it has a compatible A/V board which minimises interfacing efforts.

4.2.3 CAN Handling Abilities

As outlined in *Chapter 2*, the CAN protocol is employed as the standard for vehicle communications within Europe by automobile manufacturers. Subsequently an adequate intelligent-device preferably contains an integrated CAN Controller in order to reduce overhead and propagation delay. This would obviously lead to an overall reduction in system cost. The alternative to this is to use a peripheral CAN Controller interfaced to a processor via a *SPI* link [45].

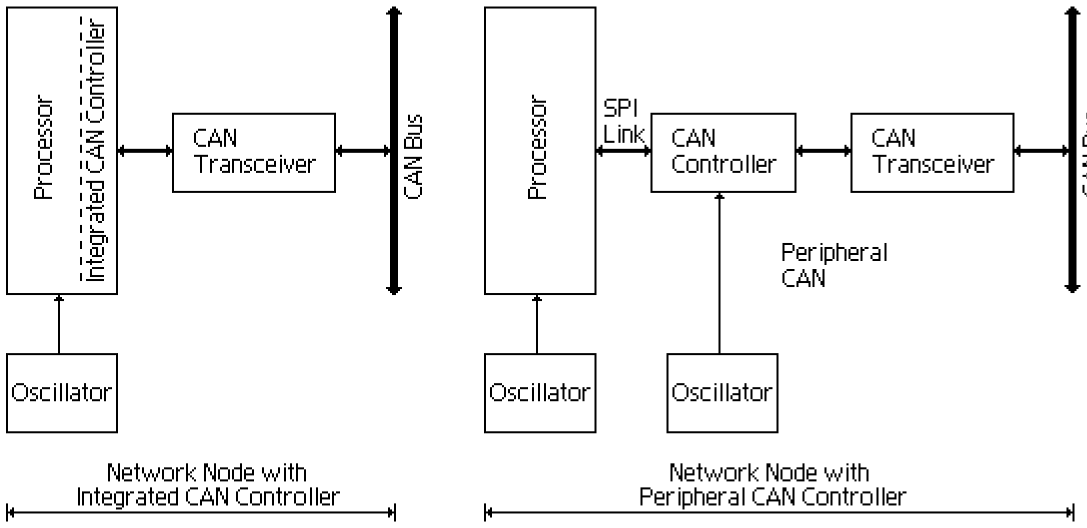


Figure 30: Integrated vs. Peripheral CAN Controller within a Network Node

As shown in the preceding diagram, the use of a peripheral CAN Controller leads to an increase in the number of components required to implement a network node.

4.2.3.1 CAN Handling Abilities of Processors under Investigation

The following table illustrates the CAN handling abilities of the intelligent-devices examined in this discussion.

Processor	**Integrated CAN Controller**	**Total Number of RX/TX Buffers**
Freescale MPC5200	Yes	8 RX 6 TX
Infineon TriCore TC1796	Yes	4 RX/TX (Programmable Bi-Directional Buffers)
Xilinx Spartan-3E FPGA	No	Not Applicable
Microchip dsPIC30F6014A	Yes	3 RX 3 TX
Analog Devices Blackfin ADSP-BF537	Yes	8 RX 8 TX 16 Configurable Buffers

Table 11: Overview of CAN Handling Abilities of Scrutinised Processors

From the preceding table it can be seen that all of the devices, with the exception of the *Spartan-3E*, contain an integrated CAN Controller.

As mentioned previously, FPGAs are user-configurable hardware devices. The development of a CAN Controller upon a FPGA would be an extremely time-consuming process, primarily because the entire mechanics of a CAN Controller would need to be described at fundamental gate-level. An alternative to the manual-development of a CAN Controller is the purchase of a CAN IP (Intellectual Property) footprint. This essentially means that a system-designer purchases a footprint of a CAN Controller developed by some other party. The footprint is simply "dropped" onto the FPGA, resulting in part of the device operating as a CAN Controller. Conversely, the retail price of a CAN Controller from one particular vendor is in the region of $15,000. It is clear to see that this is an incredibly costly alternative. For that reason, the use of the *Xilinx Spartan-3E* to implement the CAN protocol is not practical on this occasion.

Again with reference to the previous table, the four processors containing an integrated CAN Controller enclose reduced overhead both in terms of hardware and software. These four devices are sufficiently capable of handling CAN transfers for this particular project. Each of the four processors contains numerous RX/TX (Receive/Transmit) buffers. A CAN buffer acts like a mailbox for a particular CAN message. The more buffers a device contains, the more efficient it is at managing the reception/transmission of CAN messages. Therefore the *Blackfin ADSP-BF537* is the most efficient processor, in terms of CAN handling, as it contains a total of thirty-two message buffers.

4.2.4 Clock Rates & Power Consumption

In order to process video in real-time it is desirable to select a processor that operates at a relatively high clock rate. However, a high clock rate results in greater power consumption. Therefore the system-designer must take this trade-off into consideration when selecting an intelligent-device to fulfil the system's synthesis. Ideally, an adequate component contains an adjustable clock frequency feature; i.e. the clock frequency applied can be varied in real-time during program operation. This leads to reduced power consumption. In addition, the selected intelligent-device contains power adjustment features to reduce overall power consumption.

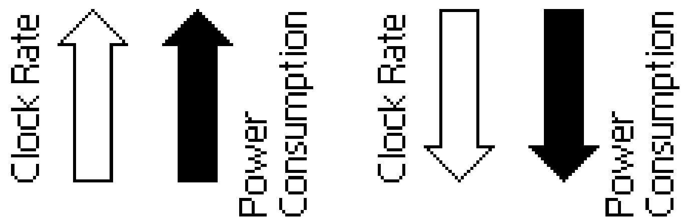


Figure 31: Relationship between Clock Rate & Power Consumption

4.2.4.1 Summary of Clock Rates & Power Consumption

The following table illustrates the clock rates and power consumption features of the processors examined in this discussion.

Processor	**Recommended Maximum Clock Frequency (MHz)**	**Real-Time Clock Adjustment Capabilities**	**Power Adjustment Features**
Freescale MPC5200	400	No	Yes
Infineon TriCore TC1796	150	No	Yes
Xilinx Spartan-3E FPGA	Not Applicable	Not Applicable	Yes
Microchip dsPIC30F6014A	160	No	Yes
Analog Devices Blackfin ADSP-BF537	600	Yes	Yes

Table 12: Synopsis of Clock & Power Adjustment Features for Examined Processors

From the preceding table it can be concluded that all of the devices with the exception of the *dsPIC30F6014A* possess adequate clocking abilities to process the high data-rates associated with video. As mentioned previously, the concept of parallel-processing fundamental to FPGAs results in the *Xilinx Spartan-3E* containing ample strength to process video data sufficiently. As seen all of the processors encompass power adjustment features.

The processor of choice in terms of clock rate and power consumption is the *Blackfin ADSP-BF537*. It is the device with the highest operating frequency. In addition, the *Blackfin* possesses the ability to adjust its clock frequency in real-time making it an attractive device for utilisation.

4.2.5 DMA - Direct Memory Access

The core of any processor is responsible for carrying out many operations. Parts of the core's duties involve managing data transfers between internal/external memory registers and peripherals. When large quantities of data are being transferred frequently a processor's core can become completely embroiled with the task of information transfer; thus preventing it from carrying out other necessary duties.

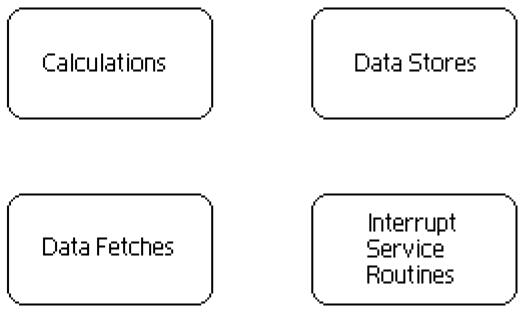


Figure 32: Core Responsibilities

DMA (Direct Memory Access) is a technique utilised to ensure efficient data-movement and relieves an intelligent-device's core from memory transfers so that it can perform other operations. An integrated DMA controller is delegated data-movement responsibilities by the processor's core, and once empowered the controller can independently manage data-transfers [18], [46].

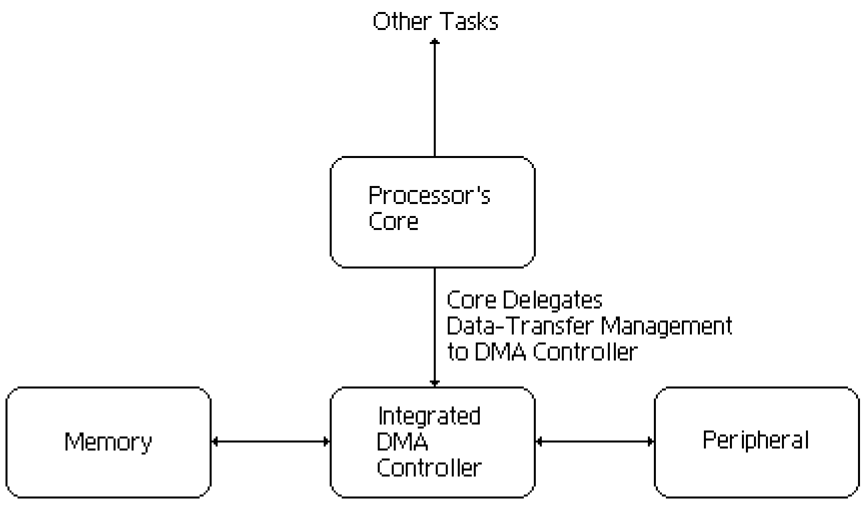

Other Tasks
Processor's Core
Core Delegates Data-Transfer Management to DMA Controller
Memory
Integrated DMA Controller
Peripheral

Figure 33: Typical DMA Flow

The presence of DMA in an application such as this is vital due to the fact that video-information is being transferred at high data-rates. If DMA is not present the core of a selected processor would essentially be congested by the constraint of having to read a data sample every time one becomes available. For that reason, the processor selected to implement this application must boast DMA competence.

4.2.5.1 DMA Competence of Evaluated Processors

The following table illustrates the DMA competence of the components examined in this discussion.

Processor	DMA Competence
Freescale MPC5200	Yes
Infineon TriCore TC1796	Yes
Xilinx Spartan-3E FPGA	No
Microchip dsPIC30F6014A	No
Analog Devices Blackfin ADSP-BF537	Yes

Table 13: Overview of DMA Competence of Inspected Processors

From the preceding table it can be seen that the *MPC5200*, *TriCore TC1796* and *ADSP-BF537* could be used in the synthesis of this system as they each contain DMA components. The additional two devices under scrutiny in this discussion do not possess any DMA functionality, and as a result would not support the efficient transfer of video-data. However, a DMA component could be constructed on the *Spartan-3E*, but again this would be a time-consuming process.

4.2.6 Programming Environment

The programming environment of an intelligent-device can encompass the language(s) supported by its compiler(s), and the ease in which the component may be reprogrammed. The programming environment of a specific choice of processor is imperative when selecting it for use within an application. Most development interfaces offer a system-designer the choice of using a high-level programming language (*C*, *C++*) or assembler to develop software on the device. Many processors can be compiled using royalty-free software packages, while others require specific compilers typically designed by the particular device's manufacturer. Nowadays, components can usually be re-programmed in-circuit with minimum effort using a USB (Universal Serial Bus) or alternative interface. The ideal processor selected for use within this project contains a user-friendly programming setting which minimises overhead and reduces needless complications.

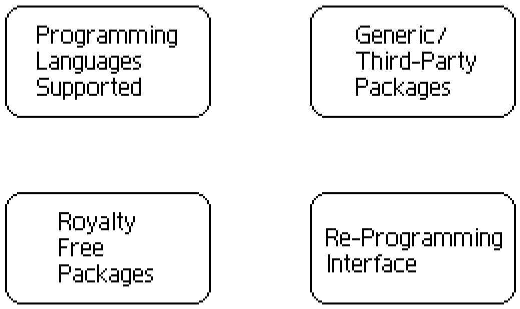

Programming Languages Supported
Generic/ Third-Party Packages
Royalty Free Packages
Re-Programming Interface

Figure 34: Factors within a Programming Environment

4.2.6.1 Freescale MPC5200 Lite5200 Evaluation Board

The development environment for this particular intelligent-device is relatively broad; numerous compilers are available from various vendors. Options exist for a *Macintosh*, *Linux* or *Windows* platform. For instance, *Freescale* offer their *CodeWarrior* interface tool for use in either a *Linux* or *Windows* setting. *CodeWarrior* allows a system-designer the freedom to cultivate software upon the *MPC5200* processor using *C*, *C++* or assembly. Third-party vendors such as *QNX* and *Green Hills* also offer development suites for this device [34]. Therefore a system-designer has a wide selection range to choose from when using this particular development-board. The *Lite5200* evaluation kit utilises a USB interface to simplify the programming-process. Overall, the programming environment for the *Freescale Lite5200* is user-friendly and extensive so particular preferences can be satisfied.

4.2.6.2 Infineon TriBoard TC1796

Infineon do not manufacture a development tool for their *TriBoard TC1796*. However, like the *Lite5200*, many third-party options exist for both the *Linux* and *Windows* platforms [47], [48]. The software package developed by *Altium* is the unofficial standard industry tool for the *TC1796* [47]. A royalty-free *GNU C/C++* programming option is also available for the *TriBoard TC1796* [49]. The *TC1796* incorporated onto this *Infineon* development board can be re-programmed via a USB interface. In general, the programming environment for the *Infineon TC1796*, like the *Freescale Lite5200*, provides a vast array of options and the particular interface tool chosen depends on the preferences of a system-designer.

4.2.6.3 Xilinx Spartan-3E Starter Kit

The *Spartan-3E*, like all FPGAs, is programmed using *VHDL*[1]. *VHDL* is not a high-level programming language like *C/C++*. In addition *VHDL* is not software; it is a hardware

[1] *VHDL* stands for *VHSIC Hardware Description Language*. *VHSIC* is an abbreviation for Very High Speed Integrated Circuit.

description language. It is a list of configuration commands used to describe the behaviour of hardware internal to a FPGA [50].

Subsequently, if a FPGA is selected for use within a project the designer would require knowledge of *VHDL* specific to the chosen FPGA, in this case it would be *Xilinx VHDL*; otherwise a new learning-curve would have to be embarked upon. Consequently, as a result of timing-constraints it is not feasible for use in this particular project.

4.2.6.4 Microchip dsPICDEM 1.1 Plus Development Board

Microchip has developed its *MPLAB* development interface for use with the *dsPIC30F14A* device incorporated onto the *Plus Development Board*. At present *MPLAB* only supports the *Windows* platform. The *MPLAB* tool presents the system-designer with the option of using either *C* or assembly language to configure the processor accordingly [40]. *Microchip's MPLAB* is user-friendly as it is relatively straight-forward to use. In addition, the *Plus Development Board* contains a USB interface which simplifies the re-programming process.

4.2.6.5 Analog Devices Blackfin ADSP-BF537 EZ Kit Lite

Like the *Freescale* and *Infineon* options already discussed, the development environment for the *Blackfin* is relatively extensive. Third-party choices exist for both the *Linux* [51] and *Windows* platforms [52], [53]. *Analog Devices* has developed its *VisualDSP++* tool for use with the *Blackfin* processor. This development component allows a designer to configure the *Blackfin* using *C/C++*, assembler, or a combination of both. It incorporates an abundance of functions and drivers to facilitate in software development. The *Blackfin EZ Kit* development board interfaces to the chosen compiler via a USB link, thus minimising re-programming efforts.

4.2.6.6 Programming Environment - A Summary

Development Board	**Programming Languages Supported**	**Generic or Third-Party Packages**	**Royalty-Free Packages Available**	**Re-Programming Resources**
Freescale MPC5200 Lite5200	*C*, *C++*, Assembly	Both	No	USB
Infineon TriBoard TC1796	*C*, *C++*, Assembly	Both	Yes	USB
Xilinx Spartan-3E Starter Kit	*VHDL*	No	No	Parallel Interface
Microchip dsPICDEM 1.1 Plus Development Board	*C*, Assembly	Generic	No	USB
Analog Devices Blackfin ADSP-BF537 EZ Kit Lite	*C*, *C++*, Assembly	Both	Yes	USB

Table 14: Summary of Programming Environments of Analysed Components

From the preceding table, with the exception of the *Xilinx Spartan-3E Starter Kit* (because it is only configurable using *VHDL*), it is clear to see that all of the intelligent-devices under scrutiny offer a considerable variety in terms of programming environments.

4.3 Synopsis of Reviewed Processors

	Automotive Environment Specifications	**Video Processing Capabilities**	**CAN Handling Abilities**	**Clock Rates & Power Consumption**	**DMA**	**Programmin g Environment**
Freescale MPC5200	Sufficient	Sufficient	Sufficient	Sufficient	Sufficient	Excellent
Infineon TriCore TC1796	Excellent	Sufficient	Sufficient	Sufficient	Sufficient	Excellent
Xilinx Spartan-3E	Sufficient	Moderate/ Sufficient	Inadequate	Sufficient	Inadequate	Moderate
Microchip dsPIC30F14 A	Excellent	Inadequate	Sufficient	Inadequate	Inadeqaute	Sufficient
Blackfin ADSP-BF537	Sufficient	Excellent	Excellent	Excellent	Sufficient	Excellent

Table 15: Synopsis of Reviewed Processors

It is concluded from the preceding table that the highly-configurable *Xilinx Spartan-3E* FPGA falls short of use in this particular application. This is primarily as a result of timing constraints required to implement customisation on this device. The *Microchip dsPIC30F14A* is also insufficient for use in this synthesis as it lacks the processing strength necessary to meet the system's specifications.

The *Freescale MPC5200*, *Infineon TriCore TC1796* and *Blackfin ADSP-BF537* are sufficiently equipped for employment in this application's development. However the *Blackfin ADSP-BD537* is the processor of choice. This is due to a number of factors. Firstly, the *EZ Kit* development board upon which the *Blackfin ADSP-BF537* is incorporated has a compatible A/V daughter board which simplifies the fulfilment of *ITU-R BT.656* video processing. Also, the *Blackfin* is excellently equipped to deal with CAN efficiently as it contains thirty-two message buffers. In addition to this, the maximum clock frequency of the *ADSP-BF537* more than adequately supports real-time video processing. The programming environment for this component is broad, thus offering a designer an array of choices. The *VisualDSP*++ development tool has been chosen for use to develop software on the device as it contains an abundance of support functions and drivers as mentioned earlier.

The *Blackfin* is a relatively new processor and its popularity is increasing exponentially. Evidence of this is found in the fact that the open-source community has embraced the *Blackfin* with many support forums offering free-ware code and advice [51], [54], [55]. *Analog Devices* are continuously developing new device-drivers and support tools to aid in implementation of new technologies.

4.4 Summary

This chapter discussed the selection of an adequate processor to implement this application. The major points to behold are as follows:

- A number of key factors need to be taken into consideration when choosing a suitable intelligent-device for use in this system.

- Numerous processors are discussed under each of the main factors taken into consideration.

- The *Blackfin ADSP-BF537* adequately meets all of the key considerations, particularly in the area of video and CAN, and as a result is selected as the processor or choice for use in this system.

Now that a suitable processor had been selected for utilisation from a number of examined devices, based on the merits outlined within this chapter, the next step was to synthesis the system incorporating correct hardware and software methodologies.

SECTION II - SYSTEM SYNTHESIS

Chapter 5 - CAN Implementation

5.1 Introduction

This chapter details the efforts involved in the development of the CAN bus network employed in this system's synthesis. The information given in this chapter is divided into numerous sections as outlined below:

- A description of the hardware and software resources utilised to develop CAN nodes employed in this system.

- A discussion on how potentiometers are incorporated into the constructed CAN nodes to mimic the operation of standard vehicle sensors, and how their functionality was verified.

- A synopsis detailing the steps taken to configure the *Blackfin's* CAN module and how the operation of the device was tested for conformity.

5.2 Construction of CAN Nodes

The fundamental hypothesis of this application involves the reading of standard vehicle information from sensors over a CAN network, processing the data, and then representing it visually upon a display-device. Consequently, some method of mimicking the operation of automobile sensing-devices is required. This is achieved by constructing two CAN nodes that incorporate several potentiometers to imitate the actions of sensors found in a vehicle. It can be said that sensors essentially function as transducers; i.e. they measure a particular physical parameter and represent it proportionally in another form; typically electrical. Therefore the rotation of a potentiometer and thus the subsequent change in output-voltage suitably impersonates the operation of a sensing-device. For instance, one potentiometer is employed to replicate the actions of an oil temperature sensor while another is used to represent a device that monitors vehicle speed.

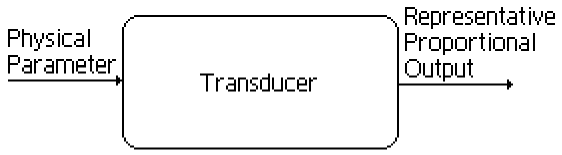


Figure 35: Function of a Transducer

5.2.1 Hardware Contents of Constructed CAN Nodes

Recall from *Chapter 2* that a typical CAN node encompasses a software application that is programmed onto an embedded device. The embedded devices incorporated into the constructed CAN nodes come from the 8-bit *PIC* microcontroller family [56]. 8-bit *PIC* microcontrollers offer a considerable performance at a competitive price which justifies their selection for use. One of the CAN nodes incorporates a *PIC18F258* [57] which contains an integrated CAN Controller. With reference to *Section 4.2.3*, the *PIC18F258* is therefore efficient in terms of CAN overhead and propagation delay. The other constructed CAN node features a *PIC16F876A* [58] which does not include an integrated

CAN Controller. Consequently, a *MCP2515* IC [59] is interfaced to the *PIC16F876A* through a *SPI* link. The use of a peripheral CAN Controller is deliberate in order for the author to be proficient, both in terms of hardware and software, with the integrated and peripheral CAN strategies. As seen below both CAN nodes utilise a *MCP2551* CAN Transceiver [60].

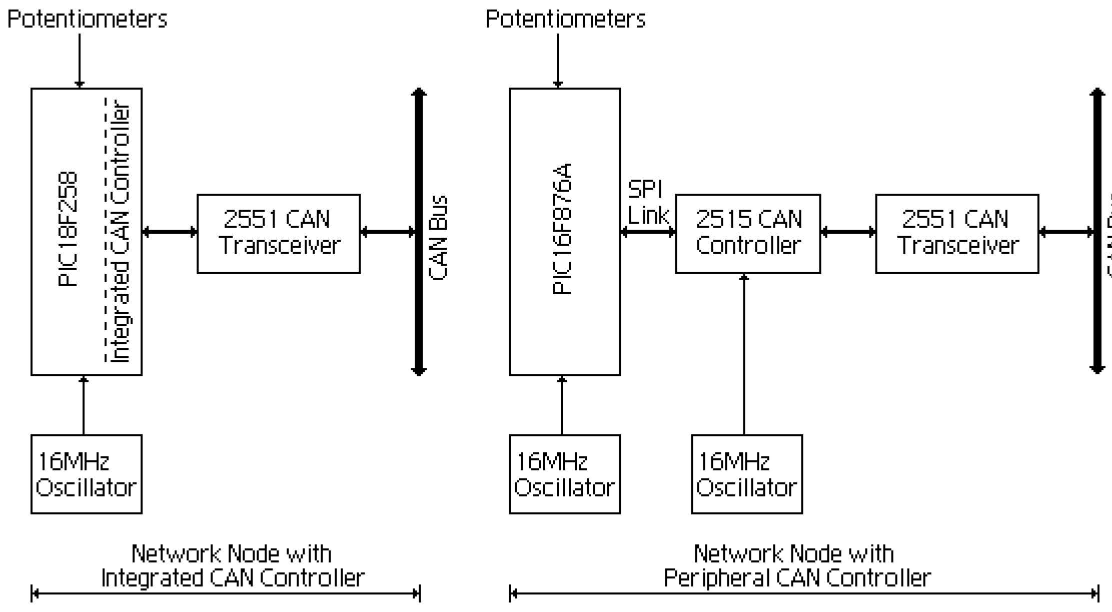


Figure 36: Hardware Components of Constructed CAN Nodes

Detailed circuit schematics for both CAN nodes are found in *Appendix A*.

From the preceding diagram it is seen that both network nodes utilise 16MHz crystal oscillators to obtain their timing. This CAN network is configured to operate at a baud rate of 500kBits/s. With reference to *Section 2.3.1.4*, it is not necessary for all CAN nodes to use the same oscillator frequency. However recall that all CAN nodes must operate at the same *NBR*. Consequently, the *BRP* of both CAN nodes is suitably set by configuring specific bits in the appropriate registers. The software routines used to do this are discussed in *Section 5.2.2.2*.

5.2.2 Software Implementation of Constructed CAN Nodes

The *MikroC* compiler [61] is used in this synthesis to program the *PIC* microcontrollers. This integrated development environment offers a rich set of functions and efficient support for the *PIC* microcontroller families; hence its utilisation is practical and convenient.

The function of the software applications executing inside both CAN nodes is to firstly perform A-D (Analogue-to-Digital) conversions upon the potentiometers. Following on from this, the software applications insert the conversion results into *Standard Data Frames* for transmission to the *Blackfin* for interpretation. The flow chart below illustrates this process. The source code for the CAN On-Board and SPI nodes can be viewed in *Appendix B* and *C* respectively.

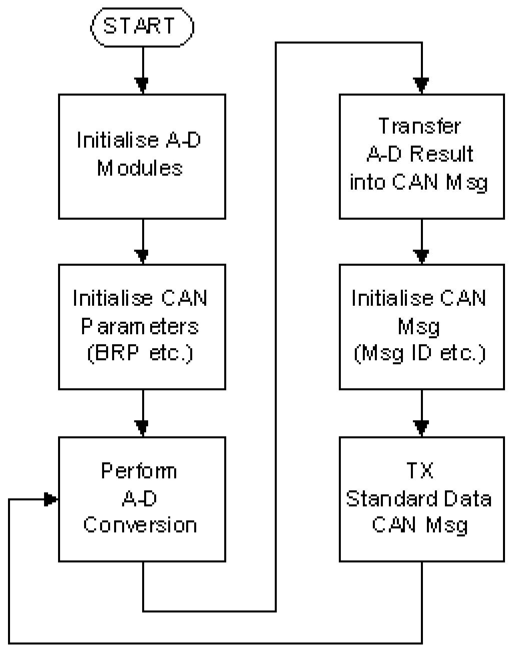

START
Initialise A-D Modules
Initialise CAN Parameters (BRP etc.)
Perform A-D Conversion
Transfer A-D Result into CAN Msg
Initialise CAN Msg (Msg ID etc.)
TX Standard Data CAN Msg

Figure 37: Flow Chart of CAN Nodes Software Applications

5.2.2.1 A-D Conversion

The *MikroC* compiler, as mentioned previously, contains a rich set of functions to simplify the programming process of the *PIC* microcontroller. The *Adc_Read()* function is used to read a 10-bit A-D conversion from a specific channel [62]. The only parameter passed to the *Adc_Read()* function is the channel number upon which A-D conversion is required. For example, the function call below results in a reading of the voltage from the potentiometer connected to channel one.

```
Ch1_res = Adc_Read(1); // Get the ADC conversion result
```

Adc_Read() also implicitly determines, from the supplied clock frequency, the time period necessary for performing A-D conversion. The *PIC16F876A* and *PIC18F258* contain five and eight A-D channels respectively. Three potentiometers are connected to the *PIC16F876A* hence a single call to *Adc_Read()* is required for each of the three channels. On the other hand two individual calls to *Adc_Read()* are made for the two potentiometers interfaced to the *PIC18F258* - see *Appendix A, B,* and C. Before the *Adc_Read()* function is utilised a certain degree of initialisation takes place. The *ADCON* registers of both *PIC* devices are configured accordingly [57], [58].

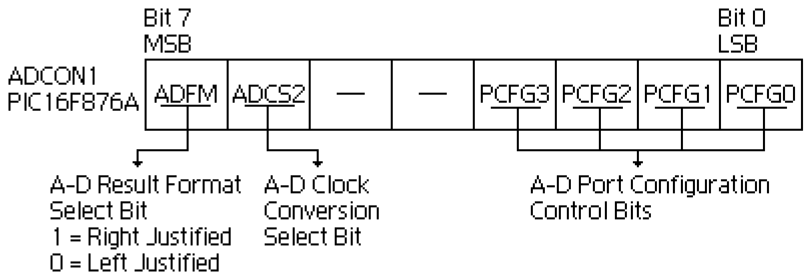


Figure 38: *ADCON1* Register of *PIC16F876A* [58]

The two microcontrollers are configured for all A-D channels to accept analogue inputs only, and conversions occur at a rate of $F_{OSC}/2$. Furthermore, both devices are initialised to issue a right-justified A-D result. Conversely they can be configured to yield a left-justified result. What is the difference between the two configurations?

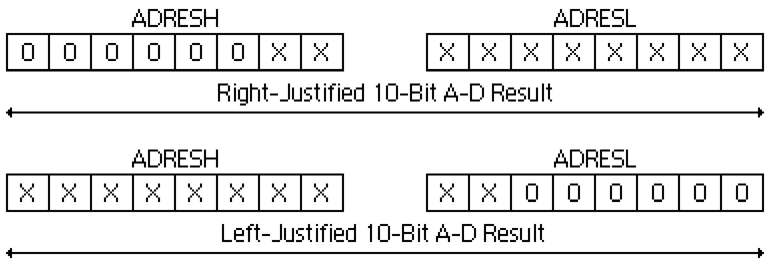


Figure 39: Right & Left Justified A-D Results

The *PIC16F876A* and *PIC18F258* are 8-bit microcontrollers; however both devices perform 10-bit A-D conversion. Consequently, a 10-bit A-D result is split between two 8-bit result registers - *ADRESH* and *ADRESL* as seen in the previous diagram. If the A-D result is right-justified the two MSBs (Most Significant Bit) of the result reside in the *ADRESH* register, while the remaining eight bits of the conversion are stored in the *ADRESL* register. In contrast, if the A-D result is left-justified the eight MSBs of the result are found in the *ADRESH* register, while the two LSBs (Least Significant Bit) are stored in the *ADRESL* register [57], [58].

Once all initialisation is complete the analogue voltages from the potentiometers are continuously read by the A-D modules using the *Adc_Read()* function.

```
typedef unsigned int iadc;
…
iadc ch0_res = 0
…
ch0_res = Adc_Read(0); // Get the ADC conversion result
```

The 10-bit result from a specific A-D channel, contained in *ADRESH* and *ADRESL*, is returned by *Adc_Read()* and the value is stored in a 16-bit unsigned integer variable.

5.2.2.2 CAN Initialisation & Transmission

MikroC provides numerous functions for the initialisation and transmission/reception of CAN messages, and adequately supports both the integrated (*PIC18F258*) and peripheral (*PIC16F876A*) CAN strategies. In most cases the only difference between a function used with an integrated CAN controller and that used with a peripheral controller is in the name of the function - see *Appendix B* and *C*. For example, the *CANWrite()* function is used to transmit a message from a node that incorporates an integrated CAN Controller. On the other hand, the *CANSPIWrite()* function is used to transmit data when a peripheral CAN controller is utilised. As mentioned earlier, *Standard Data Frames* are employed in this application.

A summary of the operation of both CAN nodes is seen below.

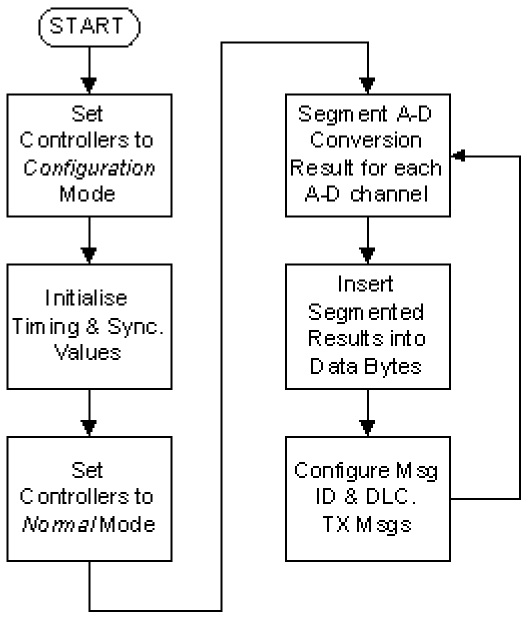

START
Set Controllers to *Configuration* Mode
Initialise Timing & Sync. Values
Set Controllers to *Normal* Mode
Segment A-D Conversion Result for each A-D channel
Insert Segmented Results into Data Bytes
Configure Msg ID & DLC. TX Msgs

Figure 40: Flowchart of CAN Initialisation & Message Transmission

Similar to the A-D conversions seen in the last section, a certain degree of initialisation takes place before any messages are transmitted. To initialise the registers of CAN modules residing in either an integrated or peripheral controller the module has to be set to the *Configuration Mode* using either the *CANSetOperationMode()* or *CANSPISetOperationMode()* respectively. In both cases, two parameters are passed to the function. The first parameter passed is the mode in which it is desired to enter into. This parameter is copied into the *CANSTAT* register of either the integrated or peripheral controller; depending on which strategy is being utilised [57], [58].

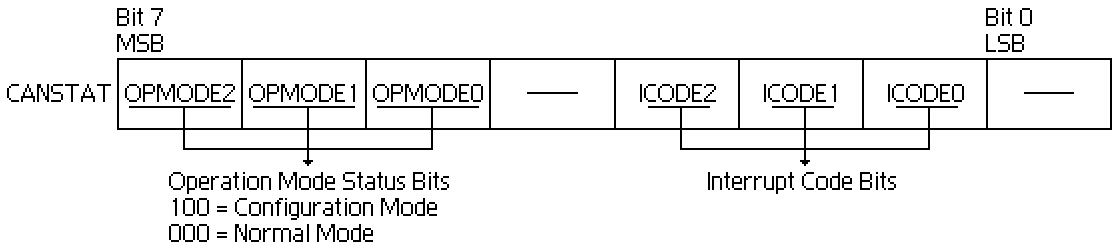


Figure 41: *CANSTAT* Register [57], [58]

The second item in the function prototype is either a “blocking” or “non-blocking” call. If it is a “blocking” call, i.e. 0xFF, the function does not return until the requested mode is entered into. If a “non-blocking” call, i.e. 0x00, is passed the function returns immediately but the system-designer must ensure that the CAN Controller is now residing in the requested mode [62].

```
CANSetOperationMode(CAN_MODE_CONFIG,0xFF); // Set CONFIGURATION mode
CANSPISetOperationMode(CAN_MODE_CONFIG,0xFF); // Set CONFIGURATION mode
```

In *Sections 2.3.1.3* and *2.3.1.4* it is stated that one of the appealing aspects of the CAN bus protocol is that its bit rate, sample and resynchronisation points are user-programmable. These parameters are initialised using the *CANInitialize()* and *CANSPIInitialize()* functions. Several items are passed to both functions.

```
CANInitialize( 2,2,3,3,1,aa); // Initialise CAN module. BAUD = 500kBit/sec
CANSPIInitialize( 2,2,3,3,1,aa); // Initialise external CAN module. BAUD = 500kBit/sec
```

Firstly, the *SJW* resynchronisation value is passed. It is assigned the value of two; therefore the bit period, t_{bit}, of a CAN message is lengthened or shortened by $2t_q$ if resynchronisation is required [57], [58].

The second parameter is the *BRP* value and this is discussed in a few moments. The third and fourth items in the function prototypes are the *PhaseSeg1* and *PhaseSeg2* values. Recall from *Section 2.3.1.3* that these elements compensate for any edge error that appears around the sample point. They are both assigned the value of three. Thus Rule 5 of *Table 4* is satisfied, i.e. $SJW \leq PhaseSeg2 \leq PhaseSeg1$. *PropSeg* is the next value passed to the function and it is used to compensate for any propagation delay. This is assigned the value of one.

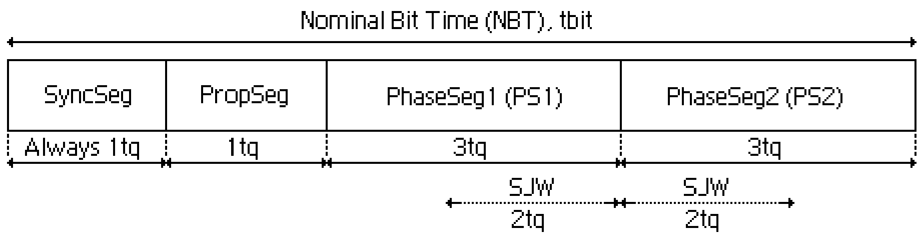


Figure 42: Assigned Parameter Values

As mentioned above the second parameter passed to both *CANInitialize()* and *CANSPIInitialize()* is the *BRP* value. Recall that the *BRP* of a particular node is used to ensure that it functions at an identical *NBR* to all other nodes connected to the network, even if it does not use the same oscillator frequency. To reiterate, this CAN network is configured to operate at 500kBits/s. Therefore from *Eq. 2.2* in *Section 2.3.1.3*:

$$NBR = f_{bit} = \frac{1}{t_{bit}} = 500kBits/s$$

$$\Rightarrow t_{bit} = \frac{1}{500kBits/s} = 2\mu s$$

Concurrently, from *Eq. 2.3*:

$$t_{bit} = t_{SyncSeg} + t_{\Pr opSeg} + t_{PS1} + t_{PS2}$$

From the assigned values in *CANInitialize()* and *CANSPIInitialize()*: (Note $t_{SyncSeg}$ is implicitly $1t_q$ in duration.)

$$t_{bit} = (1+1+3+3)\times t_q = 8t_q$$

$$\Rightarrow t_q = \frac{t_{bit}}{8} = \frac{2\mu s}{8} = 0.25\mu s$$

Both CAN nodes incorporate 16MHz oscillators to derive their timing; thus from *Eq. 2.5*:

$$t_q = \frac{2\times BRP}{f_{osc}} = 0.25\mu s = \frac{2\times BRP}{16MHz}$$

$$\Rightarrow BRP = \frac{t_q \times f_{osc}}{2} = \frac{0.25\mu s \times 16MHz}{2} = 2$$

Thus a *BRP* value of two results in both nodes operating at a *NBR* of 500kBits/s when using a 16MHz oscillator to derive their timing.

The last parameter in the *CANInitialize()* and *CANSPIInitialize()* prototypes contains a list of constants that are bitwise *ANDED* together and relate to CAN module configuration. They include factors, for example, that determine whether the logic level is sampled once or three times during the *NBT* [62].

```
aa = CAN_CONFIG_SAMPLE_THRICE &  // form value to be used
     CAN_CONFIG_PHSEG2_PRG_ON & // with CANInitialize()
     CAN_CONFIG_ALL_MSG &
     CAN_CONFIG_DBL_BUFFER_ON &
     CAN_CONFIG_LINE_FILTER_OFF;
```

Once initialisation is complete a respective CAN controller is set to *Normal Mode* in order to commence data transmission.

```
CANSetOperationMode(CAN_MODE_NORMAL,0); // Set NORMAL mode
CANSPISetOperationMode(CAN_MODE_NORMAL,0); // Set NORMAL mode
```

Recall that the previous section outlines that the 8-bit *PIC* microcontrollers used in this system perform 10-bit A-D conversion. The right-justified return value of *Adc_Read()* is stored in a 16-bit unsigned integer variable as discussed previously.

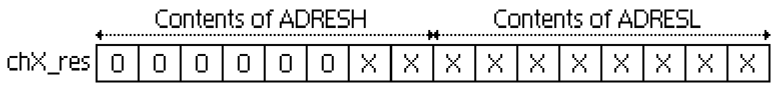


Figure 43: 16-bit Variable Contains Right-Justified 10-bit A-D Result

Section 2.3.2.1 outlined that the *Data* field of a CAN *Data* frame contains zero to eight bytes of data. For that reason the 10-bit A-D conversion result from a particular channel, contained in the 16-bit variable, is appropriately manipulated in order to insert it into two 8-bit CAN data bytes without the loss of any information. This involves segmenting the 16-bit variable into two bytes of data. Therefore two characters, i.e. two 8-bit data variables are declared. One character, *ls_chX_res*, will store the lower eight bits of the conversion result, while another character, *ms_chX_res*, will store the upper two bits of the A-D conversion.

```
typedef unsigned char uchar;
…
uchar ms_chX_res = 0; // ADC Channel X MSB result variable
uchar ls_chX_res = 0; // ADC Channel X LSB result variable
```

The assigning of the value of the 16-bit data variable to *ls_chX_res* results in the eight MSBs of the 16-bit integer being discarded and the eight LSBs of the integer being stored in the character variable.

```
ls_chX_res = chX_res; // Get bottom 8 bits of ADC Channel X conversion
```

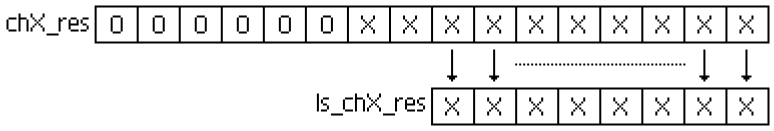


Figure 44: Assignment of Lower 8-bits of A-D Conversion

In order to assign the two MSBs of the 10-bit A-D conversion to *ms_chX_res* the 16-bit variable is manipulated using the bitshift-right operator, >>. The contents of *chX_res* are shifted eight places to the right. The resulting value is now assigned to the most significant character. Therefore the two MSBs of the A-D conversion are now the two LSBs of the *ms_chX_res*.

```
ms_chX_res = chX_res >> 8; // Get top 2 bits of ADC Channel X conversion
```

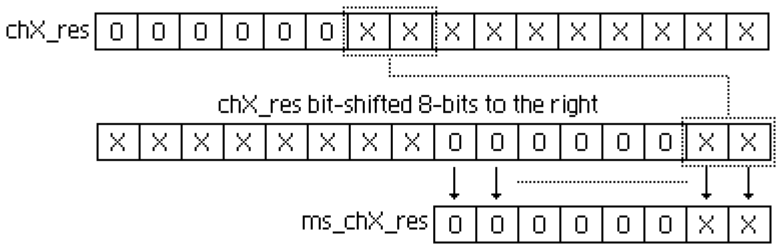

chX_res
0 0 0 0 0 0 X X X X X X X X X X
chX_res bit-shifted 8-bits to the right
X X X X X X X X 0 0 0 0 0 0 X X
ms_chX_res
0 0 0 0 0 0 X X

Figure 45: Assignment of Upper 2-bits of A-D Conversion

Once the A-D conversion result of each channel is appropriately manipulated it is inserted into CAN data byte registers - see *Appendix B* and *C*.

```
data[0] = ms_ch2_res; // 2 MSBs of Channel 2 conversion result
data[1] = ls_ch2_res; // 8 LSBs of Channel 2 conversion result
data[2] = ms_ch1_res; // 2 MSBs of Channel 1 conversion result
```

```
data[3] = ls_ch1_res; // 8 LSBs of Channel 1 conversion result
…
…
data[6] = 22; // Arbitrary Number
data[7] = 33; // Arbitrary Number
```

Lastly, the construction of the CAN *Data* messages are completed and transmitted onto the network using the *CANWrite()* and *CANSPIWrite()* functions.

```
id = 0x411; // Message ID (Decimal 1041)
len = 8; // Data Length Code
CANWrite(id,data,len,aa1); // Write CAN message

id = 0x189; // Message ID (Decimal 393)
len = 8; // Data Length Code
CANSPIWrite(id,data,len,aa1); // Write CAN message
```

The first parameter passed to both *CANWrite()* and *CANSPIWrite()* is the *Identifier Field*, which assigns an ID to a particular message. The next item in the function prototype is the address of the first data byte in the array of information (A-D conversion results plus arbitrary numbers in this application) that is transmitted. This can be up to 8-bytes in length. The third item passed is essentially the *DLC* discussed in *Section 2.3.2.1* and is used to indicate the number of bytes contained in the *data* field. The last quantity in the prototype of *CANWrite()* and *CANSPIWrite()* incorporates a list of constants that are bitwise *ANDED* together. They include factors such as message priority etc. and indicate whether the frame is a standard *Data* frame or otherwise [62].

```
aa1 = CAN_TX_PRIORITY_0 & // form value to be used
    CAN_TX_STD_FRAME & // with CANWrite()
    CAN_TX_NO_RTR_FRAME;
```

5.3 Testing of Constructed CAN Nodes

The hardware and software functionality of both CAN nodes was verified using the *CANKing* GUI (Graphic User Interface) test package [63]. *CANKing* is an easy-to-use development tool and essentially allows a computer/laptop to function as a CAN node for test purposes. It achieves this by interfacing to *Microchip's MCP2515* development board via a parallel-port connection [64]. For that reason the *Port95NT* parallel-port driver was required [65].

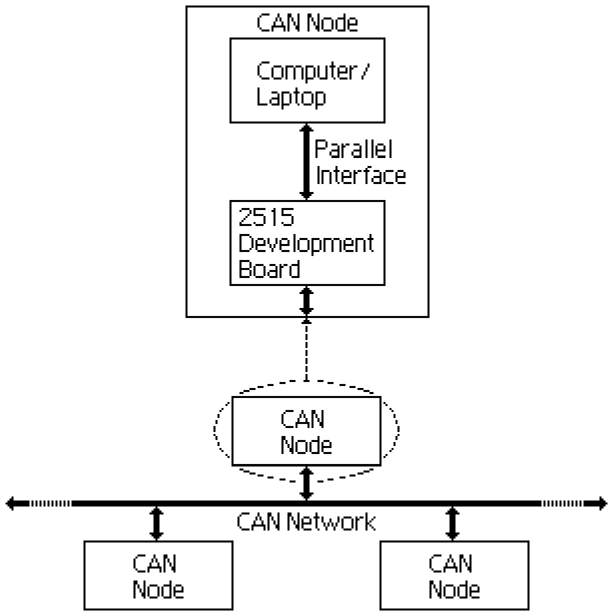
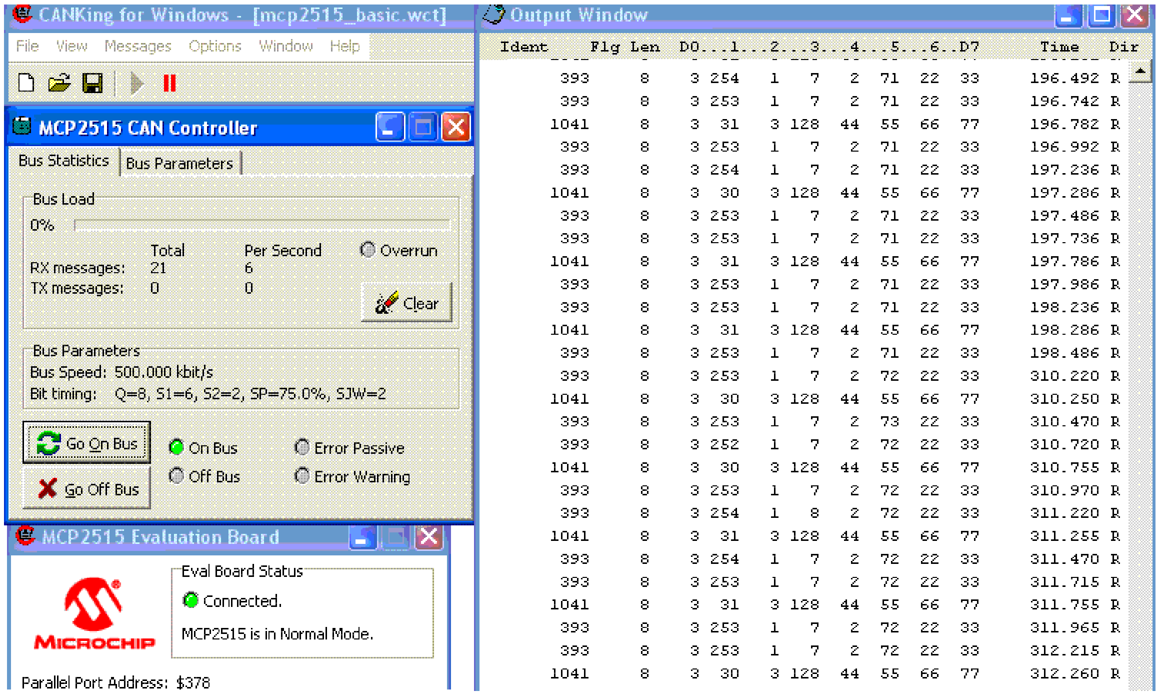


Figure 46: *CANKing* allows a Computer/Laptop to Function as a CAN Node

The *MCP2515* development board incorporates a *MCP2515* CAN Controller and *MCP2551* CAN Transceiver [59], [60]. Thus, *CANKing* is able to transmit/receive messages to/from a CAN network via the *MCP2515* development board. The development suite possesses the ability to display numerous factors like traffic and bus loading statistics, a history of messages transmitted/received, time-stamp information and data content for received/transmitted messages.

In this particular case, the author was only concerned with verifying if the constructed CAN On-Board and CAN SPI nodes transmitted the correct data. Therefore time-stamp information along with message content was paramount. The two constructed CAN nodes along with the *MCP2515* development board were connected to the same network and *CANKing* was used to monitor message activity.



Bus Load
0%

	Total	Per Second
RX messages:	21	6
TX messages:	0	0

Overrun

Bus Parameters
Bus Speed: 500.000 kbit/s
Bit timing: Q=8, S1=6, S2=2, SP=75.0%, SJW=2

On Bus
Off Bus
Error Passive
Error Warning

Eval Board Status
Connected.
MCP2515 is in Normal Mode.

Parallel Port Address: $378

```
Ident     Flg Len  D0...1...2...3...4...5...6..D7       Time   Dir
     393        8    3 254   1   7   2  71  22  33    196.492 R
     393        8    3 253   1   7   2  71  22  33    196.742 R
    1041        8    3  31   3 128  44  55  66  77    196.782 R
     393        8    3 253   1   7   2  71  22  33    196.992 R
     393        8    3 254   1   7   2  71  22  33    197.236 R
    1041        8    3  30   3 128  44  55  66  77    197.286 R
     393        8    3 253   1   7   2  71  22  33    197.486 R
     393        8    3 253   1   7   2  71  22  33    197.736 R
    1041        8    3  31   3 128  44  55  66  77    197.786 R
     393        8    3 253   1   7   2  71  22  33    197.986 R
     393        8    3 253   1   7   2  71  22  33    198.236 R
    1041        8    3  31   3 128  44  55  66  77    198.286 R
     393        8    3 253   1   7   2  71  22  33    198.486 R
     393        8    3 253   1   7   2  72  22  33    310.220 R
    1041        8    3  30   3 128  44  55  66  77    310.250 R
     393        8    3 253   1   7   2  73  22  33    310.470 R
     393        8    3 252   1   7   2  72  22  33    310.720 R
    1041        8    3  30   3 128  44  55  66  77    310.755 R
     393        8    3 253   1   7   2  72  22  33    310.970 R
     393        8    3 254   1   8   2  72  22  33    311.220 R
    1041        8    3  31   3 128  44  55  66  77    311.255 R
     393        8    3 254   1   7   2  72  22  33    311.470 R
     393        8    3 253   1   7   2  72  22  33    311.715 R
    1041        8    3  31   3 128  44  55  66  77    311.755 R
     393        8    3 253   1   7   2  72  22  33    311.965 R
     393        8    3 253   1   7   2  72  22  33    312.215 R
    1041        8    3  30   3 128  44  55  66  77    312.260 R
```

Figure 47: Verification of Correct Functionality of Constructed CAN Nodes

From the diagram above it is seen that both CAN nodes operated as desired. The CAN On-Board node, incorporating the *PIC18F258*, transmitted messages approximately every 500 milliseconds with the correct *Identifier Field* – i.e. 1041. It's *Data Field* correctly contained the segmented A-D conversion results for both potentiometers integrated into this particular node along with four arbitrary numbers. As both potentiometers were varied the relevant bytes within the *Data Field* updated accurately. Similarly, the CAN SPI node, incorporating the *PIC16F876*, sent messages over the network every 250 milliseconds using the correct *Identifier Field* – i.e. 393. The node's *Data Field* correctly contained the segmented A-D conversion results for the three potentiometers integrated into this individual node along with two arbitrary numbers. Again, the appropriate data bytes within the messages continuously transmitted from this node updated as soon as the three potentiometers were varied. Thus, as both CAN nodes operate as desired the potentiometers suitably mimic the operation of sensors and the simulated vehicle measurements are transmitted correctly over the network.

5.4 CAN Implementation upon the Blackfin ADSP-BF537

The CAN module of the *Blackfin ADSP-BF537* is configured to interpret the transmitted messages containing “sensor measurements” in order to take appropriate action to graphically-display the data upon a display-device.

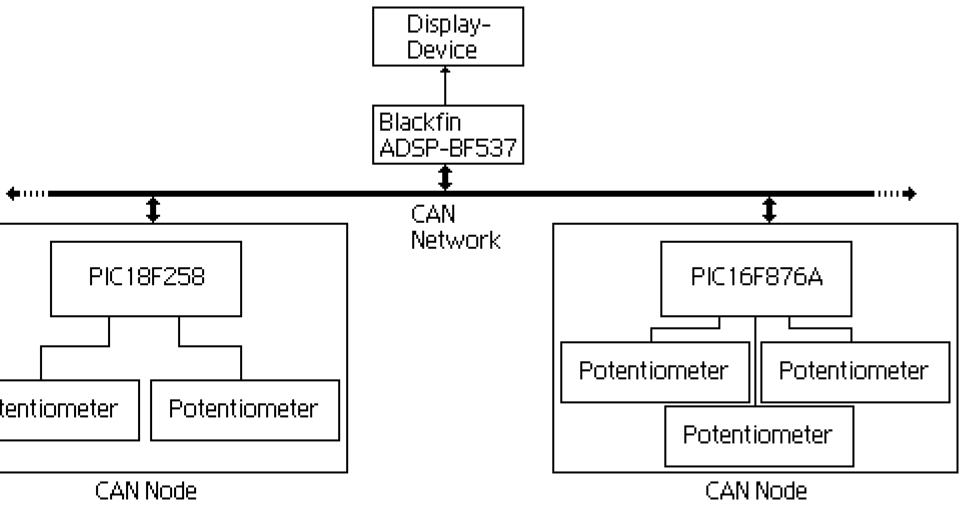


Figure 48: CAN Network Consisting of *Blackfin* & Constructed Nodes

The CAN module utilises *Port J* of the *ADSP-BF537* device and interfaces with the *Philips TJA1041* CAN Transceiver [66] incorporated onto the *ADSP-BF537 EZ Kit Lite* development board. To enable the CAN module on the *ADSP-BF537 EZ Kit Lite* all the elements of *Switch 2* must be turned on [67].

In *Section 4.2.3.1* it was outlined that the *Blackfin ADSP-BF537* processor incorporates thirty-two mailboxes (message buffers) within it's CAN module. Eight of these buffers are transmit only, another eight are receive only, while the remaining sixteen are programmable in direction. Each of these mailboxes has associative 32 or 16-bit control and data registers which are appropriately configured before a message buffer is enabled for use [42].

The flowchart below illustrates the steps taken to configure a CAN mailbox.

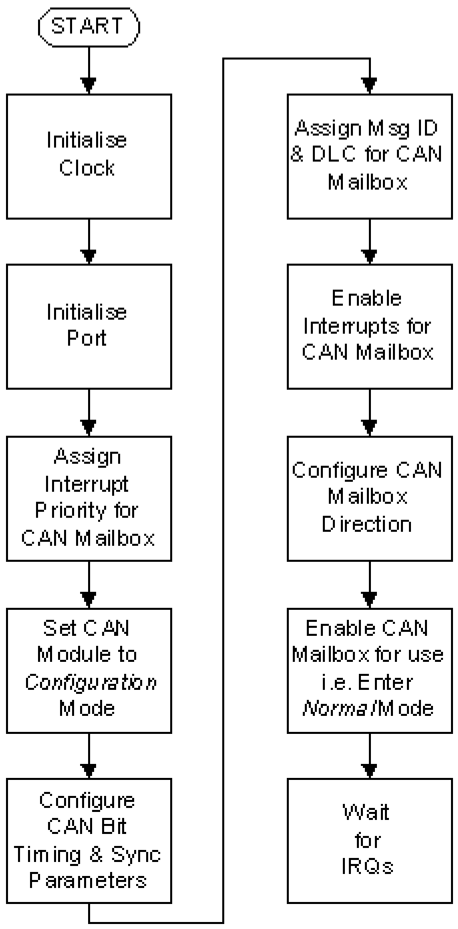

START
Initialise Clock
Initialise Port
Assign Interrupt Priority for CAN Mailbox
Set CAN Module to *Configuration* Mode
Configure CAN Bit Timing & Sync Parameters
Assign Msg ID & DLC for CAN Mailbox
Enable Interrupts for CAN Mailbox
Configure CAN Mailbox Direction
Enable CAN Mailbox for use i.e. Enter *Normal* Mode
Wait for IRQs

Figure 49: Configuration of a *Blackfin* CAN Mailbox

If not previously configured, the SCLK (Processor System Clock) of the *Blackfin* is derived from the CCLK (Processor Core Clock) [42]. A frequency value that is suitable to *BRP* derivation is typically chosen. Next, the *ADSP-BF537* CAN module is enabled by initialising *Port J* of the processor. The *Blackfin* employs an interrupt policy for it's CAN module, which is opposite to the polling strategy implemented upon the constructed CAN nodes. As a result, the interrupt priority for the mailbox undergoing the initialisation process is assigned.

Configuration Mode is entered to configure the CAN module's internal registers. On power-up or reset, the module automatically resides in *Configuration Mode*. However to explicitly enter *Configuration Mode* a request is made by setting the *CCR* bit of the *CAN_CONTROL* register to Logic 1. A designer must test to see if the module is now residing in *Configuration Mode* by polling the *CCA* bit of the *CAN_STATUS* register [42].

The *SJW*, *PhaseSeg1* and *PhaseSeg2* values of the CAN module are programmed by appropriately configuring the *CAN_TIMING* register.

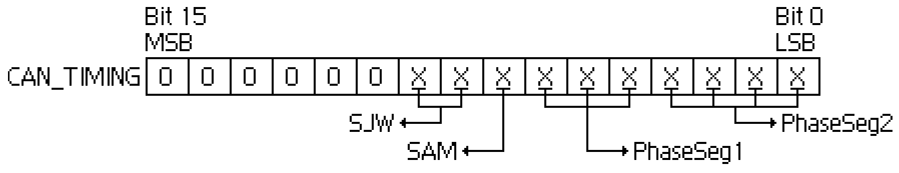


Figure 50: *ADSP-BF537's CAN_TIMING* Register [42]

The *BRP* value of the *Blackfin's* CAN module is defined using the *CAN_CLOCK* register.

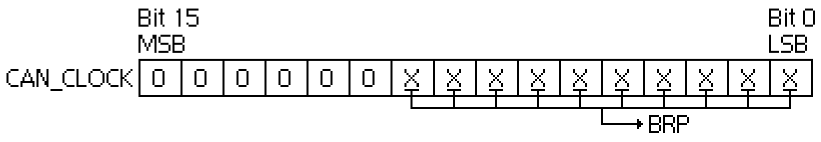


Figure 51: *ADSP-BF537's CAN_CLOCK* Register [42]

The equations governing bit timing and synchronisation for the *Blackfin* processor differ slightly to the universal formulae outlined in previous chapters [42]. This is not unusual as many manufacturers integrate certain timing parameters together resulting in minor formulae variances for bit timing calculations. The *Blackfin's* CAN module, for instance, does not distinguish between *PropSeg* and *PhaseSeg1* as defined by the *Bosch* standard [2]. The *PhaseSeg1* value is intended to cover both parameters. Thus the *NBT*, or t_{bit}, of the *Blackfin's* CAN module is found using:

$$t_{bit} = (1 + (1 + PhaseSeg1) + (1 + PhaseSeg2)) \times t_q \qquad \textit{Eq. 5.1}$$

, where t_{bit} is the bit period (Seconds),
PhaseSeg1 is a programmed integer value (0 - 15),
PhaseSeg2 is a programmed integer value (0 - 7),
t_q is the time quantum (Seconds).

The time quantum, t_q, and the *BRP* of the *Blackfin's* CAN Module are related by the following:

$$t_q = \frac{1 + BRP}{SCLK} \qquad \textit{Eq. 5.2}$$

, where t_q is the time quantum (Seconds),
BRP is a user-configurable prescalar integer unit (0 - 1023),
SCLK is the Processor System Clock (Hz).

As discussed earlier this CAN network operates at 500kBits/s. Thus, a suitable *BRP* value for the *Blackfin* CAN module is required. From *Eq. 2.2* in *Section 2.3.1.3*:

$$NBR = f_{bit} = \frac{1}{t_{bit}} = 500kBits/s$$

$$\Rightarrow t_{bit} = \frac{1}{500kBits/s} = 2\mu s$$

For *PhaseSeg1* and *PhaseSeg2* values of five and three respectively *Eq. 5.1* yields:

$$t_{bit} = (1 + (1 + PhaseSeg1) + (1 + PhaseSeg2)) \times t_q$$

$$\Rightarrow 2\mu s = (1 + (1 + 5) + (1 + 3)) \times t_q$$

$$\Rightarrow t_q = \frac{2\mu s}{11} = 0.182\mu s$$

The SCLK frequency utilised is 120MHz; thus from *Eq. 5.2*:

$$t_q = \frac{1 + BRP}{SCLK}$$

$$\Rightarrow BRP = (t_q \times SCLK) - 1$$

$$\Rightarrow BRP = (0.182\mu s \times 120MHz) - 1 \approx 21$$

Hence, for a SCLK frequency of 120MHz a *BRP* value of twenty-one results in the CAN module operating at 500kBits/s.

The *Identifier Field* of a specific mailbox is configured using the appropriate *CAN_MB_XXID1* register. In addition, the *RTR* bit of this register indicates if a message is *Remote* or *Standard* - see *Section 2.3.2.1*. If a mailbox is set up to transmit/receive *Extended Data Frames* the remainder of the *Identifier Field* is defined using the apt *CAN_MB_XXID0* register.

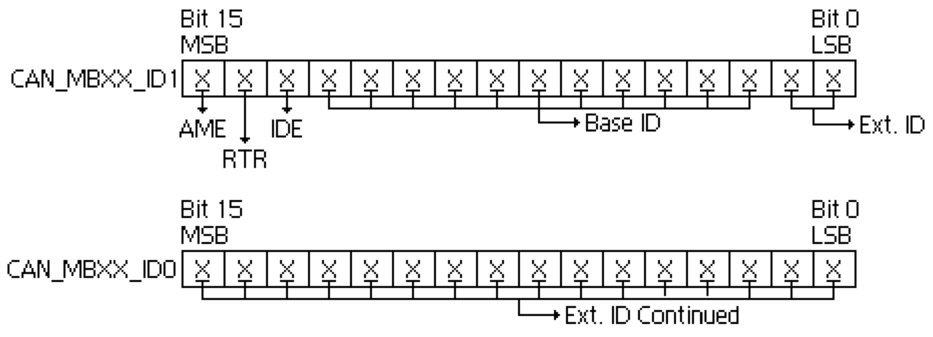


Figure 52: *CAN_MBXX_ID1* & *CAN_MBXX_ID0* Registers [42]

The *DLC* for an individual mailbox is programmed using the appropriate *CAN_MBXX_LENGTH* register. To enable a particular mailbox to generate an interrupt the corresponding bit in the *CAN_MBIMX* register is set to Logic 1 [42]. The direction of a mailbox, i.e. transmit or receive for a bi-directional buffer, is configured by programming a corresponding bit in the relevant *CAN_MDX* register. Logic 1 indicates that the mailbox is configured for message reception; while on the other hand, Logic 0 indicates that the mailbox is configured for message transmission.

Each of *Blackfin's* mailboxes include four 16-bit data byte registers – CAN*_MBXX_DATA[3..0]*. These four registers are used to store the *Data Field* members of a CAN message. Consequently two data bytes are stored in each of the four data registers. Data contained within these registers are transmitted MSB first from the *CAN_MBXX_DATA3/2/1/0* registers, respectively, based on the value defined for the *DLC*. For instance, if only one byte is transmitted or received, i.e. *DLC* = 1, then it is stored in the most significant byte of the *CAN_MBXX_DATA3* register [42].

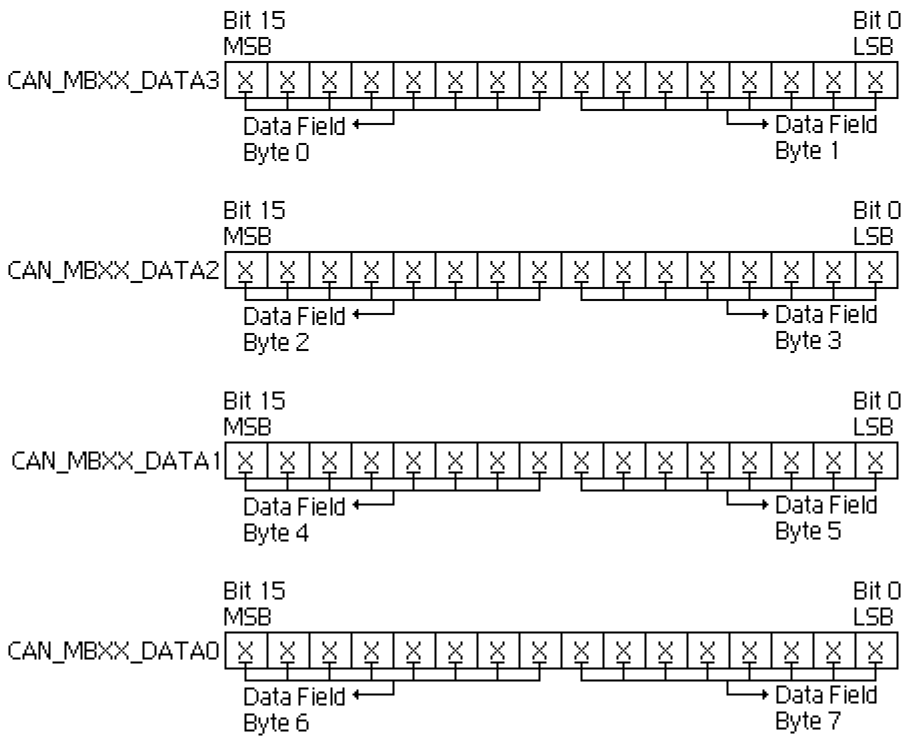


Figure 53: CAN Modules *Data Field* Registers [42]

5.5 Testing of Blackfin's CAN Module

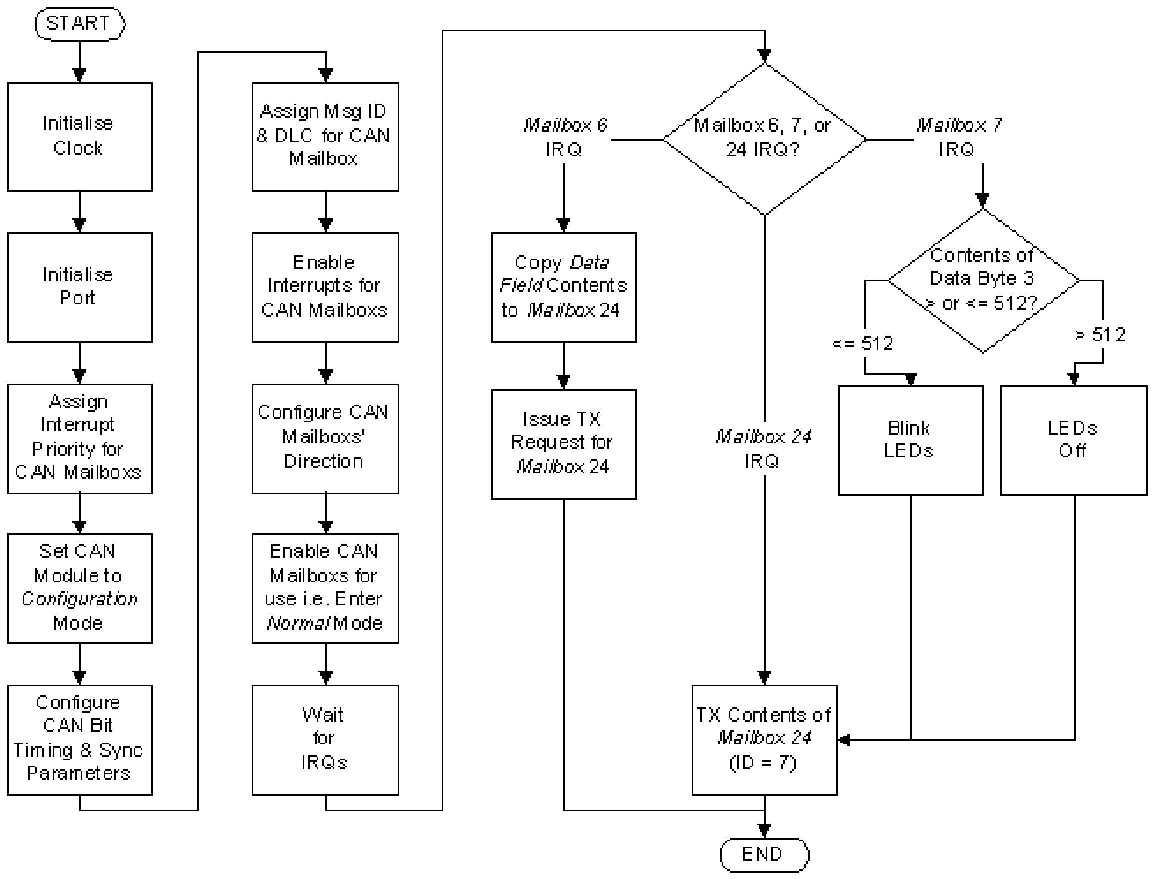
Within this application the *Blackfin* is required to receive simulated vehicle data from the constructed CAN nodes and interpret this information for further processing. Therefore a test program was developed to configure the *Blackfin's* CAN module for message transmission and reception. Note that even though CAN transmission from the *Blackfin* is not a prerequisite for this system it was developed in this test program to allow for future expansion. The source code for this test program is found in *Appendix D*.

The test program essentially involved initialising three *Blackfin* mailboxes (message buffers) appropriately. Correct message reception was verified by allocating an individual mailbox for each of the two constructed CAN nodes. *Mailbox 6* was configured to receive messages with ID 393, while *Mailbox 7* was programmed to receive messages with ID 1041. When a specific mailbox received a pertinent CAN message from the network it performed a particular ISR (Interrupt Service Routine). For instance, as soon as *Mailbox 6* received a message with ID 393 an ISR copied the *Data Field* of this message buffer to *Mailbox 24* and issued a transmission request. As a consequence of this message transmission was implicitly tested. The ISR performed when *Mailbox 7* received a relevant message involved turning on/off LEDs (Light Emitting Diode) incorporated onto the *ADSP-BF537 EZ Kit Lite* development board.

Mailbox 6	Mailbox 7	Mailbox 24
ID = 393	ID = 1041	ID = 7
DLC = 8	DLC = 8	DLC = 8

Figure 54: Mailbox Configurations for Testing of *Blackfin* CAN Module

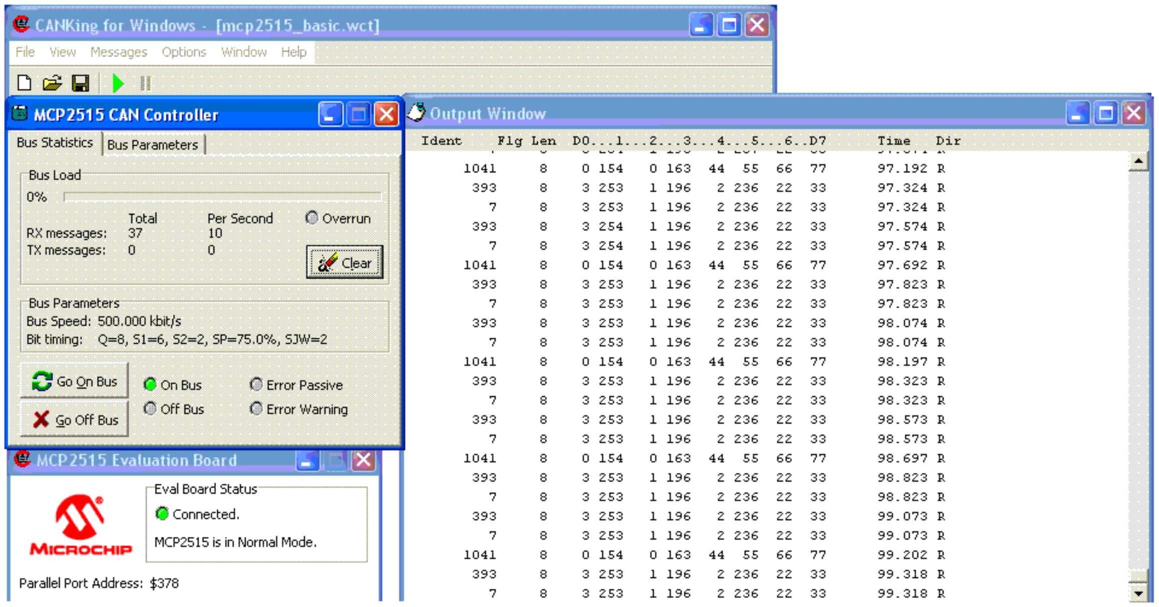
The flow chart below illustrates the operations of the test program.



START
Initialise Clock
Initialise Port
Assign Interrupt Priority for CAN Mailboxs
Set CAN Module to Configuration Mode
Configure CAN Bit Timing & Sync Parameters
Assign Msg ID & DLC for CAN Mailbox
Enable Interrupts for CAN Mailboxs
Configure CAN Mailboxs' Direction
Enable CAN Mailboxs for use i.e. Enter Normal Mode
Wait for IRQs
Mailbox 6, 7, or 24 IRQ?
Mailbox 6 IRQ
Mailbox 7 IRQ
Mailbox 24 IRQ
Copy Data Field Contents to Mailbox 24
Issue TX Request for Mailbox 24
Contents of Data Byte 3 > or <= 512?
<= 512
> 512
Blink LEDs
LEDs Off
TX Contents of Mailbox 24 (ID = 7)
END

Figure 55: Flowchart for *Blackfin* CAN Module Test Program

The functionality of the CAN module was examined by configuring and connecting the *Blackfin* to the network incorporating the two constructed CAN nodes. *CANKing* was used to monitor bus activity to establish if all components functioned as desired.



	Total	Per Second
RX messages:	37	10
TX messages:	0	0

Bus Parameters
Bus Speed: 500.000 kbit/s
Bit timing: Q=8, S1=6, S2=2, SP=75.0%, SJW=2

MCP2515 is in Normal Mode.

Parallel Port Address: $378

```
Ident     Flg Len  D0...1...2...3...4...5...6..D7      Time   Dir
   1041      8     0 154   0 163  44   55   66   77     97.192 R
    393      8     3 253   1 196   2 236   22   33     97.324 R
      7      8     3 253   1 196   2 236   22   33     97.324 R
    393      8     3 254   1 196   2 236   22   33     97.574 R
      7      8     3 254   1 196   2 236   22   33     97.574 R
   1041      8     0 154   0 163  44   55   66   77     97.692 R
    393      8     3 253   1 196   2 236   22   33     97.823 R
      7      8     3 253   1 196   2 236   22   33     97.823 R
    393      8     3 253   1 196   2 236   22   33     98.074 R
      7      8     3 253   1 196   2 236   22   33     98.074 R
   1041      8     0 154   0 163  44   55   66   77     98.197 R
    393      8     3 253   1 196   2 236   22   33     98.323 R
      7      8     3 253   1 196   2 236   22   33     98.323 R
    393      8     3 253   1 196   2 236   22   33     98.573 R
      7      8     3 253   1 196   2 236   22   33     98.573 R
   1041      8     0 154   0 163  44   55   66   77     98.697 R
    393      8     3 253   1 196   2 236   22   33     98.823 R
      7      8     3 253   1 196   2 236   22   33     98.823 R
    393      8     3 253   1 196   2 236   22   33     99.073 R
      7      8     3 253   1 196   2 236   22   33     99.073 R
   1041      8     0 154   0 163  44   55   66   77     99.202 R
    393      8     3 253   1 196   2 236   22   33     99.318 R
      7      8     3 253   1 196   2 236   22   33     99.318 R
```

Figure 56: Verification of Correct Functionality of *Blackfin* CAN Module

From the preceding diagram it is seen that the devised test code functioned as desired as the *Data Fields* of the messages with IDs 7 and 393 are identical. This proves that *Mailbox 6* within the *Blackfin's* CAN module correctly received messages from the constructed CAN SPI node and copied the contents to *Mailbox 24*. *Mailbox 24*, in turn, re-transmitted the data onto the network under an *Identifier Field* of 7. Additionally, the remaining configured message buffer, *Mailbox 7*, correctly received messages (ID 1041) from the constructed CAN On-Board node. This was proven by twisting the potentiometers situated upon the CAN On-Board node resulting in the LEDs on the *ADSP-BF537 EZ Kit Lite* turning on or off. Thus a mechanism has been developed for correctly initialising the *Blackfin* for CAN communications and integrating the device into an existing CAN network.

5.6 Summary

This chapter outlined the main steps taken to physically implement the CAN network used in this application. The key points for contemplation are as follows:

- Potentiometers are suitable for the purpose of mimicking the operation of standard vehicle sensors. The potentiometers are interfaced with embedded devices chosen from the cost-effective 8-bit PIC microcontroller family to formulate CAN nodes.

- Adequate software routines are utilised to ensure that the full 10-bit resolution of the A-D conversions upon the potentiometers are kept intact prior to message transmission.

- The calculation of the correct *BRP* value for any CAN node is paramount to ensure that all devices communicate at the same *NBR*.

- The *Blackfin ADSP-BF537* contains thirty-two CAN message buffers which require a certain degree of configuration prior to use.

- The two constructed CAN nodes communicated as desired with the *Blackfin's* CAN module. This was verified using the *CANKing* tool suite.

Chapter 6 - Video Implementation

6.1 Introduction

This chapter discusses the measures taken to implement a video display using the *Blackfin ADSP-BF537 EZ Kit Lite* and A/V development boards. The information given in this chapter is separated into the following main sectors:

- A discussion of the device drivers and system services incorporated into the *VisualDSP++* compiler, and how are they are utilised within this synthesis.

- A detailed description of the software test strategy employed to realise video processing.

- A brief outline of the hardware configuration required to achieve successful video processing using the *Blackfin ADSP-BF537 EZ Kit Lite* and A/V development boards.

6.2 Video Implementation Strategy

As a methodology for CAN implementation has been established the next step is to realise video signalling using the *Blackfin ADSP-BF537 EZ Kit Lite* and its compatible A/V daughter board. This is essential in order to visually represent the data received from the CAN network; thus a suitable process for video functionality is required. However, instead of endeavouring to develop video software that works in tandem with the CAN source code a modular approach is taken. This essentially means firstly developing a suitable standalone software strategy for video.

6.2.1 Video Software Strategy

Before delving into the software algorithms employed to implement video it is important to point out that *Analog Devices' VisualDSP++* compiler contains numerous utilities, such as device drivers and system services, which aid in developments incorporating their ICs [68], [69].

6.2.1.1 Device Drivers & System Services

Device drivers are essentially standardised API[2] (Application Program Interface) for *Blackfin* processors that allow for interaction with internal modules and hardware peripherals e.g. *PPI* and video encoders respectively.

[2] An API forms part of a software interface that a compiler or library provides in order to support requests for services to be made of it by an application program.

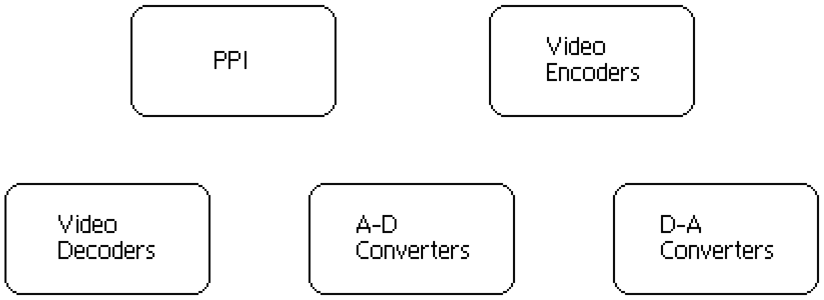


Figure 57: Examples of Supported *Blackfin* Device Drivers

The utilisation of device drivers results in modular programming and portability between *Blackfin* processors. Memory is required by device drivers in order to manage components and this is supplied at the initialisation stage of a software application. All device drivers use “handles”. A handle is quite literally a method for getting a handle on a device; therefore it is fundamentally an address that points to device specific data. For example the following source code declares a handle for an ADV7179 video encoder device driver.

```
ADI_DEV_DEVICE_HANDLE AD7179DriverHandle; // Handle to the ADV7179 Driver
```

All *VisualDSP++* device drivers encompass return codes which indicate the success or failure resulting from the use of a device driver; a zero denotes success, while a non-zero value signifies an error.

The following diagram illustrates the four major functions utilised with device drivers. The purpose of each of the four functions is self explanatory from the diagram. However the term “buffer” needs to be expanded upon. Buffers, with reference to *VisualDSP++* utilities, describe the data for a device driver to process and are provided by the application software exploiting a particular device driver. The application software essentially populates the various fields of the buffer to completely describe the data to the device driver. In other words data is shifted to/from device drivers using buffers [68].

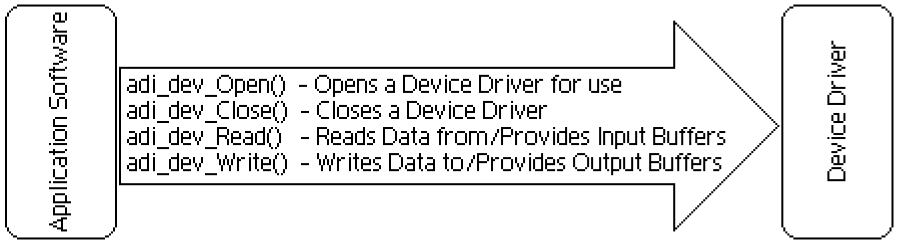

Application Software
adi_dev_Open() - Opens a Device Driver for use
adi_dev_Close() - Closes a Device Driver
adi_dev_Read() - Reads Data from/Provides Input Buffers
adi_dev_Write() - Writes Data to/Provides Output Buffers
Device Driver

Figure 58: Standard Device Driver Functions

An input buffer is employed to receive data from a device; while conversely, an output buffer contains data that is sent out to a particular device. The two main buffer categories are *1D* and *2D*. A *1D* buffer is comprised from a linear array of data that a device driver processes. On the other hand, a *2D* buffer is essentially a two-dimensional array (rows and columns) of data that a device driver manages. *2D* buffers are used in this application as they are more efficient in terms of video processing than a *1D* buffer. A *2D* buffer is comprised of the following fields [18], [69]:

- *pData*: A pointer to relevant data which can exist anywhere in memory.
- *ElementWidth*: Width of each element in terms of bytes to be read in or sent out.
- *XCount*: Specifies the number of column elements.
- *XModify*: Indicates the number of bytes to increment the address pointer after each column transfer.
- *YCount*: Specifies the number of row elements.
- *YModify*: Indicates the number of bytes to increment the address pointer after each row transfer.
- *CallBackParameter*: Null or non-null value. The idea of *Callback* is explained shortly.
- *pNext*: Pointer to the next *2D* buffer in the chain. The concept of chaining is expanded upon shortly. This parameter is assigned null if the buffer is the last/only buffer in a chain.

The concept of *Callback* with reference to *2D* buffers involves invoking a regular *C* function in response to an asynchronous event such as an interrupt. The *VisualDSP++* compiler incorporates two categories of *Callback* [69]:

1. *Live Callback*
2. *Deferred Callback*

For *Live Callbacks*, a *C* service routine is invoked as soon as an asynchronous event occurs. Conversely, for *Deferred Callbacks* a *C* function is not summoned until a short time after an asynchronous event occurs. Thus, *Live Callbacks* typically occur at hardware interrupt time (higher priority interrupt level); whereas *Deferred Callbacks* execute at software interrupt time (lower priority interrupt level). As a consequence, the use of *Live Callbacks* can have a detrimental effect on performance as associated interrupt latencies are high. Therefore *Deferred Callbacks* are used in the vast majority of applications as they possess lower interrupt latencies.

With reference to the *CallBackParameter* field of a *2D* buffer, if a value of null is assigned *Live Callbacks* are utilised. In addition, a device driver does not "call back" an application after a buffer has been processed. If the *CallBackParameter* field is allotted a non-null value, *Deferred Callbacks* are employed invoking an application's *Callback* function after a buffer has been processed by a particular device driver.

The *pNext* parameter of a *2D* buffer can be used to link numerous buffers together in a chain-like manner. Typically, with video applications a *Chained Loopback* dataflow method is used to mutually tie several buffers together [68]. This fundamentally means that the last element of one particular buffer points to the first data member of a different buffer. Therefore buffers are essentially queued one-by-one to a device driver ensuring that the component, e.g. a video encoder IC, is never starved of data. This is obviously critical in video applications.

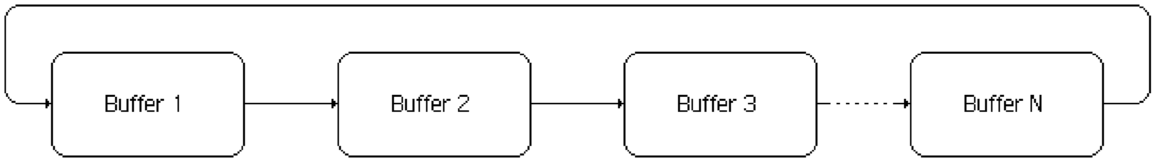


Figure 59: Chained Loopback Dataflow Methodology

System services are in essence pre-built software libraries that simplify software development and provide efficient access into components such as DMA and dynamic power modules etc [68].

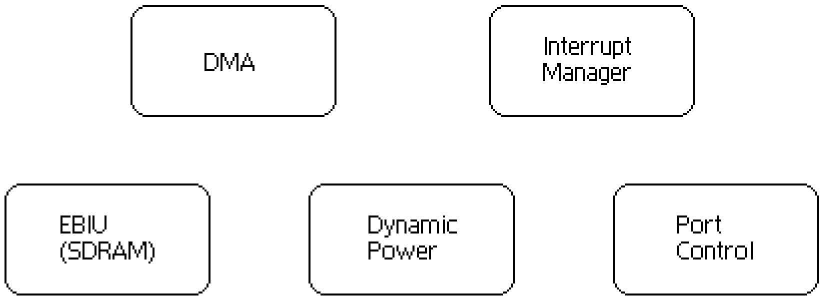


Figure 60: Examples of System Services Supported

They are used in conjunction with device drivers in order to control and interact with internal modules and external peripherals. Device drivers manage their own system services as required. To utilise device drivers and system services software algorithms must include the appropriate header files in the following order.

```
#include <services/services.h> // System Services Header File
#include <drivers/adi_dev.h> // Device Manager Header File
#include <drivers/X.h> // Device Driver X's Header File
```

System services are initialised prior to the configuration of device drivers. The *adi_dev_Init()* function is used to initialise a device driver. *VisualDSP++'s* device drivers are built on top of its system services [68], [69].

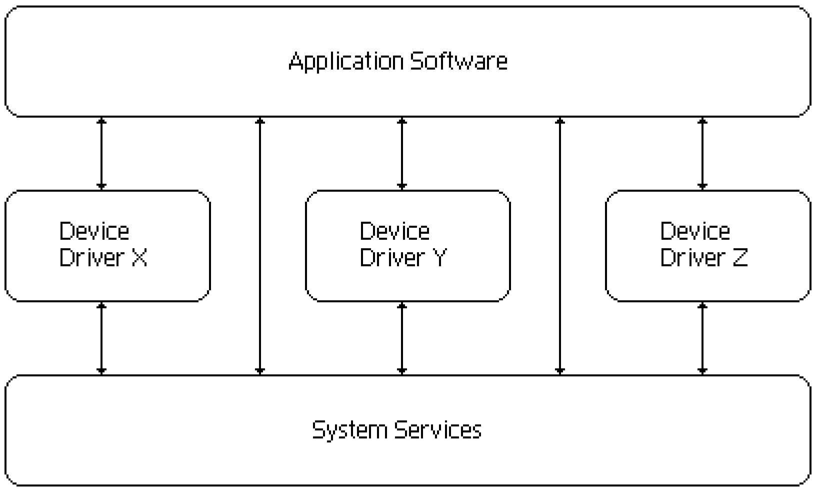
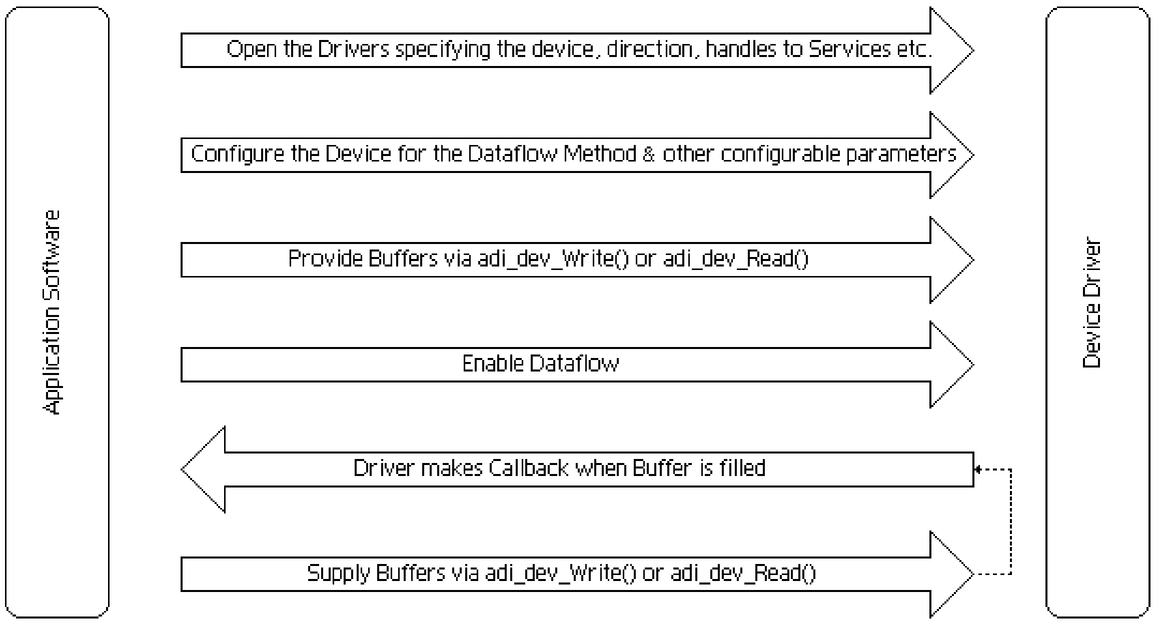


Figure 61: Layered Utilities Structure

A typical programming sequence for the utilisation of device drivers and system services is seen below.



Application Software
Open the Drivers specifying the device, direction, handles to Services etc.
Configure the Device for the Dataflow Method & other configurable parameters
Provide Buffers via adi_dev_Write() or adi_dev_Read()
Enable Dataflow
Driver makes Callback when Buffer is filled
Supply Buffers via adi_dev_Write() or adi_dev_Read()
Device Driver

Figure 62: Typical Device Driver Programming Sequence

The device drivers for the ADV7179 video encoder IC [70] and DMA module are employed in this system to implement video processing. The methodology used to realise this is discussed in the next section. Before discussing this however, it must be pointed out that the final implementation of the video encoder device driver module was only being developed by *Analog Devices* at the same time as the author was trying to utilise this particular device driver within this test algorithm. As a result, contact was made with *Analog Device's* support-team on numerous occasions to try and eradicate teething problems. However, this was sometimes in vein as the support team were only at the same development stage as the author. Consequently the author was engaged in many debug sessions in order to successfully implement the test algorithm.

6.2.1.2 Software Testing of Video Implementation

The strategy for testing the correct implementation of video processing upon the *Blackfin ADSP-BF537 EZ Kit Lite* and its A/V daughter board involved alternating the colour displayed upon a television monitor. This included creating two *ITU-R BT.656* buffers in SDRAM. Two *2D* buffers chains were declared to ensure that the ADV7179 IC was never starved of data. Once all of the device drivers, handles etc. were declared the two *ITU-R BT.656 4:2:2* frames were initialised with different colours. The ADV7179 device driver handle was then enabled and fed the data contained within the two buffers via a DMA transfer. The correct operation of the test algorithm was verified by monitoring a television screen to see if the desired colours were displayed. The flow chart below illustrates the processes undertaken to achieve successful operation. The source code for this test algorithm is found in *Appendix E.*

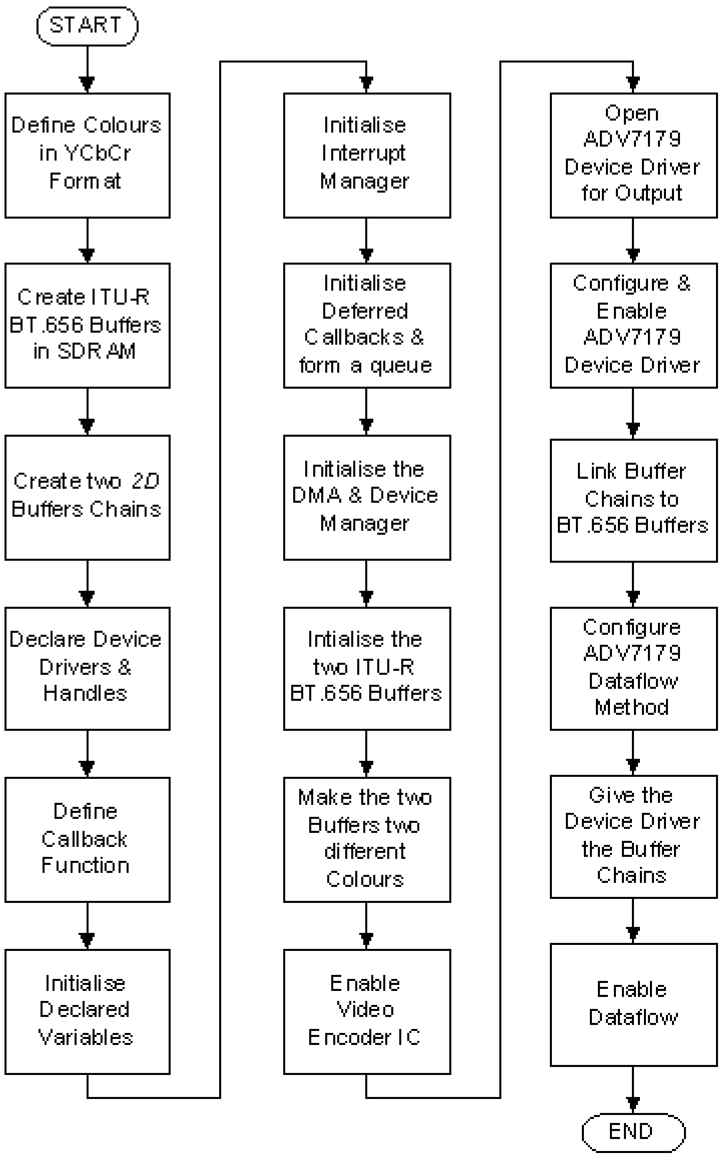

START
Define Colours in YCbCr Format
Create ITU-R BT.656 Buffers in SDRAM
Create two 2D Buffers Chains
Declare Device Drivers & Handles
Define Callback Function
Initialise Declared Variables
Initialise Interrupt Manager
Initialise Deferred Callbacks & form a queue
Initialise the DMA & Device Manager
Intialise the two ITU-R BT.656 Buffers
Make the two Buffers two different Colours
Enable Video Encoder IC
Open ADV7179 Device Driver for Output
Configure & Enable ADV7179 Device Driver
Link Buffer Chains to BT.656 Buffers
Configure ADV7179 Dataflow Method
Give the Device Driver the Buffer Chains
Enable Dataflow
END

Figure 63: Flowchart for Video Processing Test Program

Numerous colours were declared in *YCbCr* format. The period of time for which an individual colour was displayed was dependant upon the value of the *NUM_BUFFERS* constant. As seen below, for a value of 30 assigned to *NUM_BUFFERS* a colour change rate of one second was achieved.

```
#define NUM_BUFFERS (30) // Colour Change Rate = (NUM_BUFFERS/30)/second
// Colour Patterns
static u8 black[] = {0x80,0x10,0x80,0x10}; // Black pixel YCbCr format
static u8 blue[] = {0xF0,0x29,0x6E,0x29}; // Blue pixel YCbCr format
static u8 red[] = {0x5A,0x51,0xF0,0x51}; // Red pixel YCbCr format
static u8 magenta[] = {0xCA,0x6A,0xDE,0x6A}; // Magenta pixel YCbCr format
static u8 green[] = {0x36,0x91,0x22,0x91}; // Green pixel YCbCr format
static u8 cyan[] = {0xA6,0xAA,0x10,0xAA}; // Cyan pixel YCbCr format
static u8 yellow[] = {0x10,0xD2,0x92,0xD2}; // Yellow pixel YCbCr format
static u8 white[] = {0x80,0xEB,0x80,0xEB}; // White pixel YCbCr format
```

Two buffer frames were declared, *PingFrame* and *PongFrame*, both of which were configured to hold the contents of an *ITU-R BT.656* frame. In this particular case the two arrays were initialised to hold a *NTSC* video frame. *NTSC* was chosen solely for test purposes; the choice of *PAL* or *NTSC* in this case is irrelevant.

```
// Create two areas in SDRAM that will each hold a 656 frame
static u8 PingFrame[ADI_ITU656_NTSC_LINE_WIDTH * ADI_ITU656_NTSC_HEIGHT];
static u8 PongFrame[ADI_ITU656_NTSC_LINE_WIDTH * ADI_ITU656_NTSC_HEIGHT];
```

Two *2D* buffer chains, one for *PingFrame* and one for *PongFrame*, were declared in order to create a chaining method for the data fed to the ADV7179 device driver. Both buffer chains consisted of a number of elements determined by the *NUM_BUFFERS* constant.

```
ADI_DEV_2D_BUFFER PingBuffer[NUM_BUFFERS]; // Create Two Buffer Chains
ADI_DEV_2D_BUFFER PongBuffer[NUM_BUFFERS];
```

Several device drivers were used in this test algorithm. The DMA device driver was incorporated to facilitate efficient data transfers between SDRAM and the *PPI* port. The DCB (Deferred Callback) device driver was utilised as *Deferred Callbacks* were used to improve performance as outlined in *Section 6.2.1.1*.

```
// DMA Manager data (base memory + memory for 1 DMA channel)
static u8 DMAMgrData[ADI_DMA_BASE_MEMORY + (ADI_DMA_CHANNEL_MEMORY *
1)];
// Deferred Callback Manager data (memory for 1 service + 4 posted callbacks)
```

```
static u8 DCBMgrData[ADI_DCB_QUEUE_SIZE + (ADI_DCB_ENTRY_SIZE)*4];
```

The ADV7179 device driver was also employed within this system. It was used to control the ADV7179 video encoder IC in order to transform a digital *ITU-R BT.656* video stream into an analogue television signal. Recall from *Section 6.2.1.1* that memory is required by device drivers in order to manage components and that this memory is supplied at the initialisation stage of a program. During initial development of this test program an error involving the use of the ADV7179 device driver module was encountered.

```
// Device Manager Driver
static u8 DevMgrData[ADI_DEV_BASE_MEMORY + (ADI_DEV_DEVICE_MEMORY * 1)];
```

After consultation with *Analog Device's* support team and a review of [71] it was found that the ADV7179 IC, situated on the A/V daughter board, is connected to the *Blackfin ADSP-BF537 EZ Kit Lite* using two peripherals. Firstly, the *PPI* port is used for transferring video data to the encoder; while the *SPI* is used to control the ADV7179 IC. Thus the ADV7179 device driver automatically and transparently opens and controls the underlying *PPI* driver to move data through the encoder. It also opens and controls the underlying *SPI* driver to configure the ADV7179. This can be thought of as a stacked approach where the application talks exclusively to the ADV7179 device driver while the ADV7179 driver talks to the underlying *PPI* and *SPI* drivers as necessary.

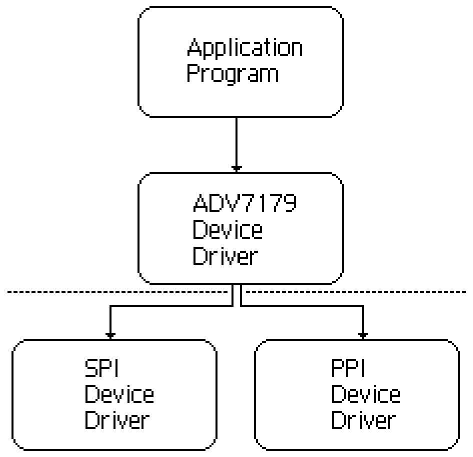


Figure 64: Stacked Approach of Device Drivers

Subsequently, to overcome the encountered error, memory for three device drivers had to be allocated for a single call to the ADV7179 device driver as it implicitly incorporates the *PPI* and *SPI* device drivers.

```
// Device Manager data (base memory + memory for 3 devices)
// Memory for 3 devices is required because usage of a 7179 device results in the usage of the PPI
// and SPI devices.
static u8 DevMgrData[ADI_DEV_BASE_MEMORY + (ADI_DEV_DEVICE_MEMORY * 3)];
```

Handles for the utilised device drivers were declared.

```
ADI_DEV_DEVICE_HANDLE AD7179DriverHandle; // Handle to the ADV7179 Driver
ADI_DCB_HANDLE DCBManagerHandle; // Handle to the Callback Service Manager
ADI_DMA_MANAGER_HANDLE DMAManagerHandle; // Handle to the DMA Manager
ADI_DEV_MANAGER_HANDLE DeviceManagerHandle; // Handle to the Device Manager
```

The *Callback* function in this application was invoked as soon as the *PPI* completed the processing of the last component in the buffer chains. Within the *Callback* function the *pNext* value of the last elements of both buffer chains was assigned the address of the first element within each individual chain in order to prevent starvation of data to the video encoder IC.

In order to initialise the DCB manager with sufficient memory for the required number of *Deferred Callback* queues the *adi_dcb_Init()* and *adi_dcb_Open()* functions were used. The DMA and ADV7179 device drivers were initialised using the *adi_dma_Init()* and *adi_dev_Init()* functions – see *Appendix E*.

The two frames, *PingFrame* and *PongFrame*, were configured to hold a progressive scan *ITU-R BT.656 4:2:2 NTSC* frame. This was achieved using a pre-written system services function, *adi_itu656_FrameFormat()*, which initialises a frame with the necessary *SAV*, *EAV*, preambles etc. required for an *ITU-R BT.656* video stream – see *Appendix E*.

```
adi_itu656_FrameFormat (PingFrame,ADI_ITU656_NTSC_PR);
adi_itu656_FrameFormat (PongFrame,ADI_ITU656_NTSC_PR);
```

Once the frames were initialised their *ITU-R BT.656* chrominance fields were filled with a particular colour; in this case white and blue respectively. Again, this was achieved using a pre-written system services function, *adi_itu656_FrameFill()* – see *Appendix E*.

```
adi_itu656_FrameFill (PingFrame,ADI_ITU656_NTSC_PR,white);
adi_itu656_FrameFill (PongFrame,ADI_ITU656_NTSC_PR,blue);
```

After resetting the ADV7179 IC through software and initialising the AV7179 device driver, the application opened the video encoder IC for use using the *adi_dev_Open()* function. This function also prescribed the inclusion of DMA transfer between SDRAM and the ADV7179 as the DMA handle was passed as the seventh parameter.

```
// Open the AD7179 Driver for Output
ezErrorCheck(adi_dev_Open(DeviceManagerHandle, // Handle controlling the Device
                          &ADIADV7179EntryPoint, // Address of Entry Point
                          ENCODER_PPI,  // Number identifying which Device is Opened
                          NULL, // No Client Handle
                          &AD7179DriverHandle, // Handle Address
                          ADI_DEV_DIRECTION_OUTBOUND, // Data Direction
                          DMAManagerHandle, // Handle to DMA Manager
```

```
                        DCBManagerHandle, // Handle to Callback Manager
                        Callback)); // Callback
```

However, problems were encountered using this function. The *Blackfin ADSP-BF537* processor contains a single *PPI* port. The third parameter passed to *adi_dev_Open()* contains a number that identifies which device is to be opened. Initially *ENCODER_PPI* was assigned a value of one to indicate the *PPI* device number. This resulted in nothing being displayed upon the television monitor when the program was tested. After reviewing [71] it was found that devices exploiting *VisualDSP++* utilities are numbered with a zero base; e.g. if there are four *PPI* ports the first is assigned zero for identification purposes, the second is assigned one etc. Subsequently to solve the encountered error *ENCODER_PPI* was assigned a value of zero.

The *adi_dev_Control()* function was used to configure the ADV7179 device driver for data flow and open the *PPI* port for data transfer.

```
// Set PPI Device Number
ezErrorCheck(adi_dev_Control(AD7179DriverHandle, // Handle identifying Device
                        // Command Identifier
                        ADI_ADV717x_CMD_SET_PPI_DEVICE_NUMBER,
                        (void*)0)); // PPI Device Number

// Open PPI Device
ezErrorCheck(adi_dev_Control(AD7179DriverHandle, // Handle identifying Device
                        ADI_ADV717x_CMD_SET_PPI_STATUS, // Command Identifier
                        // Address of Command Specific Parameter
                        (void*)ADI_ADV717x_PPI_OPEN));
```

The two buffer chains were linked to *PingFrame* and *PongFrame* respectively in order to form a loopback mechanism to ensure that data was constantly being fed from SDRAM to the ADV7179 video encoder. Recall that each buffer chain contained *NUM_BUFFERS* elements. As illustrated in the following diagram all of the elements within both chains, *PingBuffer* and *PongBuffer*, pointed to *PingFrame* and *PongFrame* respectively.

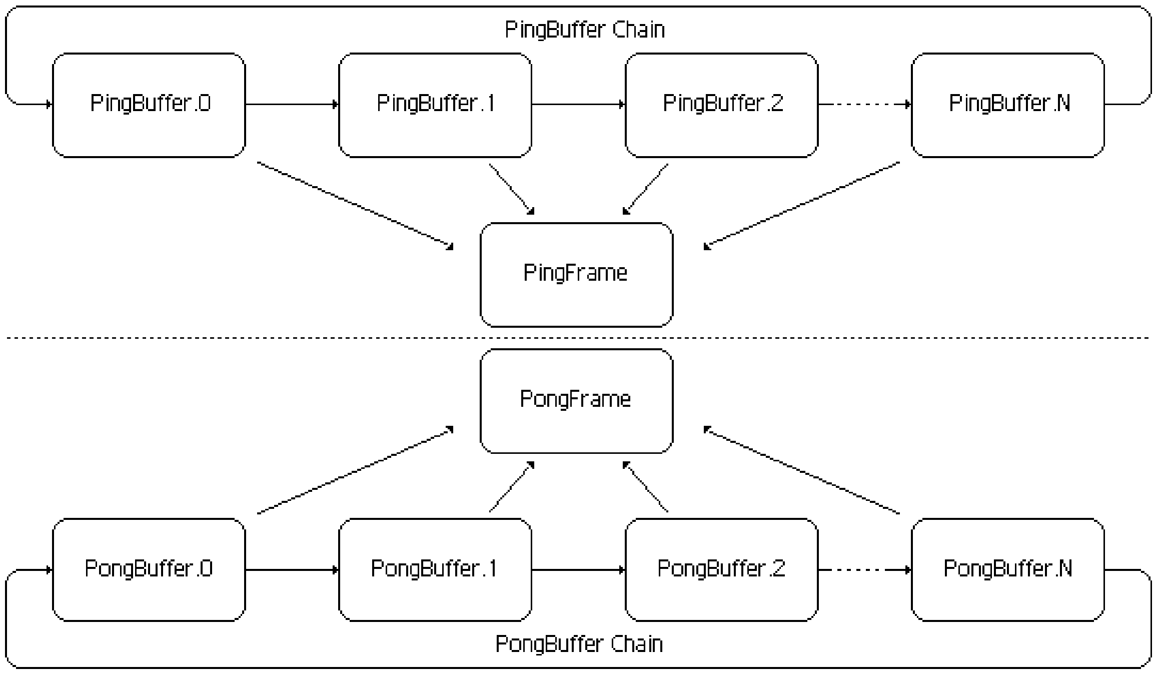


Figure 65: Elements of Buffer Chains pointing to ITU-R BT.656 Frames

This was accomplished in software by appropriately configuring the fields of all elements of both *2D* buffers; the configuration for *PingBuffer* is seen below.

```
for (i = 0; i < NUM_BUFFERS; i++) // Populate the PingBuffer
{
  PingBuffer[i].Data = PingFrame; // Point to PingFrame Data
  PingBuffer[i].ElementWidth = 2;
  PingBuffer[i].XCount = (ADI_ITU656_NTSC_LINE_WIDTH/2);
  PingBuffer[i].XModify = 2;
  PingBuffer[i].YCount = ADI_ITU656_NTSC_HEIGHT;
  PingBuffer[i].YModify = 2;
  PingBuffer[i].CallbackParameter = NULL;
  PingBuffer[i].pNext = &PingBuffer[i + 1];
}
// Callback on last buffer in chain, consequently point to first buffer in chain.
PingBuffer[NUM_BUFFERS - 1].CallbackParameter = &PingBuffer[0];
```

```
PingBuffer[NUM_BUFFERS - 1].pNext = NULL;
```

As mentioned earlier, as soon as the processing of the last buffer in the chain was terminated a *Callback* was issued in order to re-queue the data; i.e. the last buffer in the chain points back to the first buffer element. Again, this mechanism was utilised to ensure that a video stream was constantly being fed to the ADV7179.

The *adi_dev_Control()* function was again incorporated to configure the ADV7179 device driver for outbound loopback data flow.

```
// Configure the AD7179 Dataflow Method
ezErrorCheck(adi_dev_Control(AD7179DriverHandle, // Handle identifying Device
                ADI_DEV_CMD_SET_DATAFLOW_METHOD, // Command Parameter
                (void *)ADI_DEV_MODE_CHAINED_LOOPBACK)); // Outbound Loopback
```

The next step involved actually pointing the ADV7179 device driver towards the buffer chains and turning on the data flow to allow transmission of an *ITU-R BT.656 4:2:2* video stream to the ADV7179 video encoder IC.

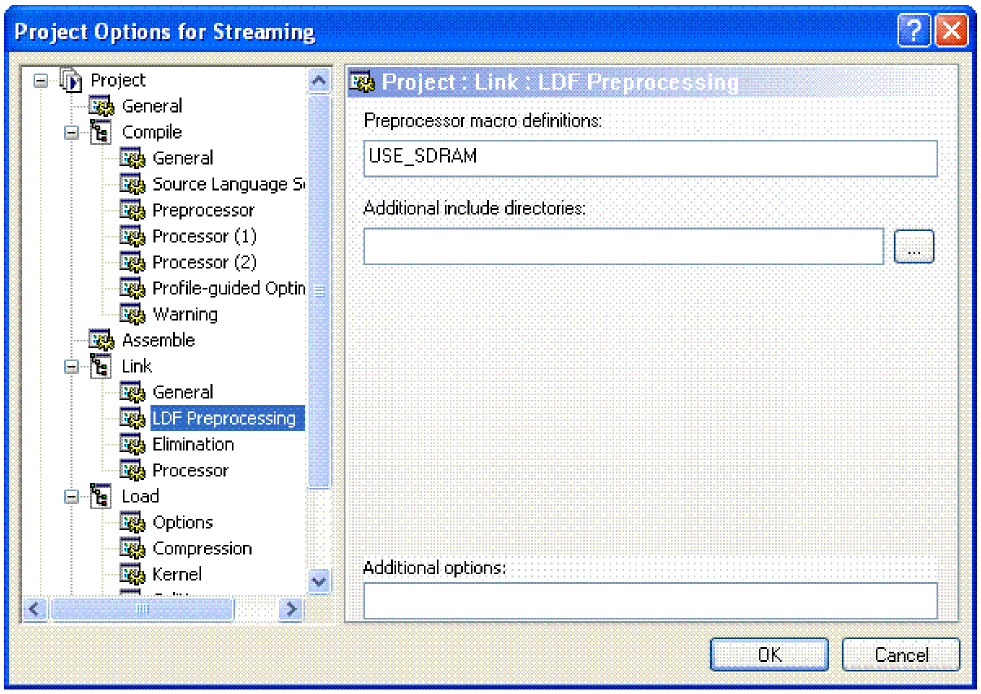
```
// Give the device the Ping and Pong buffer chains
ezErrorCheck(adi_dev_Write(AD7179DriverHandle, // Handle identifying Device
                ADI_DEV_2D, // 2D Buffer
                (ADI_DEV_BUFFER *)&PingBuffer)); // Point to PingBuffer

ezErrorCheck(adi_dev_Write(AD7179DriverHandle, // Handle identifying Device
                ADI_DEV_2D, // 2D Buffer
                (ADI_DEV_BUFFER *)&PongBuffer)); // Point to PongBuffer

// Enable data flow
ezErrorCheck(adi_dev_Control(AD7179DriverHandle, // Handle identifying Device
                ADI_DEV_CMD_SET_DATAFLOW, // Command Parameter
                (void *)TRUE)); // Turn on Dataflow
```

The reader may have noticed that most function calls discussed so far incorporated a call to *ezErrorCheck()*. This was used as a debug aid as all calls to *VisualDSP++* system services functions return a value indicating the success/failure of a particular invoked algorithm. If a specific function call returned unsuccessfully *ezErrorCheck()* illuminated LEDs located on the *Blackfin ADSP-BF537 EZ Kit Lite* development board to indicate a fault. Otherwise *ezErrorCheck()* took no action.

After eradicating all software debugs it was found that the program did not function as desired; i.e. a television screen did not display alternate colours when connected to the A/V daughter board. After reviewing [71] once more it was found that the problem resided in the project options of *VisualDSP++*. As this test algorithm incorporated SDRAM the pre-processor macro definition *USE_SDRAM* needed to be included in the project options in order to enable the use of SDRAM upon the *EZ Kit* development board.



Preprocessor macro definitions: USE_SDRAM

Figure 66: Required Pre-Processor Macro Definition

Once this problem was eliminated, the test algorithm operated as desired; i.e. a television screen connected to the A/V board continuously displayed alternate screens of white and blue. This therefore proved that the *ITU-R BT.656* video stream was being correctly initialised within SDRAM and transferred successfully, via DMA, to the ADV7179 video encoder IC. The encoder itself correctly converted the digital video stream into a standard analogue television signal. Thus a formula for video processing utilising the *Blackfin ADSP-BF537 EZ Kit Lite* and A/V development boards has been determined.

6.2.2 Video Hardware Strategy

The main focus so far has been on the software strategy implemented to realise video processing. However a certain degree of hardware configuration strategy, albeit relatively small, also took place involving signal routing on the *ADSP-BF537 EZ Kit Lite* and A/V development boards. This was conducted by modifying jumper settings upon the A/V daughter board. The correct hardware settings in conjunction with an accurate software strategy led to the successful operation of this test program.

The following table illustrates the jumper settings instigated on the A/V daughter board. Note that the table only contains details of jumper settings that were relative to this particular video processing implementation. An individual jumper contains a number of pins; so for example the description JP3.5/7 refers to the connection of pins 5 and 7 of *Jumper 3* [72].

Jumper Number	Pin Connections	Outcome
JP3	JP3.5/7	ADSP-BF537's TWI Interface is utilised to potentially reconfigure the ADV7179 IC
	JP3.6/8	
JP4	JP4.1/2	Connects 27MHz A/V On-board Clock to Video Encoder
	JP4.3/4	
JP5	JP5.3/4	Enables PPI0 to drive ADV7179 IC
JP8	JP8.1/3	Selects PPI0 as the source for the frame synchronisation signals for ADV7179 IC
	JP8.2/4	
	JP8.7/8	Enables output data synchronisation signal
JP9	JP9.1/3	Video Encoder Reset with Flag Pin

Table 16: A/V Daughter Board Jumper Settings

Jumper 3 is the *TWI*[3] (Two Wire Interface) source selection jumper [42], [72]. It is used to select between a software emulated and actual *TWI* interface. The ADV7179 IC is reconfigurable using a *TWI* interface. Consequently this jumper is required as some *Blackfin* processors do not contain a *TWI* interface; therefore they need to emulate the interface in software. On the other hand, the *ADSP-BF537* does contain a *TWI* interface; subsequently JP3.5/7 and JP3.6/8 were installed.

The ADV7179 video encoder requires some sort of clocking mechanism. The A/V daughter board contains an on-board 27MHz oscillator. JP4.1/2 and JP4.3/4 were used to route the 27MHz clock signal to the ADV7179. *Jumper 5* dictates where the video encoder incorporated onto the A/V daughter board sources its video data from. As the *ADSP-BF537* contains a single *PPI* port, namely *PPI0*, JP5.3/4 was used to route the *ADSP-BF537's PPI* data to the ADV7179 [72].

[3] TWI is a communications protocol used in small industrial networks. Its operation is similar to I^2C.

JP8.1/3 and JP8.2/4 were used to select the *ADSP-BF537's PPI* port as the source for synchronisation signals feeding the video encoder. JP8.7/8 was inserted to enable the output video synchronisation signals from the ADV7179 IC. *Jumper 9* is partly used to select between the resources utilised to reset the ICs incorporated onto the A/V board. In this particular test algorithm a software reset was utilised therefore JP9.1/3 was inserted. Again to reiterate, the correct hardware settings outlined in conjunction with the described software strategy led to the successful operation of this test program.

6.3 Summary

This chapter described the test strategy employed for video processing utilising the *Blackfin ADSP-BF537 EZ Kit Lite* and A/V development boards. The core issues to note are as follows:

- *VisualDSP++'s* device drivers and system services are powerful resources which aid in this video synthesis.

- A test algorithm involving the display of analogue video signals, derived from an *ITU-R BT.656* video stream in SDRAM, upon a television monitor has been developed.

- Numerous software and hardware modifications were made to the initial testing strategy resulting in a successful outcome; i.e. alternate colours were continuously displayed on a television monitor.

Chapter 7 - Application Synthesis

7.1 Introduction

This chapter details the efforts involved in the implementation of this application's synthesis by combining the developed CAN and video mechanisms. The information given in this chapter is separated into numerous sections as outlined below:

- An explanation of the initial strategy undertaken to develop the application's synthesis.

- A description of the problems encountered with the initial synthesis strategy and the debug session embarked on to identify the cause of the occurring errors and their eradication with fitting solutions.

- An outline of the final strategy utilised and the test procedure incorporated to assess its correct functionality.

- A depiction of future developments which could be made to further improve the system's performance.

7.2 Displaying Simulated Vehicle Data

Over the last two chapters suitable procedures have been established for CAN networking and video processing. As a result, the next step is to combine the two independent strategies to endeavour to formulate the synthesis of this application. This essentially involves utilising the CAN strategy to receive simulated vehicle data from the network and pass this information over to the video processing module in order to visually represent the data appropriately on a television monitor.

The general hypothesis behind the amalgamation of the two devised procedures is to exploit the contents of received CAN messages to manipulate global variables in software, resulting in the modulation of a monitor's display. This is carried out repeatedly in software resulting in a television screen being constantly updated with live vehicle information.

7.2.1 Initial Strategy

Originally the CAN and video processing strategies were merged together with the initialisation sequence for various parameters as illustrated in the following diagram. The functions that were utilised in this particular implementation can be found (not in the same order as the subsequent diagram) in *Appendix F*.

At first, it was decided to use only CAN messages received with ID 1041 to modulate a monitor's display. Thus, recalling from *Section 5.5*, only messages received into *Blackfin's Mailbox 7* were employed to vary a television screen's display. This approach was taken in order to develop the application's synthesis in steps; i.e. implement screen modulation using one particular CAN message and then develop on from this. Message ID 1041 was chosen to represent vehicle speed and, at the outset, it was decided to signify this quantity by varying the colour displayed on the screen. A CAN network of 500kBits/s was again implemented as it satisfactorily transferred simulated vehicle data at an appropriate rate.

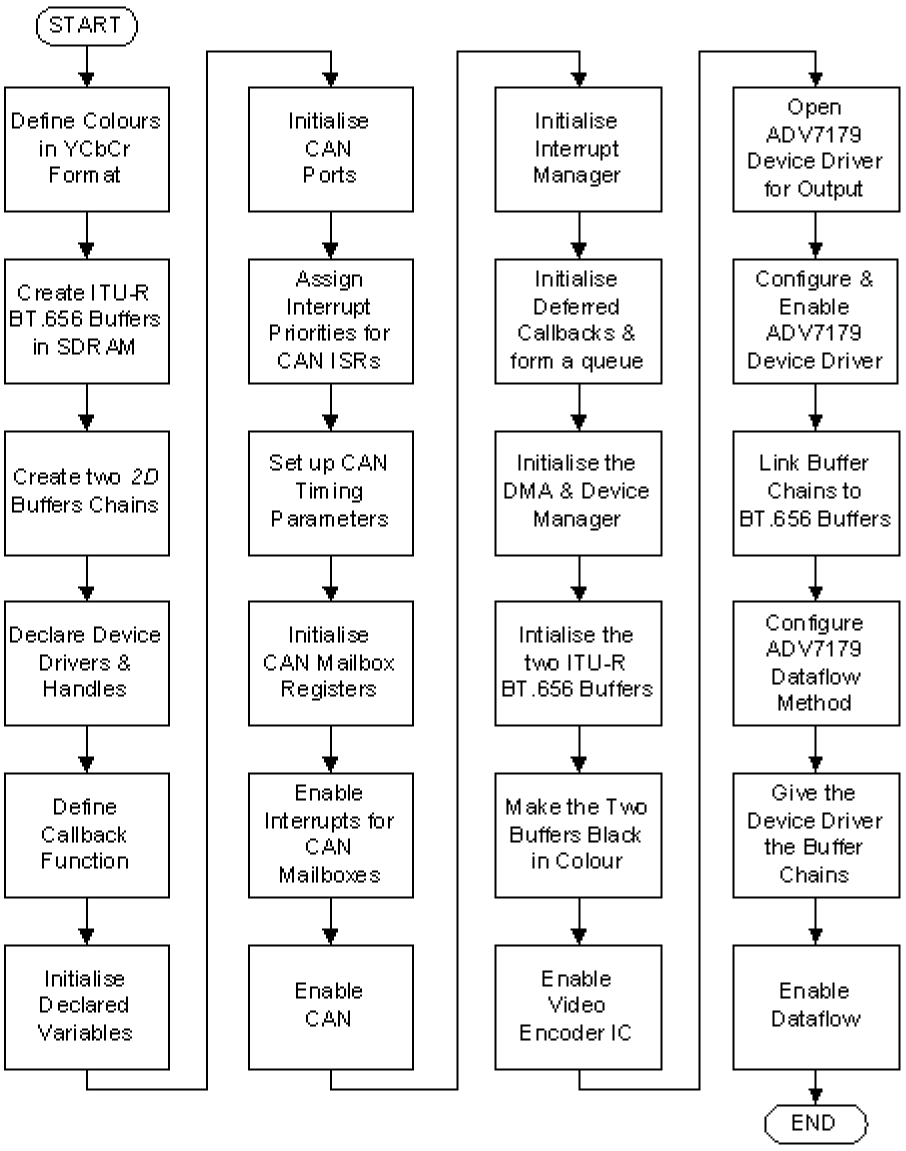

START
Define Colours in YCbCr Format
Create ITU-R BT.656 Buffers in SDRAM
Create two 2D Buffers Chains
Declare Device Drivers & Handles
Define Callback Function
Initialise Declared Variables
Initialise CAN Ports
Assign Interrupt Priorities for CAN ISRs
Set up CAN Timing Parameters
Initialise CAN Mailbox Registers
Enable Interrupts for CAN Mailboxes
Enable CAN
Initialise Interrupt Manager
Initialise Deferred Callbacks & form a queue
Initialise the DMA & Device Manager
Intialise the two ITU-R BT.656 Buffers
Make the Two Buffers Black in Colour
Enable Video Encoder IC
Open ADV7179 Device Driver for Output
Configure & Enable ADV7179 Device Driver
Link Buffer Chains to BT.656 Buffers
Configure ADV7179 Dataflow Method
Give the Device Driver the Buffer Chains
Enable Dataflow
END

Figure 67: Flowchart of Original Initialisation Strategy

To utilise the information received into *Mailbox 7* an ISR, *CAN_RCV_HANDLER*, was developed which varied the value of a global variable, *clr_screen*, depending upon the contents of the mailbox's *CAN_MB7_DATA3* register. Once all initialisation was complete, this ISR was invoked as soon as a CAN message was received by the *Blackfin*. The value of the *clr_screen* global variable was used within the *main()* function to attempt to vary the colour displayed upon the monitor. As seen below, within this initial implementation a value of *clr_screen* equal to two depicted red; while a value of one represented blue. Note that the developed ISR included code to process a message received into *Mailbox 6*. However, this was only incorporated to allow for future expansion because as stated earlier, an approach was taken to develop the application's synthesis in steps.

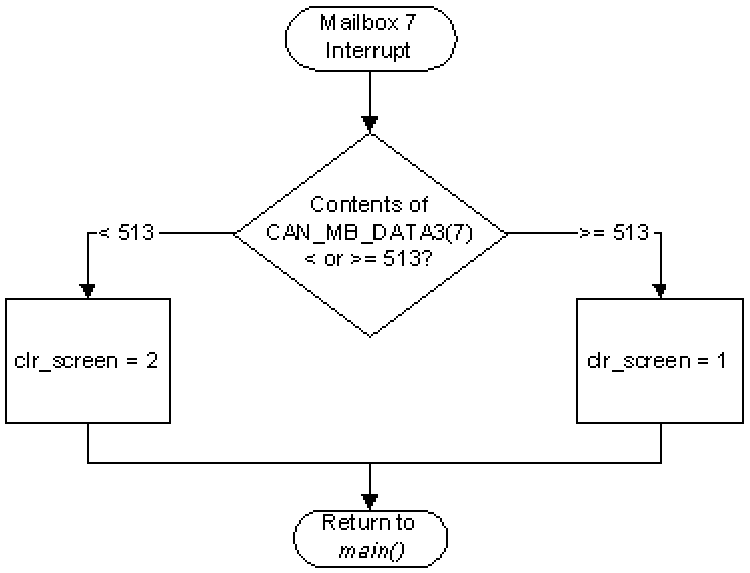


Figure 68: Received CAN Message Interrupt Service Routine

```
if(bit_pos = = 0x8) //if Mailbox7 IRQ // ISR for Mailbox 7
{
  if((*(pCAN_MB_DATA3(7)) = = 0) || (*(pCAN_MB_DATA3(7)) <= 512))
  {
     clr_screen = 2; // Display RED
  }
  …
  if(*(pCAN_MB_DATA3(7)) >= 513)
  {
     clr_screen = 1; // Display BLUE
  }
}
```

Within the *main()* function the *clr_screen* variable was evaluated using a *switch-case C* statement and the appropriate colour was written to *PingFrame* and *PongFrame* in *ITU-R*

BT.656 NTSC format. Note NTSC was chosen for use at this stage as it is default format for the ADV7179 IC; configuring the device for *PAL* usage requires developing software algorithms outside of the application's scope.

```
switch(clr_screen) // Determine which colour is displayed on the screen
{
   case(0): // Display WHITE
     adi_itu656_FrameFill (PingFrame, ADI_ITU656_NTSC_PR, white);
     adi_itu656_FrameFill (PongFrame, ADI_ITU656_NTSC_PR, white);
     break;

  case(1):  // Display BLUE
     adi_itu656_FrameFill (PingFrame, ADI_ITU656_NTSC_PR, blue);
     adi_itu656_FrameFill (PongFrame, ADI_ITU656_NTSC_PR, blue);
     break;
  …
  …
  case(6): //  Display YELLOW
     adi_itu656_FrameFill (PingFrame, ADI_ITU656_NTSC_PR, yellow);
     adi_itu656_FrameFill (PongFrame, ADI_ITU656_NTSC_PR, yellow);
     break;

  default: //Display BLACK
     adi_itu656_FrameFill (PingFrame, ADI_ITU656_NTSC_PR, black);
     adi_itu656_FrameFill (PongFrame, ADI_ITU656_NTSC_PR, black);
     break;
}
```

After writing the *ITU-R BT.656 4:2:2* data, two buffer chains were linked to the frames to form a loopback mechanism as discussed in the last chapter. Once this was completed the buffer chains were passed to the ADV7179 handle with the aim of displaying the information on a monitor. This entire procedure was carried out continuously in software by placing it within a *while(1)* loop.

7.2.1.1 Problems Encountered

When the developed application was tested it did not function as desired. The default colour, black, was continuously displayed upon a connected television monitor regardless of the value contained within the *CAN_MB7_DATA3* register of *Mailbox 7*. Consequently, a debug session was undertaken in an attempt to establish the likely cause of the error. After stepping through the source code it was found that the *CAN_RCV_HANDLER* ISR was not being correctly registered with the *Blackfin's CEC* (Core Event Controller) and *SIC* (System Interrupt Controllers) modules. These two modules are responsible for assigning priority levels and mapping ISRs. To program a particular ISR with a specific *IVG* (Interrupt Vector Group), i.e. priority level, the necessary bits in the appropriate *SIC_IARx* register must be configured accordingly. *IVG* levels range from 0 to 15; lower numbers possess higher priority while higher numbers bear lowest priority. *IVG* levels 7 to 15 are considered general purpose software or peripheral level interrupts. All other priority levels are reserved for supervisory ISRs; e.g. hardware errors [42].

The *CAN_RCV_HANDLER* was initially allocated an *IVG* level of eleven by assigning a value of 0x4 to bits 31 to 29 of *SIC_IAR1*. After configuring the priority of an individual ISR within the *Blackfin* it has to be registered with the *CEC* module. The *VisualDSP++* compiler incorporates two ways to accomplish this. Firstly, the utilisation of device drivers and system services can be used to automatically register an ISR. On the other hand an ISR can be manually registered using the *register_handler()* function. The *register_handler()* function was used to log the *CAN_RCV_HANDLER* with the *CEC*.

```
register_handler(ik_ivg9, CAN_RCV_HANDLER); // Register ISR with CEC
```

Conversely speaking, the devised video processing strategy incorporated device drivers and system services to register ISRs utilised within the DMA transfers. In the initialisation sequence illustrated in *Figure 67* the *CAN_RCV_HANDLER* was registered with the *CEC* and *SIC* prior to the logging of the ISRs utilised by the video processing strategy. The cause of the occurring problem was that the registering of the video processing ISRs overwrote the logging of the *CAN_RCV_HANDLER* ISR. Therefore, as far as the *Blackfin* was concerned, the *CAN_RCV_HANDLER* was no longer registered as an ISR. As a result of this, when this program was tested the ISR for *Mailbox 7* was never invoked; thus the global variable *clr_screen* remained at its default value of zero. This is why black was being continuously displayed on the connected television monitor.

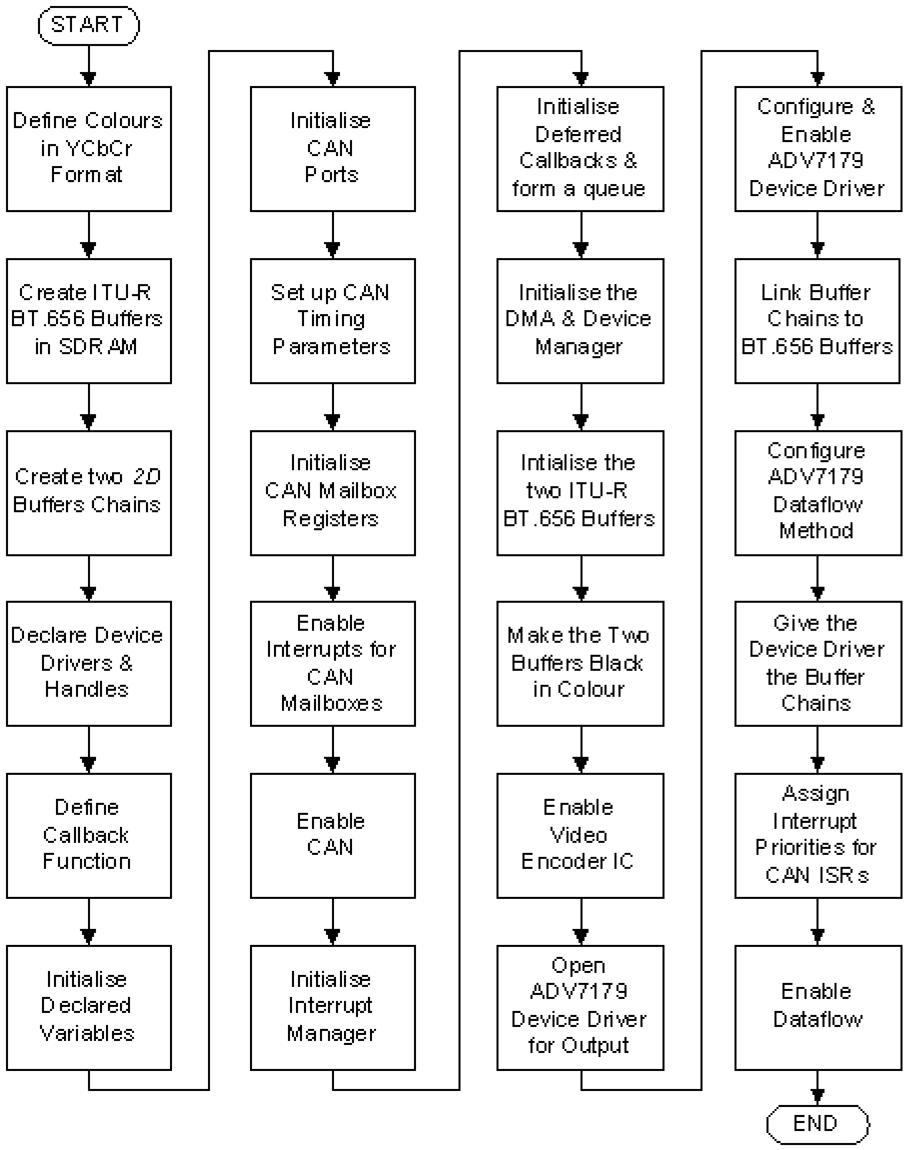

START
Define Colours in YCbCr Format
Create ITU-R BT.656 Buffers in SDRAM
Create two 2D Buffers Chains
Declare Device Drivers & Handles
Define Callback Function
Initialise Declared Variables
Initialise CAN Ports
Set up CAN Timing Parameters
Initialise CAN Mailbox Registers
Enable Interrupts for CAN Mailboxes
Enable CAN
Initialise Interrupt Manager
Initialise Deferred Callbacks & form a queue
Initialise the DMA & Device Manager
Intialise the two ITU-R BT.656 Buffers
Make the Two Buffers Black in Colour
Enable Video Encoder IC
Open ADV7179 Device Driver for Output
Configure & Enable ADV7179 Device Driver
Link Buffer Chains to BT.656 Buffers
Configure ADV7179 Dataflow Method
Give the Device Driver the Buffer Chains
Assign Interrupt Priorities for CAN ISRs
Enable Dataflow
END

Figure 69: Revised Initialisation Strategy

A solution to this problem was realised by subtly amending the initialisation strategy as seen in the preceding diagram. This initialisation sequence was almost identical to its predecessor with the exception that the assigning of interrupt priority for *CAN_RCV_HANDLER* did not take place until just before the enabling of dataflow. This essentially meant that the registering of the *CAN_RCV_HANDLER* ISR was appended onto the registering of the video processing ISRs using the *register_handler()* function. This initialisation strategy was employed within the final synthesis – see *Appendix F*.

When the revised application was tested it still did not operate as anticipated. The simulated vehicle speed data contained inside *Mailbox 7* did depict what was displayed upon the monitor. However, the screen's display did not update when the potentiometer used to mimic the operation of a sensor measuring speed was varied. Instead, the first CAN message received into *Mailbox 7* dictated what colour was statically displayed on the monitor. For instance, if the first CAN message received into *Mailbox 7* resulted in *CAN_MB7_DATA3* containing a value less than 513 the colour red was continuously displayed regardless of any further deviations in received data into this specific message buffer.

7.2.2 Final Strategy

A review was carried out to eradicate the reoccurring problem and establish how to exploit the received CAN data to continuously refresh the connected monitor's display with new information. It was found that once any device controlled through *VisualDSP++* utilities is opened the dataflow method for the specific device, particularly the DMA, must only be set once [68], [71]. This therefore was the fault in the initial synthesis strategy; the dataflow method for the ADV7179 device driver was configured during the initialisation stage and again inside the *while(1)* loop, thus violating standard procedure. Consequently, it can be seen that the employment of a *while(1)* loop to continuously update the connected television's display would not suffice as the dataflow method can only be configured once.

An alternative methodology was found through the exploitation of the *Callback* function. Recall that this function was invoked as soon as the *PPI* (via the ADV7179 device driver) completed the processing of the last component in the buffers chains. The *Callback* function was used to write the appropriate *ITU-R BT.656* chrominance information, representing vehicle speed, to the data buffers based on the value of the global variable *clr_screen*. In other words, the *switch-case C* statement outlined earlier was modified and incorporated into the *Callback* function:

```
switch(clr_screen) // Update data buffer with new colour
{
  case 0: // Fill frame with BLACK colour
    adi_itu656_FrameFill (pBuffer->Data,Frame,black);
    break;

  case 1: // Fill frame with BLUE colour
    adi_itu656_FrameFill (pBuffer->Data,Frame,blue);
    break;
  ...
  ...
  case 6: // Fill frame with YELLOW colour
    adi_itu656_FrameFill (pBuffer->Data,Frame,yellow);
    break;

  default: // Fill frame with WHITE colour
    adi_itu656_FrameFill (pBuffer->Data,Frame,white);
    break;
}
break;
```

Therefore, upon completion of processing of the last component in both buffer chains fresh data was given to the ADV7179 IC for conversion to a standard analogue video signal for display on the monitor – see *Appendix F*.

This exploitation of the *Callback* function meant that an interrupt policy, in opposition to a polling strategy embedded within a *while(1)* loop, was employed for continuously refreshing the ADV7179 video encoder with new data. This implicitly made more efficient use of both hardware and software resources. The interrupt policy involved initialising all of the incorporated *VisualDSP++* utilities and configuring the relevant components for dataflow etc. Once all of this was completed the system simply waited for suitable data to arrive for video processing via the CAN network, thus invoking *CAN_RCV_HANDLER*.

The value of *clr_screen* was again determined within the *CAN_RCV_HANDLER* ISR by evaluating the value contained within the *CAN_MB7_DATA3* register of *Mailbox 7*:

```
if(bit_pos = = 0x8) // if Mailbox7 IRQ
{
  if((*(pCAN_MB_DATA3(7)) >= 128) && (*(pCAN_MB_DATA3(7)) <= 255))
  {
    clr_screen = 1; // Display BLUE
  }

  if((*(pCAN_MB_DATA3(7)) >= 256) && (*(pCAN_MB_DATA3(7)) <= 383))
  {
    clr_screen = 2; // Display RED
  }
  ...
  ...
  if((*(pCAN_MB_DATA3(7)) >= 768) && (*(pCAN_MB_DATA3(7)) <= 895))
  {
    clr_screen = 6; // Display YELLOW
  }

  if(*(pCAN_MB_DATA3(7)) >= 896)
  {
    clr_screen = 7; // Display WHITE
  }
} // end if Mailbox 7
```

7.2.2.1 Testing of Final Strategy

The devised application methodology was tested by connecting the *Blackfin ADSP-BF537 EZ Kit Lite* development board to the constructed CAN network described in *Section 5.2*. In addition, a television monitor was interfaced to the video terminals of the A/V daughter board.

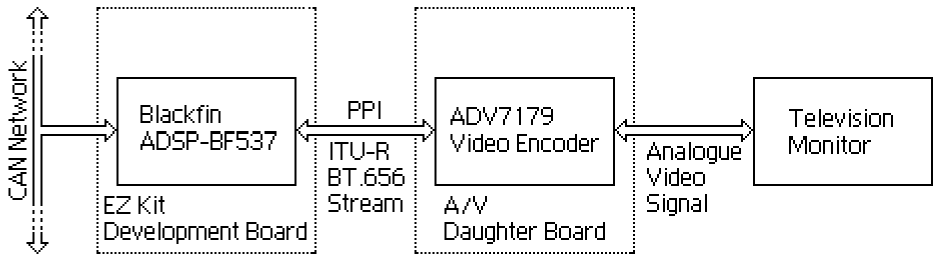


Figure 70: Application Components

Once the system was powered up the potentiometer incorporated onto the CAN On-Board node was rotated to simulate the action of an automobile sensor measuring speed. This resulted in the colour on the monitor's display varying in line with the rotation of the potentiometer; hence proving that the simulated vehicle data, transmitted via the CAN network, was correctly processed by the video module and represented visually. Therefore the application operated successfully as desired.

7.2.3 Capacity for Expansion

Earlier it was outlined that a modular approach was adopted in order to develop the application's synthesis in steps. However due to project timing constraints the devised methodology only accounted for the visual representation of a single CAN message's contents on a connected graphical display. Nonetheless, room for expansion in terms of CAN message processing is contained within the *CAN_RCV_HANDLER* ISR. As mentioned previously, source code for the processing of CAN messages received into *Mailbox 6* is incorporated into *CAN_RCV_HANDLER* to allow for future development.

Therefore it can be envisaged that the potential exists for the processing of suitably configured multiple CAN mailboxes. Consequently this allows for the handling of numerous quantities of simulated automobile data.

In addition simulated automobile data was represented visually upon a connected display monitor using colour information only. The derivation of a system to symbolise data in a graphical manner, using dial and gauges, was not fulfilled due to timing constraints. However a groundwork mechanism, incorporating CAN and video processing, which would form the cornerstone of a graphical system representing vehicle data, has been successfully established.

7.3 Summary

This chapter discussed the steps taken to formulate the system's implementation through the combination of the CAN and video processing methodologies described in previous chapters. The major points to behold are as follows:

- Care was taken with the initialisation sequence to ensure that all parameters were correctly configured in the appropriate order.

- To achieve desirable functionality the system was implemented with a full interrupt policy; i.e. once initialisation had concluded and all ISRs were defined the application only commenced processing when a particular interrupt occurred.

- Simulated vehicle data extracted from the constructed CAN network was visually represented using colour upon a connected display device.

- Due to project timing constraints it was not possible to devise a methodology to visually represent more than a single quantity of vehicle data at the same time; nor was it achievable to represent vehicle data in a graphical manner.

- The successfully devised application methodology can be used as the foundation stone of a graphical system used to symbolise vehicle data.

Chapter 8 - Conclusion

8.1 Introduction

This chapter summarises the research and methodologies carried out for this thesis, it outlines the results and conclusions that have been drawn from the project and offers suggestions on how to possibly further the research.

This research project commenced by outlining the protocols, technologies and components reviewed to formulate a suitable methodology for this application. *Chapter 2* discussed CAN with a view to automotive networking and highlighted the durability and reliability of the protocol. *Chapter 3* described to the reader the fundamentals of video processing and gave a detailed account of the *ITU-R BT.656* digital video standard. *Chapter 4* explained how numerous intelligent-devices were evaluated under several headings to establish the most suitable for employment in this research project.

The research project then moved on to synthesising the application with respect to the findings of *Chapters 2*, *3* and *4*. *Chapter 5* described the steps taken to simulate vehicle data and detailed how the CAN network employed in this system was constructed. *Chapter 6* discussed how a correct video module strategy was devised for the intelligent-device selected in *Chapter 4*. *Chapter 7* portrayed how the developed CAN and video algorithms were combined to formulate the overall application.

8.2 Conclusion

A system that visually represents simulated automobile data has been successfully implemented. The synthesised application illustrated, using colour variation, vehicle speed upon a connected graphical display. The operation of automobile sensing-devices was mimicked using potentiometers located within developed CAN nodes. The simulated speed data was received from the constructed CAN network operating at 500kBits/s. This information was then manipulated into *ITU-R BT.656* format by the selected *Blackfin ADSP-537* convergent processor. Once appropriately configured the data was given to a video encoder IC which converted the digital stream into an analogue video signal for display upon a connected television monitor.

Problems such as memory allocation, compiler glitches and ISR registration were overcome on the way to devising the successful system implementation. These problems were eradicated through the combination of the review of pertinent literature, consultation with the relevant bodies and software debugging.

The search for the appropriate intelligent-device required for this research project, with respect to the factors outlined in *Section 4.2*, yielded the *Blackfin ADSP-BF537* as the most suitable processor. This component adequately met all of the key considerations thus justifying it’s selection over the other devices evaluated.

As mentioned above the implemented system represented vehicle data using colour. The chrominance data displayed on the screen updated in synchronisation with the rotation of a specific potentiometer. Thus real-time vehicle data representation was achieved. This is obviously paramount in an actual implementation of this system within an automobile as a driver requires a live feed of critical vehicle data such as speed. However due to time constraints it was only possible to display a single item of simulated vehicle data on the monitor. The derivation of a system to symbolise multiple data items in a graphical manner, using dials and gauges, was not fulfilled. In conclusion, the devised system nevertheless possesses the potential to form the cornerstone of a graphical system to represent automobile data in real-time.

An actual implementation of this system would lead to economies of scale as the same graphical display could be incorporated into all vehicle models developed by a particular automotive manufacturer. Style variations between vehicle models could be still maintained by simply devising different software graphics for each model – see *Appendix G*.

8.3 Recommendations for Further Research & Development

As mentioned previously in *Section 7.2.3* the developed system could support the representation of multiple data items by expanding the devised software algorithms. In conjunction with this, the application's functionality could be enhanced by constructing a graphical mechanism to illustrate automobile information using dials and gauges; thus formulating a digital dash-display.

The monitor used in a practical implementation of the devised system would be much smaller than the television used in this project. It would be located where the dash-panel of an automobile is located; i.e. it would replace the analogue dash-panel located behind the steering wheel. This application displays video data dynamically upon the connected monitor. As a result the potential exists to integrate additional vehicle information into the graphical display. For example, GPS (Global Positioning System) systems are typically located on the centre console of a vehicle cabin. The GPS information could be displayed on the screen located behind the steering wheel resulting in the driver deviating his/her attention from the road for less time. Research into the actual benefits realisable with such a system would be similar to previous studies such as [73] and [74].

References

[1] V. Hillier, "*Fundamentals of Automotive Electronics, Second Edition*", Stanley Thornes Ltd., 1996.

[2] Bosch Inc., "*CAN Specification Version 2.0*", September 1991.

[3] Siemens Microcontrollers Inc., "*Controller Area Network*", October 1998.

[4] Motorola Inc., "*CAN Technical Overview*", January 2007.

[5] S. Ball, "*Analog Interfacing to Embedded Microprocessor Systems, Second Edition, Embedded Technology Series*", Newnes, 2004.

[6] D. Paret, et al, "*The I^2C Bus from Theory to Practice*", John Wiley & Sons, 1997.

[7] O. Pfeiffer, et al, "*Embedded Networking with CAN and CANopen*", Annabooks/Rtc Books, 2003.

[8] M. Jamali, "*A Comparative Study of Physical Layers of In-Vehicle Multiplexing Systems*", Society of Automotive Engineers Technical Paper Series 1999-01-1271, 1999.

[9] B. Negley, "*Getting Control Through CAN*", http://www.sensorsmag.com/articles/1000/18/main.shtml, October 2001.

[10] P. Richards, Microchip Technologies Inc., "*A CAN Physical Layer Discussion*", September 2001.

[11] W. Stallings, "*Data and Computer Communications, Seventh Edition*", Prentice Hall, 2004.

[12] K. Dietmayer, et al, "*CAN Bit Timing Requirements*", Society of Automotive Engineers Technical Paper Series 970295, 1997.

[13] T. Floyd, "*Digital Fundamentals - 3rd Edition*", Merrill, 1986.

[14] P. Richards, Microchip Technologies Inc., "*Understanding Microchip's CAN Module Bit Timing*", 2001.

[15] CIA Inc., "*CAN Data Link Layer*", 2006.

[16] K. Etschberger, "*Controller Area Network*", IXXAT Automation GmbH, 2001.

[17] National Instruments Inc., "*Anatomy of a Video Signal*", http://zone.ni.com/devzone/cda/tut/p/id/3020, 2004.

[18] D. Katz, et al, "*Embedded Media Processing*", Newnes, 2006.

[19] D. Vrhovnik, et al, "*The Basics of Interlaced Video and the Techniques used in De-Interlacing*", Digital TV Design Line - http://www.techonline.com, February 2007.

[20] K. Jack, "*Video Demystified - 4th Edition*", Newnes, 2004.

[21] G. Kennedy, "*Electronic Communication Systems - 4th Edition*", Lake Forest, 1993.

[22] B. Furht, et al, "*Video and Image Processing in Multimedia Systems*", Kluwer, 1995.

[23] Digital Creation Labs Inc., "*Digital Video Overview - AN10*", April 2004.

[24] University of California, “*Checkpoint 2 - Video Encoder*”, 2004.

[25] C. Poynton, “*Digital Video and HDTV: Algorithms and Interfaces*”, Kaufmann, 2003.

[26] International Telecommunications Union, http://www.itu.int/home/index.html, 2005.

[27] Rohde & Schwarz Broadcast Division Inc., “*The Digital Video Standard according to ITU-R BT. 601/656*”, 2006.

[28] International Telecommunications Union, “*Recommendation ITU-R BT.656-4 - Interfaces for Digital Component Video Signals in 525-Line & 625-Line Television Systems Operating at the 4:2:2 Level of Recommendation ITU-R BT.601 (Part A)*”, 1998.

[29] Intersil Inc., “*BT.656 Video Interface for ICs*”, July 2002.

[30] Berkley Design Technologies Inc., http://www.bdti.com, 2005.

[31] Freescale Semiconductor Inc., “*MPC5200 Data Sheet - Rev4.0*”, January 2005.

[32] Freescale Semiconductor Inc., “*MPC5200 Users Guide - 3.1*”, March 2006.

[33] Freescale Semiconductor Inc., “*MPC5200 Microprocessor Technical Summary*”, August 2004.

[34] Freescale Semiconductor Inc., “*Product Brief - Lite5200 EVB Kits*”, March 2004.

[35] Infineon Inc., “*TriCore TC1796 32-Bit Single-Chip Microcontroller Data Sheet - V0.7*”, March 2006.

[36] Infineon Inc., "*TriBoard TC1796 Hardware Manual TC1796.300 - V3.1*", January 2005.

[37] Xilinx Inc., "*Spartan-3E FPGA Family: Complete Data Sheet*", November 2006.

[38] Xilinx Inc., "*Spartan-3E Starter Kit Board User Guide - V1.0*", March 2006.

[39] Microchip Inc., "*dsPIC30F6014A Data Sheet*", 2006.

[40] Microchip Inc., "*dsPICDEM 1.1 Plus Development Board User's Guide*", 2006.

[41] Analog Devices Inc., "*Blackfin Embedded Processor ADSP-BF536/7 Preliminary Technical Data*", 2005.

[42] Analog Devices Inc., "*ADSP-BF537 Blackfin Processor Hardware Reference - Rev2.0*", December 2005.

[43] H. Minorikawa, et al, "*Current Status and Future Trends of Electronic Packaging in Automotive Applications*", Society of Automotive Engineers Technical Paper Series 901134, 1990.

[44] K. Skahill, "*VHDL for Programmable Logic*", Addison-Wesley, 1996.

[45] C. Szydlowski, "*Tradeoffs between Stand-Alone and Integrated CAN Peripherals*", Society of Automotive Engineers Technical Paper Series 941655, 1994.

[46] J. Bacon, "*The Motorola MC68000: An Introduction to Processor, Memory and Interfacing*", Prentice Hall, 1986.

[47] Altium Inc., "*TriCore Software Development Toolset - v2.2*", 2005.

[48] Green Hills Software Inc., "*TriCore Family*", 2002.

[49] HighTec GNU C/C++ Compiler, http://www.hightec-rt.com/index.php?option=com_content&task=view&id=16&Itemid=28, 2006.

[50] V. Pedroni, "*Circuit Design with VHDL*", Massachusetts Institute of Technology, 2004.

[51] μCLinux Embedded Linux/Microprocessor Project, http://www.uclinux.org, 2005.

[52] Green Hills Software Inc., "*Blackfin Processor Family Embedded Software Solutions*", 2005.

[53] Analog Devices Inc., "*LabVIEW Embedded Module for Analog Devices Blackfin Processors*", 2006.

[54] Blackfin.org - The Processor Forum, http://www.blackfin.org, 2005.

[55] DSPRelated.com, http://www.dsprelated.com, 2005.

[56] Microchip Inc., http://www.microchip.com, 2005.

[57] Microchip Inc., "*PIC18F2480/2580/4480/4580 Data Sheet*", 2004.

[58] Microchip Inc., "*PIC16F87XA Data Sheet*", 2003.

[59] Microchip Inc., "*MCP2510/15 Data Sheet*", 2002.

[60] Microchip Inc., "*MCP2551 - High Speed CAN Transceiver Data Sheet*", 2003.

[61] Mikroelektronika Inc., http://www.mikroelektronika.co.yu, 2005.

[62] Mikroelektronika Inc., MikroC Help Files, 2005.

[63] Kvaser Inc., http://www.kvaser.com, 2005.

[64] Microchip Inc., "*MCP2515 Development User's Guide*", 2003.

[65] Scientific Software Tools Inc., http://www.driverlinx.com/DownLoad/DlPortIO.htm, 2005.

[66] Philips Inc., "*TJA1041 High speed CAN Transceiver Data Sheet*", 2003.

[67] Analog Devices Inc., "*ADSP-BF537 EZ Kit Lite Evaluation Manual*", August 2005.

[68] Analog Devices Inc., "*VisualDSP++4.5 Device Drivers and System Services Manual for Blackfin Processors*", August 2006.

[69] Analog Devices Inc., "*Blackfin Online Learning and Development (BOLD) Video Tutorials*", http://www.demosondemand.com/clients/analogdevices/001/page/index.asp?ref=DSPS052, 2006.

[70] Analog Devices Inc., "*ADV7174/79 Chip Scale PAL/NTSC Video Encoder with Advanced Power Management - Data Sheet*", 2004.

[71] Analog Devices Inc., VisualDSP++ 4.5 Help Files, 2006.

[72] Analog Devices Inc., "*Blackfin A-V EZ-Extender Manual*", January 2005.

[73] H. Kosaka, et al, "*Evaluation of a New In-Vehicle HMI System Composed of Steering wheel Switch and Head-Up Display*", Society of Automotive Engineers Technical Paper Series 2006-01-0576, 2006.

[74] P. Desroches, et al, "*The Impact of Navigation Systems on the Perception Time of Young and Older Drivers*", Society of Automotive Engineers Technical Paper Series 2006-01-0577, 2006.

Appendix A - CAN Circuit Schematics

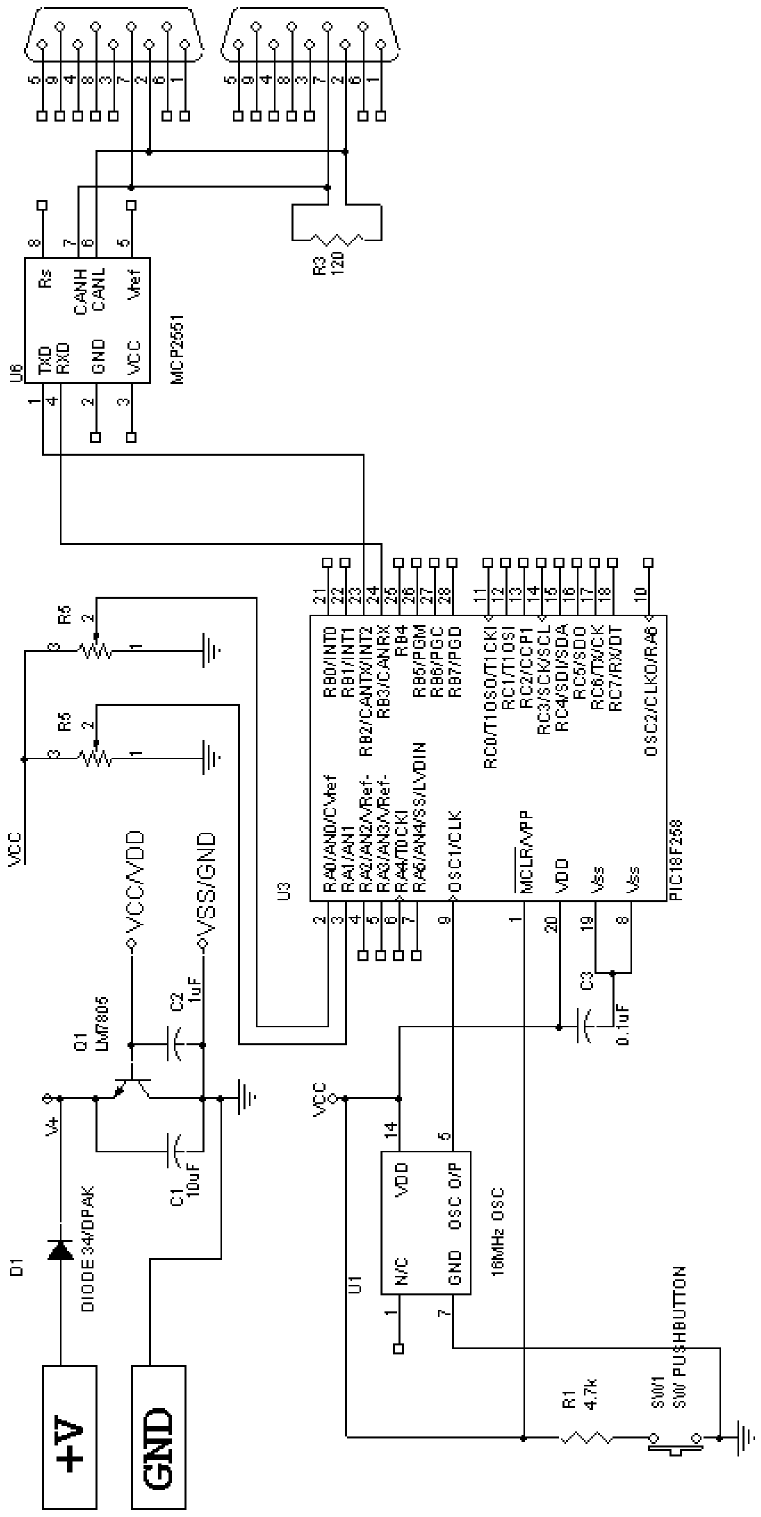


Figure 71: CAN On-Board Circuit Schematic

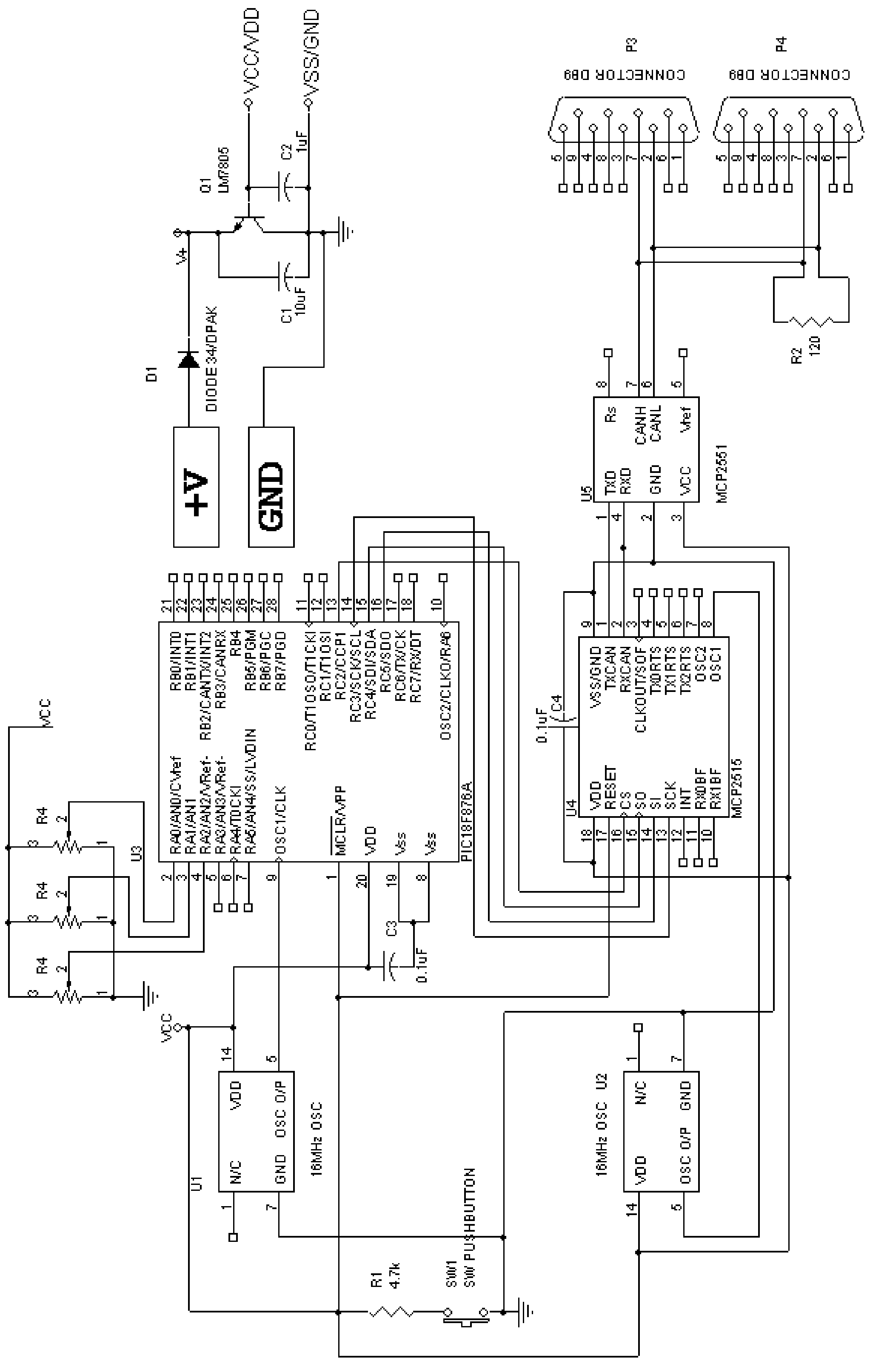


Figure 72: CAN SPI Circuit Schematic

Appendix B - CAN On-Board Source Code

```
/*****************************************************************************
*
* Device: PIC Microcontroller P18F258
* Osc: 16MHz
* File Name: "CAN_On_Board.c"
* Author: Dominick O' Brien
* Date: 29-Mar-06
* Version 1.00
*
*****************************************************************************/

/*****************************************************************************
*
* Type Declarations
*
*****************************************************************************/
typedef unsigned char uchar;
typedef long l_ID;
typedef unsigned int iadc;

/*****************************************************************************
*
* Variable Declarations
*
*****************************************************************************/
uchar aa = 0;
uchar aa1 = 0;
uchar len = 0; // CAN DLC
uchar data[8]; // CAN Data Bytes
l_ID id = 0; // CAN ID

iadc ch0_res = 0; // ADC Channel 0 result variable
uchar ms_ch0_res = 0; // ADC Channel 0 MSB result variable
uchar ls_ch0_res = 0; // ADC Channel 0 LSB result variable

iadc ch1_res = 0; // ADC Channel 1 result variable
uchar ms_ch1_res = 0; // ADC Channel 1 MSB result variable
uchar ls_ch1_res = 0; // ADC Channel 1 LSB result variable

void main()
{
  TRISC.f2 = 0;
  PORTC.f2 = 0;
  PORTC.f0 = 1; // Chip Select line of MCP2510
  TRISC.f0 = 0; // Make Port C Pin 0 an Output
  ADCON1 = 0x00; // Configure ALL analog inputs, Result RIGHT justified & Fosc/2
  TRISA  = 0xFF; // PORTA all inputs

  aa = CAN_CONFIG_SAMPLE_THRICE &  // form value to be used with CANInitialize()
       CAN_CONFIG_PHSEG2_PRG_ON &
       CAN_CONFIG_ALL_MSG &
```

```
            CAN_CONFIG_DBL_BUFFER_ON &
            CAN_CONFIG_LINE_FILTER_OFF;

        aa1 =  CAN_TX_PRIORITY_0 & // form value to be used with CANWrite()
                CAN_TX_STD_FRAME &
                CAN_TX_NO_RTR_FRAME;

        PORTC.f2 = 1;
        PORTC.f0 = 1; // CS line of MCP2510 HIGH

        CANSetOperationMode(CAN_MODE_CONFIG,0xFF); // Set CONFIGURATION mode
        // 16MHz, BRP = 16 => 62.5kbits/sec
        // 16MHz, BRP = 8 => 125kbits/sec
        // 16MHz, BRP = 4 => 250kbits/sec
        // 16MHz, BRP = 2 => 500kbits/sec
        // 8MHz, BRP = 8 => 62.5kbits/sec
        // 8MHz, BRP = 4 => 125kbits/sec
        // 8MHz, BRP = 2 => 250kbits/sec
        // 8MHz, BRP = 1 => 500kbits/sec
        CANInitialize( 2,2,3,3,1,aa); // Initialise CAN module. BAUD = 500kbits/sec
        CANSetOperationMode(CAN_MODE_NORMAL,0); // Set NORMAL mode

        while (1)
         {
         /**************************************************************************
         *
         * Remember the PICs ADC result is a 10 bit number
         * therefore we need two bytes to hold the 10 bit result
         *
         /**************************************************************************
           ch0_res = Adc_Read(0); // Get the ADC conversion result
           ls_ch0_res = ch0_res; // Get bottom 8 bits of ADC Channel 0 conversion
           ms_ch0_res = ch0_res >> 8; // Get top 2 bits of ADC Channel 0 conversion

           ch1_res = Adc_Read(1); // Get the ADC conversion result
           ls_ch1_res = ch1_res; // Get bottom 8 bits of ADC Channel 1 conversion
           ms_ch1_res = ch1_res >> 8; // Get top 2 bits of ADC Channel 1 conversion

           data[0] = ms_ch1_res; // 2 MSBs of Channel 1 conversion result
           data[1] = ls_ch1_res; // 8 LSBs of Channel 1 conversion result
           data[2] = ms_ch0_res; // 2 MSBs of Channel 0 conversion result
           data[3] = ls_ch0_res; // 8 LSBs of Channel 0 conversion result
           data[4] = 44; // Arbitrary Number
           data[5] = 55; // Arbitrary Number
           data[6] = 66; // Arbitrary Number
           data[7] = 77; // Arbitrary Number

           id = 0x411; // Message ID (Decimal 1041)
           len = 8; // Data Length Code
           CANWrite(id,data,len,aa1); // Write CAN message
           delay_ms(500); // Delay 500 milliseconds
         }

      } // EOF
```

Appendix C - CAN SPI Source Code

```
/*****************************************************************************
*
* Device: PIC Microcontroller P16F876A
* Osc: 16MHz
* File Name: "CAN_SPI.c"
* Author: Dominick O' Brien
* Date: 06-Dec-05
* Version 1.00
*
*****************************************************************************/

/*****************************************************************************
*
* Type Declarations
*
*****************************************************************************/
typedef unsigned char uchar;
typedef long l_ID;
typedef unsigned int iadc;

/*****************************************************************************
*
* Variable Declarations
*
*****************************************************************************/
uchar aa = 0;
uchar aa1 = 0;
uchar len = 0; // CAN DLC
uchar data[8]; // CAN Data Bytes
l_ID id = 0; // CAN ID

iadc ch0_res = 0; // ADC Channel 0 result variable
uchar ms_ch0_res = 0; // ADC Channel 0 MSB result variable
uchar ls_ch0_res = 0; // ADC Channel 0 LSB result variable

iadc ch1_res = 0; // ADC Channel 1 result variable
uchar ms_ch1_res = 0; // ADC Channel 1 MSB result variable
uchar ls_ch1_res = 0; // ADC Channel 1 LSB result variable

iadc ch2_res = 0; // ADC Channel 2 result variable
uchar ms_ch2_res = 0; // ADC Channel 2 MSB result variable
iadc ls_ch2_res = 0; // ADC Channel 2 LSB result variable

void main()
{
  Spi_Init(); // Initialise SPI

  TRISC.f2 = 0;
  PORTC.f2 = 0;
  PORTC.f0 = 1; // CS line of MCP2510
  TRISC.f0 = 0; // Make Port C Pin 0 an Output
```

```
        ADCON1 = 0x80; // Configure ALL analog inputs, Fosc/2 & Result RIGHT justified
        TRISA  = 0xFF; // PORTA all inputs

        aa = CAN_CONFIG_SAMPLE_THRICE &  // form value to be used with CANSPIInitialize()
             CAN_CONFIG_PHSEG2_PRG_ON &
             CAN_CONFIG_ALL_MSG &
             CAN_CONFIG_DBL_BUFFER_ON &
             CAN_CONFIG_LINE_FILTER_OFF;

        aa1 =  CAN_TX_PRIORITY_0 & // form value to be used with CANSPIWrite()
               CAN_TX_STD_FRAME &
               CAN_TX_NO_RTR_FRAME;

        PORTC.f2 = 1;
        PORTC.f0 = 1; // CS line of MCP2510 HIGH

        CANSPISetOperationMode(CAN_MODE_CONFIG,0xFF); // Set CONFIGURATION mode
        // 16MHz, BRP = 16 => 62.5kbits/sec
        // 16MHz, BRP = 8 => 125kbits/sec
        // 16MHz, BRP = 4 => 250kbits/sec
        // 16MHz, BRP = 2 => 500kbits/sec
        // 8MHz, BRP = 8 => 62.5kbits/sec
        // 8MHz, BRP = 4 => 125kbits/sec
        // 8MHz, BRP = 2 => 250kbits/sec
        // 8MHz, BRP = 1 => 500kbits/sec
        CANSPIInitialize( 2,2,3,3,1,aa); // Initialise external CAN module. BAUD = 500kbits/sec
        CANSPISetOperationMode(CAN_MODE_NORMAL,0); // Set NORMAL mode

        while (1)
        {
         /***************************************************************************
         *
         * Remember the PICs ADC result is a 10 bit number
         * therefore we need two bytes to hold the 10 bit result
         *
         ***************************************************************************/
          ch0_res = Adc_Read(0); // Get the ADC conversion result
          ls_ch0_res = ch0_res; // Get bottom 8 bits of ADC Channel 0 conversion
          ms_ch0_res = ch0_res >> 8; // Get top 2 bits of ADC Channel 0 conversion

          ch1_res = Adc_Read(1); // Get the ADC conversion result
          ls_ch1_res = ch1_res; // Get bottom 8 bits of ADC Channel 1 conversion
          ms_ch1_res = ch1_res >> 8; // Get top 2 bits of ADC Channel 1 conversion

          ch2_res = Adc_Read(2); // Get the ADC conversion result
          ls_ch2_res = ch2_res; // Get bottom 8 bits of ADC Channel 2 conversion
          ms_ch2_res = ch2_res >> 8; // Get top 2 bits of ADC Channel 2 conversion

          data[0] = ms_ch2_res; // 2 MSBs of Channel 2 conversion result
          data[1] = ls_ch2_res; // 8 LSBs of Channel 2 conversion result
          data[2] = ms_ch1_res; // 2 MSBs of Channel 1 conversion result
          data[3] = ls_ch1_res; // 8 LSBs of Channel 1 conversion result
          data[4] = ms_ch0_res; // 2 MSBs of Channel 0 conversion result
          data[5] = ls_ch0_res; // 8 LSBs of Channel 0 conversion result
          data[6] = 22; // Arbitrary Number
          data[7] = 33; // Arbitrary Number
```

```

    id = 0x189; // Message ID (Decimal 393)
    len = 8; // Data Length Code
    CANSPIWrite(id,data,len,aa1); // Write CAN message
    delay_ms(500); // Delay 500 milliseconds
   }

} // EOF
```

Appendix D - Blackfin CAN Module Source Code

CAN_Test.h

```
/*****************************************************************************
*
* Device: ADSP-BF537
* Osc: SCLK = 120MHz
* File Name: "CAN_Test.h"
* Author: Dominick O' Brien
* Date: 10-May-06
* Version 1.00
*
*****************************************************************************/
#ifndef   _CAN_RX_H
#define _CAN_RX_H

#include <cdefBF537.h>
#include <ccblkfn.h>
#include <sys/exception.h>

/*****************************************************************************
*
* Constants
*
*****************************************************************************/
#define CAN_TX_MB_LO          0x0000
#define CAN_TX_MB_HI          0x0100  // Mailbox24
#define CAN_RX_MB_LO          0x00C0  // Mailbox 7 and Mailbox 6
#define CAN_RX_MB_HI          0x0000

/*****************************************************************************
*
* Global Data
*
*****************************************************************************/
extern char blink, off, change;
extern volatile unsigned int delay;
extern short display;
extern volatile unsigned short * CAN_MB_ID1[];
extern volatile unsigned short * CAN_MB_ID0[];
extern volatile unsigned short * CAN_MB_TIMESTAMP[];
extern volatile unsigned short * CAN_MB_LENGTH[];
extern volatile unsigned short * CAN_MB_DATA3[];
extern volatile unsigned short * CAN_MB_DATA2[];
extern volatile unsigned short * CAN_MB_DATA1[];
extern volatile unsigned short * CAN_MB_DATA0[];

/*****************************************************************************
*
* Function Prototypes
*
```

```
*******************************************************************************/
// In Initialization.c
void Init_PLL(void);
void Init_Port(void);
void Init_CAN_Timing(void);
void Init_CAN_Mailboxes(void);
void Init_Interrupts(void);

// In CAN_Functions.c
void CAN_Enable(void);
void CAN_Transmit(void);
void CAN_Setup_Interrupts(void);

// In Interrupts.c
EX_INTERRUPT_HANDLER(CAN_RCV_HANDLER);
EX_INTERRUPT_HANDLER(CAN_XMT_HANDLER);

#endif    // _CAN_RX_H
```

Initialization.c.

```
/*******************************************************************************
*
* Device: ADSP-BF537
* Osc: SCLK = 120MHz
* File Name: "Initialization.c"
* Author: Dominick O' Brien
* Date: 12-May-06
* Version 1.00
*
*******************************************************************************/
#include "CAN_Test.h"

/*******************************************************************************
*
* Init_PLL – Configures the PLL so that the CAN BRP can easily be derived. Sets the CCLK to
*                600MHz and SCLK to 120MHz
*
*******************************************************************************/
void Init_PLL()
{
   *pPLL_CTL = SET_MSEL(24); // Set PLL: (25MHz X 24 (MSEL = 24)): CCLK=600MHz
   idle();
   *pPLL_DIV = SET_SSEL(4); // Set SCLK Divisor: (600MHz / (SSEL=5)): SCLK=120MHz
   ssync();
} // End Init_PLL

/*******************************************************************************
*
* Init_Port – Sets up the Ports for CAN use and configured the PFx pins for access to the
*                on-board LEDs.
*
*******************************************************************************/
void Init_Port ()
{
  short temp_fix;
  // Configure CAN RX and CAN TX pins on GPIO Port
   temp_fix = *pPORT_MUX;
   ssync();

   *pPORT_MUX = PJCE_CAN; // Enable CAN Pins On Port J
   ssync();
   *pPORT_MUX = PJCE_CAN; // #22 work-around: write it a few times
   ssync();
   *pPORT_MUX = PJCE_CAN; // #22 work-around: write it a few times
   ssync();

   temp_fix = *pPORT_MUX; // #22 work-around: read PORT_MUX after writing
   ssync();

   // Configure Port F pins for LED access
   *pPORTFIO_DIR   = 0x0FC0; // Enable PF6-11 As Outputs (LEDs)
   ssync();
} // End Init_Port ()

/******************************************************************************
*
* Init_CAN_Timing – Sets up the CAN_TIMING & CLOCK Registers
*
******************************************************************************/
void Init_CAN_Timing()
{
  //        ====================================================
  //        BIT TIMING:
  //
  //        CCLK 600 MHz
  //        SCLK 120 MHz
  //
  //        CAN_CLOCK  : Prescaler (BRP)
  //        CAN_TIMING : SJW = 2, TSEG2 = 3, TSEG1 = 5
  //
  //        tBIT = TQ x (1 + (TSEG1 + 1) + (TSEG2 + 1))
  //        2e-6 = TQ x (1 + (5 + 1) + (3 + 1))
  //        TQ = 1.82e-7
  //
  //        TQ = (BRP+1) / SCLK
  //        1.82e-7 = (BRP+1) / 120e6
  //        (BRP+1) = 21.84
  //        BRP = 20.84 ~ 21
  //        ====================================================
  //        Set Bit Configuration Registers ...
  //        ====================================================
  *pCAN_TIMING = 0x0235;
   *pCAN_CLOCK  = 21; // [0x15] 500kHz CAN Clock :: tBIT = 2us
  ssync();
} // End Init_CAN_Timing()

/******************************************************************************
*
* Init_CAN_Mailboxes – Configures Mailbox 24 to transmit a specific message ID with a
*                           message length of 8 bytes. Configures Mailbox 6 and 7 to each receive
*                               a specific message ID
*
******************************************************************************/
void Init_CAN_Mailboxes()
{
  short msgID;                  // Variable for Mailbox 24 ID
  short msgID_OnB;          // Variable for Mailbox 7 ID
  short msgID_SPI;           // Variable for Mailbox 6 ID

  volatile char mbID;
  volatile char mbID_OnB; // Variable for Mailbox # for ON_B
  volatile char mbID_SPI; // Variable for Mailbox # for SPI
  // Mailbox 24 Will Transmit ACK  to the Network via ID 0x007
  msgID = 0x007;
  mbID  = 24;

  *(pCAN_MB_ID1(mbID)) = msgID << 2; // ID1, mask disabled, remote frame disable, 11 bit
                                                          // identifier
  *(pCAN_MB_ID0(mbID)) = 0; // ID0 = all 0's
  *(pCAN_MB_LENGTH(mbID)) = 8; // DLC = 8 bytes

  // Mailbox 7 will Receive CAN Command from Network via ID 0x411
  // Mailbox 6 will Recieve CAN Command from Network via ID 0x189
  msgID_OnB = 0x411; // ID = dec 1041
  msgID_SPI = 0x189; // ID = dec 393
  mbID_OnB  = 7; // Mailbox 7
  mbID_SPI = 6; // Mailbox 6

  *(pCAN_MB_ID1(mbID_OnB)) = msgID_OnB << 2; // ID1, mask disabled, remote frame
                                                                   // disable, 11 bit identifier
  *(pCAN_MB_ID0(mbID_OnB)) = 0; // ID0 = all 0's
  *(pCAN_MB_ID1(mbID_SPI)) = msgID_SPI << 2; // ID1, mask disabled, remote frame
                                                                // disable, 11 bit identifier
  *(pCAN_MB_ID0(mbID_SPI)) = 0; // ID0 = all 0's
  *(pCAN_MB_LENGTH(mbID_SPI)) = 8; // DLC = 8 bytes
} // End Init_CAN_Mailboxes()

/******************************************************************************
*
* Init_Interrupts – Assigns interrupt priorities for CAN TX and CAN RX.
*
******************************************************************************/
void Init_Interrupts()
{
  // Configure Interrupt Priorities
  *pSIC_IAR0 = 0x77777777;
  *pSIC_IAR1 = 0x07777777; // CAN RX IRQ   : 0=IVG7
  *pSIC_IAR2 = 0x77777771; // CAN TX IRQ   : 1=IVG8
  *pSIC_IAR3 = 0x77777777;

// Register Interrupt Handlers and Enable Core Interrupts
  register_handler(ik_ivg7, CAN_RCV_HANDLER);
  register_handler(ik_ivg8, CAN_XMT_HANDLER);
  // Enable SIC Level Interrupts
  *pSIC_IMASK |= (IRQ_CAN_RX|IRQ_CAN_TX);
  } // End Init_Interrupts
```

CAN_Functions.c.

```
/*******************************************************************************
*
* Device: ADSP-BF537
* Osc: SCLK = 120MHz
* File Name: "CAN_Functions.c"
* Author: Dominick O' Brien
* Date: 12-May-06
* Version 1.00
*
*******************************************************************************/
#include "CAN_Test.h"

/*******************************************************************************
*
* CAN_Setup_Interrupts – Enables Mailbox Interrupts for Mailboxes Used
*
*******************************************************************************/
void CAN_Setup_Interrupts()
{
 *pCAN_MBIM1 = 0x00C0; // Enable Interrupts for Mailbox 7 and Mailbox 6
 *pCAN_MBIM2 = 0x0100; // Enable Interrupt for Mailbox 24
 ssync();
} // End CAN_Setup_Interrupts

/*******************************************************************************
*
* CAN_Enable – Writes Mailbox Direction and Enables Registers before issuing a CAN
*                    Configuration Request and waiting for a CAN Configuration acknowledge
*                    before continuing.
*
*******************************************************************************/
void CAN_Enable()
{
 // Set Mailbox Direction
 *pCAN_MD1 = CAN_RX_MB_LO; // No Low Mailboxes (MB 0-15) Are RX
 *pCAN_MD2 = CAN_TX_MB_LO; // Mailbox 24 Enabled For TX

 // Enable Mailboxes
 *pCAN_MC1 = CAN_RX_MB_LO; // Enables Mailbox 7 and Mailbox 6
 *pCAN_MC2 = CAN_TX_MB_HI; // Enables Mailbox 24
 ssync();

 *pCAN_CONTROL &= ~CCR; // Enable CAN Configuration Mode (Clear CCR)

while(*pCAN_STATUS & CCA); // Wait for CAN Configuration Acknowledge (CCA)

} // End CAN_Enable
```

Interrupts.c.

```
/*******************************************************************************
*
* Device: ADSP-BF537
* Osc: SCLK = 120MHz
* File Name: "Interrupts.c"
* Author: Dominick O' Brien
* Date: 15-May-06
* Version 1.00
*
******************************************************************************/
#include "CAN_Test.h"

/******************************************************************************
*
* CAN_RCV_HANDLER – This ISR checks for the highest priority RX Mailbox with an
*                   active interrupt and clears it.
*                   If the IRQ is from MB7, the appropriate operating flags are set
*                   based on the current mode and the contents of MB7.
*                   If the IRQ is from MB6, the received data in MB6 is transferred to
*                   MB24 and a request to transmit this data is made.
*
******************************************************************************/
EX_INTERRUPT_HANDLER(CAN_RCV_HANDLER)
{
  char  highMB; // Which CAN Registers Should Be Used (1 or 2)
  // short data type is 16 bits
  short mbim_status; // Temp Location for Interrupt Status
  short bit_pos = 0; // Offset Into MBxIF Registers

  mbim_status = *pCAN_MBRIF2;

  if (mbim_status == 0) // If High 16 MBoxes Have No Active IRQ
  {
    mbim_status = *pCAN_MBRIF1; // Check Low 16 MBoxes
    highMB = 0; // Clear High/Low* Indicator
  }

  else // Otherwise, Active High MBox IRQ Found
  {
    highMB = 1; // Set High/Low* Indicator
  }

  while (!(mbim_status & 0x8000)) // Scan Status Register For Highest MB IRQ
  {
    mbim_status <<= 1;
    bit_pos++; // bit_pos Contains Offset from MB31
  }

  if (highMB)
  {
    *pCAN_MBRIF2 = (1 << (15 - bit_pos));
  }

else // Low Mailbox Interrupt
  {
    if(bit_pos == 0x8) // if Mailbox7 IRQ
    {
      if((*(pCAN_MB_DATA3(7)) == 0) || (*(pCAN_MB_DATA3(7)) <= 512))
        {
            if(blink) // if blinking already
            {
              change = 0; // no mode change
            }

            else // otherwise it was off
            {
              change = 1; // set mode change
               off = 0; // make sure OFF is cleared
               blink = 1; // set BLINK flag
              display = 0x0FC0; // display all LEDs on
            }

        if(*(pCAN_MB_DATA3(7)) >= 513)
        {
          if (!off) // if not in OFF mode
          {
            off = 1; // set OFF flag
            blink = 0; // clear BLINK flag
          } // End if off
        }
      } // End if Mailbox 7

      if(bit_pos == 0x9)        // if Mailbox 6 IRQ
      {
        // Place Received Commands Into CAN TX Mailbox
        *(pCAN_MB_DATA3(24)) = *(pCAN_MB_DATA3(6));
        *(pCAN_MB_DATA2(24)) = *(pCAN_MB_DATA2(6));
        *(pCAN_MB_DATA1(24)) = *(pCAN_MB_DATA1(6));
        *(pCAN_MB_DATA0(24)) = *(pCAN_MB_DATA0(6));

        // Issue CAN Transmit Request for Mailbox 24
        *pCAN_TRS2 = CAN_TX_MB_HI;
        ssync();
      } // End if Mailbox 6

      *pCAN_MBRIF1 = (1 << (15 - bit_pos));            // Write-1-to-Clear RX IRQ
    } // End Low Mailbox Interrupt

    ssync();

  } // End CAN_RCV_HANDLER

  /*****************************************************************************
  *
  * CAN_XMT_HANDLER – This ISR checks for the highest priority TX Mailbox with an
  *                              active interrupt and clears it.
  *
  *****************************************************************************/
  EX_INTERRUPT_HANDLER(CAN_XMT_HANDLER)
  {
    char  highMB; // Which CAN Registers Should Be Used (1 or 2)
    short mbim_status; // Temp Location for Interrupt Status
    short bit_pos = 0;            // Offset Into MBxIF Registers

    mbim_status = *pCAN_MBTIF2; // Check High Mailboxes First
    if (mbim_status == 0) // If No High MB Interrupts
    {
      mbim_status = *pCAN_MBTIF1; // Check Low MB Interrupts
      highMB = 0; // Clear High/Low* Mailbox Indicator
    }

    else highMB = 1; // Set High/Low* Mailbox Indicator

    while (!(mbim_status & 0x8000)) // Find Highest Mailbox W/ Active IRQ
    {
      mbim_status <<= 1;
      bit_pos++;
    } // Interrupting Mailbox Found

    if (highMB) // Process High Mailbox IRQ
    {
      *pCAN_MBTIF2 = (1 << (15 - bit_pos));
    }

    else // Else, Process Low Mailbox IRQ
    {
      *pCAN_MBTIF1 = (1 << (15 - bit_pos));
    }
  ssync();

  } // End CAN_XMT_HANDLER
```

main.c.

```
/*******************************************************************************
*
* Device: ADSP-BF537
* Osc: SCLK = 120MHz
* File Name: "Interrupts.c"
* Author: Dominick O' Brien
* Date: 18-May-06
* Version 1.00
*
*******************************************************************************/
#include "CAN_Test.h"

/*******************************************************************************
*
* Global Data
*
*******************************************************************************/
char  blink = 0; // Display Select  (1=blink,   0=scroll)
char  change = 0; // Change Display Flag (1=Changed, 0=Same)
char  off = 0; // Clear Display (1=Clear,   0=Not)
short display; // LED Display Value
volatile unsigned int delay = 0x400000;

main()
{
  Init_PLL(); // Set PLL
```

```
 Init_Port(); // Initialize Ports
 Init_Interrupts(); Initialize Interrupts
 Init_CAN_Timing(); // Setup CAN Timing
 Init_CAN_Mailboxes(); // Initialize CAN Mailbox Area
 CAN_Setup_Interrupts(); // Configure CAN Mailbox Interrupts
 CAN_Enable(); // Enable CAN

 display = 0; // All LEDs off

 while(1) // wait for IRQs
 {
   *pPORTFIO = display;  // write display

    while(delay--); // wait

    delay = 0x400000; // reset delay

    if (off) // if OFF flag is set
    {
      display = 0x0000; // turn LEDs off
    }

    else if (blink) // else if blink flag is set
    {
      display = ~display; // toggle display
    }

 } // End while forever

} // End main
```

Appendix E - Blackfin Video Implementation Source Code

ezkitutilities.h

```
/*****************************************************************************
*
* Device: ADSP-BF537
* Osc: SCLK = 120MHz
* File Name: "ezkitutilities.h"
* Author: Dominick O' Brien
* Date: 13-Nov-06
* Version 1.00
* Modified version of Analog Device's "ezkitutilities.h" found in VisualDSP++4.5
* References: "..\Blackfin\Examples\ADSP-BF533 EZ-Kit Lite\Drivers\PPI\Streaming.."
*
*****************************************************************************/
#ifndef EZKITUTILITIES_H
#define EZKITUTILITIES_H

/*****************************************************************************
*
* Board Specific Info
*
*****************************************************************************/
#define EZ_NUM_LEDS (6) // Number of LEDs on the board

/*****************************************************************************
*
* LED Defines
*
*****************************************************************************/
#define EZ_FIRST_LED (0) // First LED
#define EZ_LAST_LED (EZ_NUM_LEDS - 1) // Last LED

ADI_FLAG_ID ezLEDToFlag[]; // Structure containing the pf mappings for flags

/*****************************************************************************
*
* Functions Provided by the Utilities
*
*****************************************************************************/
void ezInit (u32 NumCores); // Initialises power, ebiu, any async, flash etc.
void ezInitPower u32 NumCores); // Initialises Power
void ezInitLED (u32 Led); // Enables/configures an LED for use
void ezTurnOffLED (u32 Led); // Dims an LED
void ezCycleLEDs (void); // Cycles LEDs
void ezSetDisplay (u32 Display); // Sets the LED pattern
void ezDelay (u32 msec); // Delays for approximately 'n' milliseconds
void ezErrorCheck (u32 Result); // Lights LEDs and spins to indicate an error if Result != 0
void ezEnableVideoEncoder (void); // Enables the 7179 video encoder

#endif  // EZKITUTILITIES_H
```

adi_itu656.h

```
/******************************************************************************
*
* Device: ADSP-BF537
* Osc: SCLK = 120MHz
* File Name: "adi_656.h"
* Author: Dominick O' Brien
* Date: 13-Nov-06
* Version 1.00
* Modified version of Analog Device's "adi_itu656.h" found in VisualDSP++4.5
* References: "..\Blackfin\Examples\ADSP-BF533 EZ-Kit Lite\Drivers\PPI\Streaming.."
*
******************************************************************************/
#ifndef ADI_ITU656_H // Define adi_itu656.h
#define ADI_ITU656_H

/******************************************************************************
*
* Common Definitions
*
******************************************************************************/
#define ADI_ITU656_EAV_SIZE 4 // EAV size (bytes)
#define ADI_ITU656_SAV_SIZE 4 // SAV size (bytes)

/******************************************************************************
*
* NTSC Definitions: Resolution - 720x480, 525/60 Video System
*
******************************************************************************/
#define ADI_ITU656_NTSC_WIDTH (720) // NTSC Resolution
#define ADI_ITU656_NTSC_HEIGHT (525) // Including Active & Blank lines
#define ADI_ITU656_NTSC_ACTIVE_FLINES (240) // Active Field lines

// Active Lines in a Frame
#define ADI_ITU656_NTSC_ACTIVE_LINES (ADI_ITU656_NTSC_ACTIVE_FLINES * 2)
#define ADI_ITU656_NTSC_BLANKING (268) // Blanking Size for NTSC

// Total Line Width
#define ADI_ITU656_NTSC_LINE_WIDTH ((ADI_ITU656_NTSC_WIDTH * 2)    +  \
                        ADI_ITU656_NTSC_BLANKING       +  \
                        ADI_ITU656_EAV_SIZE            +  \
                        ADI_ITU656_SAV_SIZE)

// Interlaced NTSC Definitions
#define ADI_ITU656_NTSC_ILF1_START (23) // NTSC Interlaced Active Frame Field1 (odd)
                                        // Start Line
#define ADI_ITU656_NTSC_ILF1_END (262) // NTSC Interlaced Active Frame Field1 (odd)
                                       // Finish Line
#define ADI_ITU656_NTSC_ILF2_START (286) // NTSC Interlaced Active Frame Field2
                                         // start line
#define ADI_ITU656_NTSC_ILF2_END (525) // NTSC Interlaced active frame field2 (even)
                                       // (even) Start Line
// Progressive NTSC Definitions
#define ADI_ITU656_NTSC_PRF_START (46) // NTSC Progressive Active Frame Start Line
#define ADI_ITU656_NTSC_PRF_END (525) // NTSC Progressive Active Frame Finish Line
```

```
/*******************************************************************************
*
* PAL Definitions: 720x576, 625/50 Video System
*
*******************************************************************************/
#define ADI_ITU656_PAL_WIDTH (720) // PAL resolution
#define ADI_ITU656_PAL_HEIGHT (625) // Including Active & Blank Lines
#define ADI_ITU656_PAL_ACTIVE_FLINES (288) // Active Field Lines

// Active Lines in a Frame
#define ADI_ITU656_PAL_ACTIVE_LINES (ADI_ITU656_PAL_ACTIVE_FLINES * 2)
#define ADI_ITU656_PAL_BLANKING (280) // Blanking Size for PAL (Bytes)

// Total Line Width
#define ADI_ITU656_PAL_LINE_WIDTH ((ADI_ITU656_PAL_WIDTH * 2)    +  \
                                   ADI_ITU656_PAL_BLANKING       +  \
                                   ADI_ITU656_EAV_SIZE           +  \
                                   ADI_ITU656_SAV_SIZE)
// Interlaced PAL Definitions
#define ADI_ITU656_PAL_ILF1_START (23) // PAL Interlaced Active Frame Field1 (odd)
                                       // Start Line
#define ADI_ITU656_PAL_ILF1_END (310) // PAL Interlaced Active Frame Field1 (odd)
                                      // Finish Line
#define ADI_ITU656_PAL_ILF2_START (336) // PAL Interlaced Active Frame Field2 (even)
                                        // Start Line
#define ADI_ITU656_PAL_ILF2_END (623) // PAL Interlaced Active Frame Field2 (even)
                                      // Start Line
// Progressive PAL Definitions
#define ADI_ITU656_PAL_PRF_START (45) // PAL Progressive Active Frame Start Line
#define ADI_ITU656_PAL_PRF_END (620) // PAL Progressive Active Frame Finish Line

/*******************************************************************************
*
* Enumerations for Video Formats
*
*******************************************************************************/
typedef enum{ // Video Formats
  ADI_ITU656_NTSC_IL, // NTSC Interlaced Frame
  ADI_ITU656_PAL_IL, // PAL Interlaced Frame
  ADI_ITU656_NTSC_PR, // NTSC Progressive Frame
  ADI_ITU656_PAL_PR // PAL Progressive Frame
}ADI_ITU656_FRAME_TYPE;

/*******************************************************************************
*
* API Function Declarations
*
*******************************************************************************
void adi_itu656_FrameFormat ( // Formats an Area in Memory into a Video Frame
  u8 *frame_ptr, // Pointer to an Area of Memory used for Frame
  ADI_ITU656_FRAME_TYPE frametype // Memory will be Formatted for this Frame Type
);

void adi_itu656_FrameFill ( // Fills Active Video Portions of Formatted Frame to Specified
                            // Colour
  u8 *frame_ptr, // Pointer to a Formatted Video Frame in Memory
  ADI_ITU656_FRAME_TYPE frametype, // Formatted Memory Frame Type
  u8 *ycbcr_data // 4 byte Array of 32 bit Colour Value of YCbCr Data
);

void adi_itu656_RowFill ( // Fills a Row of Pixels in Active Video Portion of Formatted Frame
                                    // with Specified Colour
  u8 *frame_ptr, // Pointer to a Formatted video frame in memory
  ADI_ITU656_FRAME_TYPE frametype, // Formatted Memory Frame Type
  u32 row_value, // 32 Bit Value Corresponding to Row Number of Active Field
  u8 *ycbcr_data // 4 byte Array of 32 bit Colour Value of YCbCr Data
);

void adi_itu656_ColumnFill ( // Fills a Column of Pixels in Active Video Portion of Formatted
                                       // Frame with Specified Colour
  u8 *frame_ptr,    // Pointer to a formatted video frame in memory
  ADI_ITU656_FRAME_TYPE frametype, // Formatted Memory Frame type
  u32 column_value,  // 32 bit Value Corresponding to Column Number of Active Field
  u8 *ycbcr_data // 4 byte Array of 32 bit Colour Value of YCbCr Data
);

#endif // End itu656.h definition
```

ezkitutilities.c

```
/*****************************************************************************
*
* Device: ADSP-BF537
* Osc: SCLK = 120MHz
* File Name: "ezkitutilities.c"
* Author: Dominick O' Brien
* Date: 16-Nov-06
* Version 1.00
* Modified version of Analog Device's "ezkitutilities.c" found in VisualDSP++4.5
* References: "..\Blackfin\Examples\ADSP-BF533 EZ-Kit Lite\Drivers\PPI\Streaming.."
*
*****************************************************************************/
#include <services/services.h> // System Service Includes
#include <sysreg.h> // System Configuration Definitions
#include <defBF537.h> // Include all MMR's and bit definitions
#include "ezkitutilities.h" // EZ-Kit Utility Definitions

ADI_FLAG_ID ezLEDToFlag[] = {
  ADI_FLAG_PF6, // LED 0
  ADI_FLAG_PF7, // LED 1
  ADI_FLAG_PF8, // LED 2
  ADI_FLAG_PF9, // LED 3
  ADI_FLAG_PF10, // LED 4
  ADI_FLAG_PF11 // LED 5
};

/*****************************************************************************
*
* LED Control
*
```

```
******************************************************************************/
static u32 LEDDisplay; // Bit Field representing the LED display
static u32 LEDEnables; // Bit Field representing the Enabled LEDs

/******************************************************************************
*
* ezInitPower - Initialises and Sets Power management SDRAM parameters on the EZ-Kit.
*
******************************************************************************/
#define DO_NOT_CHANGE_MMR_SETTINGS 0
static void ezInitPower(u32 NumCores)
{
  ADI_EBIU_RESULT EBIUResult;
  ADI_PWR_RESULT  PWRResult;

  // It is important that the EBIU module is configured before Power module so that changes to
  // the clock frequencies are correctly reflected in the SDRAM settings.
  ADI_EBIU_COMMAND_PAIR ezkit_sdram[] = // Initialises the EBIU module
  {
    { ADI_EBIU_CMD_SET_EZKIT, (void*)ADI_EBIU_EZKIT_BF537 },
    { ADI_EBIU_CMD_END, 0}
  };

  EBIUResult = adi_ebiu_Init( ezkit_sdram, DO_NOT_CHANGE_MMR_SETTINGS );

  if ((EBIUResult != ADI_EBIU_RESULT_SUCCESS) && (EBIUResult !=
  ADI_EBIU_RESULT_CALL_IGNORED))
  {
    ezErrorCheck(EBIUResult);
  }

  ADI_PWR_COMMAND_PAIR ezkit_power[] = // Initialises the Power Management Module
  {
    { ADI_PWR_CMD_SET_EZKIT, (void*)ADI_PWR_EZKIT_BF537_600MHZ },
    { ADI_PWR_CMD_END, 0}
  };

  PWRResult = adi_pwr_Init(ezkit_power);

  if ((PWRResult != ADI_PWR_RESULT_SUCCESS) && (PWRResult !=
  ADI_PWR_RESULT_CALL_IGNORED))
  {
    ezErrorCheck(PWRResult);
  }

  ezErrorCheck( adi_pwr_SetFreq( 0, 0, ADI_PWR_DF_NONE ) );
  ezErrorCheck( adi_pwr_SetMaxFreqForVolt( ADI_PWR_VLEV_115 ) )
}

/******************************************************************************
*
* ezInit – Initialises the EZ Kit board. Specifically Configuring:
*                                                   - Async Memory
*                                                   - Flash
*                                                   - CCLK = 600MHz, SCLK = 120MHz
*
```

```
*****************************************************************************/
void ezInit(u32 NumCores)
{
  // Configure Async Memory
  *pEBIU_AMBCTL0  = 0x7bb07bb0; // Write Access Time = 7 Cycles, Read Access Time =
                                // 11Cycles, No ARDY
  *pEBIU_AMBCTL1  = 0x7bb07bb0; // Hold Time = 2 Cycles, Setup time = 3 Cycles,
                                // Transition time = 4 cycles
  *pEBIU_AMGCTL   = 0x00FF;

  // Configure Flash
  *pFlashA_PortA_Out = 0; // Resets Port A to Initial Value
  *pFlashA_PortA_Dir = 0xFF; // Configure Everything on Port A as Outputs
  *pFlashA_PortB_Out = 0; // Resets Port B to Initial Value
  *pFlashA_PortB_Dir = 0x3f; // Configure Everything on Port B as Outputs

  ezInitPower(NumCores); // Configure Power
}

/*****************************************************************************
*
* ezInitLEDs - Enables an LED for use
*
*****************************************************************************/
void ezInitLED(u32 LED) // Enables an LED
{
  if (LED >= EZ_NUM_LEDS)
  {
    return; // Make sure the LED is Valid
  }

  LEDEnables |= (1 << LED); // Set the Enable bit
  adi_flag_Open(ezLEDToFlag[LED]); // Configure the Flag for Output
  adi_flag_SetDirection(ezLEDToFlag[LED], ADI_FLAG_DIRECTION_OUTPUT);
  ezTurnOffLED(LED); // Dim the LED
}

/*****************************************************************************
*
* ezTurnOffLED - Dims an LED
*
*****************************************************************************/
void ezTurnOffLED(u32 LED)
{
  ezSetDisplay(LEDDisplay & ~(1 << LED)); // Update
}

/*****************************************************************************
*
* ezCycleLEDs - Cycles LEDs
*
*****************************************************************************/
void ezCycleLEDs(void)
{
  static u32 CycleDisplay;

```

```
 if (LEDEnables = = 0) // Insure at least 1 LED is Enabled
 {
   return;
 }

 do { // calculate the pattern
   CycleDisplay <<= 1;
   if (CycleDisplay = = 0)
   {
     CycleDisplay = 1;
   }
 } while ((CycleDisplay & LEDEnables) = = 0);

 ezSetDisplay(CycleDisplay); // Update
}

/*******************************************************************************
*
* ezSetDisplay - Sets the display pattern
*
*******************************************************************************/
void ezSetDisplay(u32 Display)
{
 u32 i;
 u32 Mask;

 LEDDisplay = Display & LEDEnables; // Update the Display

 for (i = 0, Mask = 1; i < EZ_NUM_LEDS; i++, Mask <<= 1) // FOR (each LED)
 {
   if (LEDDisplay & Mask) // IF (the LED should be lit)
   {
     adi_flag_Set(ezLEDToFlag[i]); // Light It
   }

   else if (LEDEnables & Mask)
   {
     adi_flag_Clear(ezLEDToFlag[i]); // Dim It
   } // end if
 } // end for

}

/*******************************************************************************
*
* ezDelay - Delays for approximately 1msec when running at 600 MHz
*
*******************************************************************************/
void ezDelay(u32 msec)
{
 volatile u32 i,j;

 for (j = 0; j < msec; j++) // value of 0x3000000 is about 1 sec so 0xc49b is about 1msec
 {
   for (i = 0; i < 0xc49b; i++) ;
 }
```

```
}

/*****************************************************************************
*
* ezErrorCheck - Function is intended to be used as a means to quickly determine if a function
*                has returned a non-zero (hence an error) return code. All driver and system
*                services functions return a value of zero for success and a non-zero value
*                when a failure occurs.  This function makes all LEDs glow dimly when a non
*                zero value is passed to it.
*
*****************************************************************************/
void ezErrorCheck(u32 Result)
{
  while (Result != 0)
    {
      ezCycleLEDs();
    }
}

/*****************************************************************************
*
* ezEnableVideoEncoder - Enables the AD7179 Video Encoder IC
*
*****************************************************************************/
void ezEnableVideoEncoder(void)
{
  adi_flag_Open(ADI_FLAG_PF6); // Open PF6

  // ADSP-BF537 Blackfin PF6 pin must be set as an Output
  adi_flag_SetDirection(ADI_FLAG_PF6, ADI_FLAG_DIRECTION_OUTPUT);
  ssync();

  adi_flag_Clear(ADI_FLAG_PF6); // Clear bit to reset ADV7179, Blackfin pin PF6
  ssync();

  adi_flag_Set(ADI_FLAG_PF6);
  ssync();
}
```

adi_itu656.c

```
/*****************************************************************************
*
* Device: ADSP-BF537
* Osc: SCLK = 120MHz
* File Name: "adi_itu656.c"
* Author: Dominick O' Brien
* Date: 21-Nov-06
* Version 1.00
* Modified version of Analog Device's "adi_itu656.c" found in VisualDSP++4.5
* References: "..\Blackfin\Examples\ADSP-BF533 EZ-Kit Lite\Drivers\PPI\Streaming.."
*
*****************************************************************************/
#include <services/services.h> // System Services Definitions
#include "adi_itu656.h" // ITU-656 Utilities Header File
```

```

/******************************************************************************
*
* Constants
*
******************************************************************************/
#define ADI_ITU656_EAV 1 // Defines End of Active Video
#define ADI_ITU656_SAV 2 // Defines Start of Active video

/******************************************************************************
*
* Function Prototypes
*
******************************************************************************/
static void generate_XY (
  u32 scanline, // Current Scanline Number
  ADI_ITU656_FRAME_TYPE frametype, // Video Frame Type
  u8 *preambleXY, // Holds the Calculated XY Value for EAV/SAV
  u32 videostatus // Indicates XY Calculation for EAV or SAV
);

static void calculate_address (
  u8 *frame_ptr, // Pointer to the Formatted Video Frame in Memory
  ADI_ITU656_FRAME_TYPE frametype, // Frame Type of the Formatted Memory
  u8 **address1, // Holds Address of Field 1 First Active Line's Active Data Start
                 // Address (for Interlaced Frame Type) OR First Active Line's Active Data Start
                 // Address (for Progressive Frame type)
  u8 **address2, // Holds Address of Field 2 First Active Line's Active Data Start Address (for
                 // Interlaced Frame Type)
  u32 *f1start, // Holds Field1 Active Line Start Value (for Interlaced Frame Format) OR Active
                // Line Start Value (for Progressive Frame Format)
  u32 *f1end, // Holds Field1 Active Line End Value (for Interlaced Frame Format) OR Active
              // Line End Value (for Progressive Frame Format)
  u32 *f2start, // Holds Field2 Active Line Start Value (for Interlaced Frame Format)
  u32 *f2end, // Holds Field2 Active Line End Value (for Interlaced Frame Format)
  u32 *widthcount // Holds the Value of NTSC/PAL Frame Width
);

/******************************************************************************
*
* adi_itu656_FrameFormat - This function formats an area in memory into a video frame active
*                          fields set blank.
*
******************************************************************************/
void adi_itu656_FrameFormat ( u8 *frame_ptr,ADI_ITU656_FRAME_TYPE frametype )
{
  u32 i;
  u32 j;
  u32 linecount;
  u32 blankcount;
  u32 widthcount;
  u8 preambleXY;

  switch (frametype) { // Switch to Frame Type
    case (ADI_ITU656_NTSC_IL): // Format frame as NTSC Interlaced or NTSC Progressive
    case (ADI_ITU656_NTSC_PR):
            linecount = ADI_ITU656_NTSC_HEIGHT;
            blankcount = ADI_ITU656_NTSC_BLANKING;
            widthcount = ADI_ITU656_NTSC_WIDTH;
            break;

         case (ADI_ITU656_PAL_IL): // Format frame as PAL Interlaced or PAL Progressive
         case (ADI_ITU656_PAL_PR):
            linecount = ADI_ITU656_PAL_HEIGHT;
            blankcount = ADI_ITU656_PAL_BLANKING;
            widthcount = ADI_ITU656_PAL_WIDTH;
            break;

         default: // Default as NTSC Frame
            linecount = ADI_ITU656_NTSC_HEIGHT;
            blankcount = ADI_ITU656_NTSC_BLANKING;
            widthcount = ADI_ITU656_NTSC_WIDTH;
            break;
      }

      for(i = 1; i <= linecount; i++) // Formats Frame Memory as EAV, Blanking, SAV, Active lines
      {
         // Generate BT656 Preamble
         generate_XY(i,frametype,&preambleXY,ADI_ITU656_EAV); // EAV - FF 00 00 XY
         *frame_ptr++ = 0xFF;
         *frame_ptr++ = 0x00;
         *frame_ptr++ = 0x00;
         *frame_ptr++ = preambleXY;

         for(j = 0; j < (blankcount / 2); j++) // Blanking
         {
            *frame_ptr++ = 0x80;
            *frame_ptr++ = 0x10;
         }

         generate_XY(i,frametype,&preambleXY,ADI_ITU656_SAV); // SAV - FF 00 00 XY
         *frame_ptr++ = 0xFF;
         *frame_ptr++ = 0x00;
         *frame_ptr++ = 0x00;
          *frame_ptr++ = preambleXY;

         for(j = 0; j < (widthcount); j++) // Output Empty Horizontal Data to Blank All Lines
         {
            *frame_ptr++ = 0x80;
            *frame_ptr++ = 0x10;
         }
      }
   }

   /******************************************************************************
   *
   * adi_itu656_FrameFill - This function fills the active video portion(s) of a formatted frame with
   *                         a specified colour.
   *
   ******************************************************************************/
   void adi_itu656_FrameFill ( u8 *frame_ptr,ADI_ITU656_FRAME_TYPE frametype,u8
   *ycbcr_data )
```

```
{
 u32 i;
 u32 j;
 u32 f1start;
 u32 f1end;
 u32 f2start;
 u32 f2end;
 u32 widthcount;
 u8 *address1;
 u8 *address2;

 address1 = frame_ptr; // Initialise the Pointers
 address2 = frame_ptr;

 // Calculate the Active Line Address & Update Widthcount, Frame Field Start and End Values
calculate_address (frame_ptr,frametype,&address1,&address2,&f1start,&f1end,&f2start,
 &f2end,&widthcount);

 // Paints Active Lines with Provided YCbCr Colour Value
 // Paints Field1 if frameformat is Interlaced or Whole frame if frameformat is Progressive
 for(i = f1start; i <= f1end; i++)
 {
   for(j = 0; j < (widthcount / 2); j++) // Output YCbCr data (4:2:2 format)
   {
     *address1++ = *ycbcr_data;
     *address1++ = *(ycbcr_data+1);
     *address1++ = *(ycbcr_data+2);
     *address1++ = *(ycbcr_data+3);
   }
 }

 if ((frametype = = ADI_ITU656_NTSC_IL) || (frametype = = ADI_ITU656_NTSC_PR))
 {
   address1 = address1 + 276;
 }

else
   address1 = address1 + 288;

 // Paints Field2 only when frametype is Interlaced
 if ((frametype = = ADI_ITU656_NTSC_IL) || (frametype = = ADI_ITU656_PAL_IL))
 {
   for(i = f2start; i <= f2end; i++)
   {
     for(j = 0; j < (widthcount / 2); j++) // Output YCbCr data (4:2:2 format)
     {
       *address2++ = *ycbcr_data;
       *address2++ = *(ycbcr_data+1);
       *address2++ = *(ycbcr_data+2);
       *address2++ = *(ycbcr_data+3);
     }

     if (frametype = = ADI_ITU656_NTSC_IL)
     {
       address2 = address2 + 276;
     }
```

```
            else
              address2 = address2 + 288;
          }
        }
      }

      /*****************************************************************************
      *
      * adi_itu656_RowFill - This function fills a row of pixels in active video portion of formatted
      *                                  frame with specified colour
      *
      *****************************************************************************/
      void adi_itu656_RowFill ( u8 *frame_ptr,ADI_ITU656_FRAME_TYPE frametype,u32
      row_value,u8 *ycbcr_data )
      {
        u32     i,j,f1start,f1end,f2start,f2end,widthcount;
        u8      *address1,*address2;

        // Initialise the pointers
        address1 = frame_ptr;
        address2 = frame_ptr;

        // Calculate the active line address & update widthcount, frame field start and end values
        calculate_address(frame_ptr,frametype,&address1,&address2,&f1start,&f1end,&f2start,
        &f2end,&widthcount);

        // Paints active lines with provided YCbCr color value
        // Paints Field1 if frameformat is interlaced OR whole frame if frameformat is Progressive
        for(i = f1start; i <= f1end; i++)
        {
          if (i == row_value) // Is this the row to be painted with YCbCr data?
          {
            for(j = 0; j < (widthcount / 2); j++) // Output YCbCr data (4:2:2 format)
            {
              *address1++ = *ycbcr_data;
              *address1++ = *(ycbcr_data+1);
              *address1++ = *(ycbcr_data+2);
              *address1++ = *(ycbcr_data+3);
            }
          }

          else // Paint all other rows as blank
          {
            for(j = 0; j < (widthcount); j++)
            {
              *address1++ = 0x80;
              *address1++ = 0x10;
            }
          }

          if ((frametype == ADI_ITU656_NTSC_IL) || (frametype == ADI_ITU656_NTSC_PR))
          {
            address1 = address1 + 276;
          }

          else
          {
            address1 = address1 + 288;
          }
        }

        // Paints Field2 only when frametype is Interlaced
        if ((frametype == ADI_ITU656_NTSC_IL) || (frametype == ADI_ITU656_PAL_IL))
        {
          for(i = f2start; i <= f2end; i++)
          {
            if (i == row_value) // Is this the row to be painted with YCbCr data?
            {
              for(j = 0; j < (widthcount / 2); j++) // Output YCbCr data (4:2:2 format)
              {
                *address2++ = *ycbcr_data;
                *address2++ = *(ycbcr_data+1);
                *address2++ = *(ycbcr_data+2);
                *address2++ = *(ycbcr_data+3);
              }
            }

            else // Paint all other rows as blank
            {
              for(j = 0; j < (widthcount); j++)
              {
                *address2++ = 0x80;
                 *address2++ = 0x10;
              }
            }

            if (frametype == ADI_ITU656_NTSC_IL)
            {
              address2 = address2 + 276;
            }

            else
            {
              address2 = address2 + 288;
            }
          }
        }
      }

      /*****************************************************************************
      *
      * adi_itu656_ ColumnFill - This function fills a column of pixels in active video portion of
      *                          formatted frame with a specified colour.
      *
      *****************************************************************************/
      void adi_itu656_ColumnFill ( u8 *frame_ptr,ADI_ITU656_FRAME_TYPE frametype,u32
      column_value,u8 *ycbcr_data )
      {
        u32    i,j,f1start,f1end,f2start,f2end,widthcount;
        u8     *address1,*address2;

        address1 = frame_ptr; // Initialise the pointers
```

```
address2 = frame_ptr;

// Calculate the active line address & update widthcount, frame field start and end values
calculate_address(frame_ptr,frametype,&address1,&address2,&f1start,&f1end,&f2start,
&f2end,&widthcount);

// Paints active lines with provided YCbCr color value
// Paints Field1 if frameformat is interlaced OR whole frame if frameformat is Progressive
for(i = f1start; i <= f1end; i++)
{
  for(j = 0; j < (widthcount / 2); j++)
  {
    if (j == column_value) // Is this the column to be painted with YCbCr data?
    { // Yes, Output YCbCr data (4:2:2 format)
      *address1++ = *ycbcr_data;
      *address1++ = *(ycbcr_data+1);
      *address1++ = *(ycbcr_data+2);
      *address1++ = *(ycbcr_data+3);
    }

    else
    { // No - Paint the column as blank
      *address1++ = 0x80;
      *address1++ = 0x10;
      *address1++ = 0x80;
      *address1++ = 0x10;
    }
  }

  if ((frametype == ADI_ITU656_NTSC_IL) || (frametype == ADI_ITU656_NTSC_PR))
  {
    address1 = address1 + 276;
  }

  else
  {
    address1 = address1 + 288;
  }
}

// Paints Field2 only when frametype is Interlaced
if ((frametype == ADI_ITU656_NTSC_IL) || (frametype == ADI_ITU656_PAL_IL))
{
  for(i = f2start; i <= f2end; i++)
  {
    for(j = 0; j < (widthcount / 2); j++) // Output YCbCr data (4:2:2 format)
    {
      if (j == column_value) // Is this the column to be painted with YCbCr data?
      { // Yes, Output YCbCr data (4:2:2 format)
        *address2++ = *ycbcr_data;
        *address2++ = *(ycbcr_data+1);
        *address2++ = *(ycbcr_data+2);
        *address2++ = *(ycbcr_data+3);
      }

      else
```

```
            { // No - Paint the column as blank
              *address2++ = 0x80;
              *address2++ = 0x10;
              *address2++ = 0x80;
              *address2++ = 0x10;
            }
          }

          if (frametype == ADI_ITU656_NTSC_IL)
          {
            address2 = address2 + 276;
          }

          else
          {
            address2 = address2 + 288;
          }
        }
      }
    }

    /*****************************************************************************
    *
    * generate_XY - This function generates the XY preamble for EAV & SAV
    *
    *****************************************************************************/
    static void generate_XY ( u32 scanline,ADI_ITU656_FRAME_TYPE frametype,u8
    *preambleXY,u32 videostatus )
    {
      if(frametype == ADI_ITU656_NTSC_IL) // Frame type is NTSC interlaced
      {
        if((scanline >= 1) && (scanline <= 3)) // 1-3 Blanking Field 2
        {
          if(videostatus == ADI_ITU656_EAV)
          {
            *preambleXY = 0xF1;
          }

          else if(videostatus == ADI_ITU656_SAV)
          {
            *preambleXY = 0xEC;
           }
         }

        else if((scanline >= 4) && (scanline <= 22)) // 4-22 Blanking Field 1
        {
          if(videostatus == ADI_ITU656_EAV)
          {
            *preambleXY = 0xB6;
          }

          else if(videostatus == ADI_ITU656_SAV)
          {
            *preambleXY = 0xAB;
          }
        }

        else if((scanline >= 23) && (scanline <= 262)) // 23-262 Active Video Field 1
        {
          if(videostatus == ADI_ITU656_EAV)
          {
            *preambleXY = 0x9D;
          }

          else if(videostatus == ADI_ITU656_SAV)
          {
            *preambleXY = 0x80;
          }
        }

        else if((scanline >= 263) && (scanline <= 265)) // 263-265 Blanking Field 1
        {
          if(videostatus == ADI_ITU656_EAV)
          {
            *preambleXY = 0xB6;
          }

          else if(videostatus == ADI_ITU656_SAV)
          {
            *preambleXY = 0xAB;
          }
        }

        else if((scanline >= 266) && (scanline <= 285)) // 266-285 Blanking Field 2
        {
          if(videostatus == ADI_ITU656_EAV)
          {
            *preambleXY = 0xF1;
          }

          else if(videostatus == ADI_ITU656_SAV)
          {
            *preambleXY = 0xEC;
          }
        }

        else if((scanline >= 286) && (scanline <= 525)) // 286-525 Active Video Field 2
        {
          if(videostatus == ADI_ITU656_EAV)
          {
            *preambleXY = 0xDA;
          }

          else if(videostatus == ADI_ITU656_SAV)
          {
            *preambleXY = 0xC7;
          }
        }
      }

      else if(frametype == ADI_ITU656_PAL_IL) // Frame type is PAL interlaced
      {
```

```
      if((scanline >= 1) && (scanline <= 22)) // 1-22 Blanking Field 1
      {
        if(videostatus == ADI_ITU656_EAV)
        {
          *preambleXY = 0xB6;
        }

        else if(videostatus == ADI_ITU656_SAV)
        {
          *preambleXY = 0xAB;
        }
      }

      else if((scanline >= 23) && (scanline <= 310))  // 23-310 Active Video Field 1
      {
        if(videostatus == ADI_ITU656_EAV)
        {
          *preambleXY = 0x9D;
        }

        else if(videostatus == ADI_ITU656_SAV)
        {
          *preambleXY = 0x80;
        }
      }

      else if((scanline >= 311) && (scanline <= 312)) // 311-312 Blanking Field 1
      {
        if(videostatus == ADI_ITU656_EAV)
        {
          *preambleXY = 0xB6;
        }

        else if(videostatus == ADI_ITU656_SAV)
        {
          *preambleXY = 0xAB;
        }
      }

      else if((scanline >= 313) && (scanline <= 335)) // 313-335 Blanking Field 2
      {
        if(videostatus == ADI_ITU656_EAV)
        {
          *preambleXY = 0xF1;
        }

        else if(videostatus == ADI_ITU656_SAV)
        {
          *preambleXY = 0xEC;
         }
      }

      else if((scanline >= 336) && (scanline <= 623)) // 336-623 Active Video Field 2
      {
        if(videostatus == ADI_ITU656_EAV)
        {
            *preambleXY = 0xDA;
          }

          else if(videostatus == ADI_ITU656_SAV)
          {
            *preambleXY = 0xC7;
          }
        }

        else if((scanline >= 624) && (scanline <= 625)) // 624-625 Blanking Field 2
        {
          if(videostatus == ADI_ITU656_EAV)
          {
            *preambleXY = 0xF1;
          }

          else if(videostatus == ADI_ITU656_SAV)
          {
            *preambleXY = 0xEC;
          }
        }
      }

      else if(frametype == ADI_ITU656_NTSC_PR) // Frame type is NTSC Progressive
      {
        if((scanline >= 1) && (scanline <= 45)) // 1-45 Blanking
        {
          if(videostatus == ADI_ITU656_EAV)
          {
            *preambleXY = 0xB6;
          }

          else if(videostatus == ADI_ITU656_SAV)
          {
            *preambleXY = 0xAB;
          }
        }

        else if((scanline >= 46) && (scanline <= 525)) // 46-525 Active Video
        {
          if(videostatus == ADI_ITU656_EAV)
          {
            *preambleXY = 0x9D;
          }

          else if(videostatus == ADI_ITU656_SAV)
          {
            *preambleXY = 0x80;
          }
        }
      }

      else if(frametype == ADI_ITU656_PAL_PR) // Frame type is PAL Progressive
      {
        if((scanline >= 1) && (scanline <= 44)) // lines 1-44 Blanking
        {
          if(videostatus == ADI_ITU656_EAV)
          {
            *preambleXY = 0xB6;
          }

          else if(videostatus == ADI_ITU656_SAV)
          {
            *preambleXY = 0xAB;
          }
        }

        else if((scanline >= 45) && (scanline <= 620)) // lines 46-620 Active Video
        {
          if(videostatus == ADI_ITU656_EAV)
          {
            *preambleXY = 0x9D;
          }

          else if(videostatus == ADI_ITU656_SAV)
          {
            *preambleXY = 0x80;
          }
        }

        else if((scanline >= 621) && (scanline <= 625))  // lines 621-625 Blanking
        {
          if(videostatus == ADI_ITU656_EAV)
          {
            *preambleXY = 0xB6;
          }

          else if(videostatus == ADI_ITU656_SAV)
          {
            *preambleXY = 0xAB;
          }
        }
    }
}

/******************************************************************************
*
* calculate_address - This function calculates active line address & updates widthcount, frame
*                              field start and end values
*
******************************************************************************/
static void calculate_address ( u8 *frame_ptr,ADI_ITU656_FRAME_TYPE frametype,u8
**address1,u8 **address2,u32 *f1start,
u32 *f1end,u32 *f2start,u32 *f2end,u32 *widthcount)
{
  switch (frametype)
  { // Switch to Frame Type
    case (ADI_ITU656_NTSC_IL): // Frame format is NTSC Interlaced
      *widthcount = ADI_ITU656_NTSC_WIDTH;
      *f1start = ADI_ITU656_NTSC_ILF1_START; // active line start - Field1
      *f1end = ADI_ITU656_NTSC_ILF1_END; // active line end - Field1

```

```
            // Calculate Field 1 first active line's active data start address
            *address1 = frame_ptr + ((ADI_ITU656_NTSC_ILF1_START-1) * 1716) + 276;
            *f2start = ADI_ITU656_NTSC_ILF2_START;  // active line start - Field2
            *f2end = ADI_ITU656_NTSC_ILF2_END;      // active line end - Field2

           // Calculate Field 2 first active line's active data start address
            *address2 = frame_ptr + ((ADI_ITU656_NTSC_ILF2_START-1) * 1716) + 276;
            break;

          case (ADI_ITU656_PAL_IL): // Frame format is PAL Interlaced
            *widthcount = ADI_ITU656_PAL_WIDTH;
            *f1start = ADI_ITU656_PAL_ILF1_START; // active line start - Field1
            *f1end = ADI_ITU656_PAL_ILF1_END; // active line end - Field1

            // Calculate Field 1 first active line's active data start address
            *address1 = frame_ptr + ((ADI_ITU656_PAL_ILF1_START-1) * 1728) + 288;
            *f2start = ADI_ITU656_PAL_ILF2_START; // active line start - Field2
            *f2end = ADI_ITU656_PAL_ILF2_END; // active line end - Field2

            // Calculate Field 2 first active line's active data start address
            *address2 = frame_ptr + ((ADI_ITU656_PAL_ILF2_START-1) * 1728) + 288;
            break;

          case (ADI_ITU656_NTSC_PR): // Frame format is NTSC Progressive
            *widthcount = ADI_ITU656_NTSC_WIDTH;
            *f1start = ADI_ITU656_NTSC_PRF_START; // active line start
            *f1end = ADI_ITU656_NTSC_PRF_END; // active line end

            // Calculate First active line's active data start address
            *address1 = frame_ptr + ((ADI_ITU656_NTSC_PRF_START-1)* 1716) + 276;
            break;

          case (ADI_ITU656_PAL_PR): // Frame format is PAL Progressive
            *widthcount = ADI_ITU656_PAL_WIDTH;
            *f1start = ADI_ITU656_PAL_PRF_START; // active line start
            *f1end = ADI_ITU656_PAL_PRF_END; // active line end

            // Calculate First active line's active data start address
            *address1 = frame_ptr + ((ADI_ITU656_PAL_PRF_START-1)* 1728) + 288;
            break;

          default:    // Default as NTSC Progressive
            *widthcount = ADI_ITU656_NTSC_WIDTH;
            *f1start = ADI_ITU656_NTSC_PRF_START; // active line start
            *f1end = ADI_ITU656_NTSC_PRF_END; // active line end

            // Calculate First active line's active data start address
            *address1 = frame_ptr + ((ADI_ITU656_NTSC_PRF_START-1)* 1716) + 276;
            break;
        }
      }
```

adv7179.c

```
/*****************************************************************************
```

```
*
* Device: ADSP-BF537
* Osc: SCLK = 120MHz
* File Name: "adv7179.c"
* Author: Dominick O' Brien
* Date: 21-Nov-06
* Version 1.00
* References: "..\Blackfin\Examples\ADSP-BF533 EZ-Kit Lite\Drivers\PPI\Streaming.."
*
*****************************************************************************/

/*****************************************************************************
* Description - This is the driver source code for the ADV7179 Video Encoder. It is layered
*                      on top of the PPI and TWI device drivers, which are configured for the specific
*                      use ADV7179 peripheral.
*****************************************************************************/

/*****************************************************************************
*
* ADV7179 device macro define
*
*****************************************************************************/
#define ADI_ADV7179_DEVICE
#include "adi_adv717x.c"          // Driver Register Access Includes
```

adv717x.c - Note: This is a standard system service that is un-modified within this application. Reference "..\VisualDSP4.5\Blackfin\lib\src\drivers\encoder" directory. To utilise this program it must not be directly included within a VisualDSP++4.5 project; however it has to be situated within a project directory.

main.c

```
/*****************************************************************************
*
* Device: ADSP-BF537
* Osc: SCLK = 120MHz
* File Name: "main.c"
* Author: Dominick O' Brien
* Date: 24-Nov-06
* Version 1.00
* Modified version of Analog Device's "ezkitutilities.c" found in VisualDSP++4.5
* References: "..\Blackfin\Examples\ADSP-BF533 EZ-Kit Lite\Drivers\PPI\Streaming.."
* Target Processor: ADSP-BF537
* Target Tools Revision: ADSP VisualDSP++ v4.5 (September 2006 Update)
*
*****************************************************************************/
#include <services\services.h> // System Services
#include <drivers\adi_dev.h> // Device Manager Includes
#include <drivers\ppi\adi_ppi.h> // PPI Driver Includes
#include <defBF537.h> // Include all MMR's and bit defs
#include <drivers\encoder\adi_adv717x.h> // 7179 Device Driver Includes
```

```
#include "ezkitutilities.h" // EZ-Kit Utilities
#include "adi_itu656.h" // ITU656 Utilities

/*********************************************************************************
*
* ADSP-BF537 Switch Settings
*
*********************************************************************************
*
* SW1: ALL OFF
* SW2: ALL ON
* SW3: ALL OFF
* SW4: OFF, ON, OFF, ON
* SW5: ALL ON
* SW6: ALL ON
* SW7: ALL ON
* SW8: ON, ON, OFF, OFF, OFF, OFF
*
*********************************************************************************/

/*********************************************************************************
*
* A/V Extender Board Jumper Settings
*
*********************************************************************************
*
* JP1: NOT USED
* JP2: NOT USED
* JP3: JP3.5/7 & JP3.6/8 --> Processor's TWI
* JP4: JP4.1/2 & JP4.3/4 --> 27MHz A V extender card onboard clock to source PPI CLK
* JP5: JP5.3/4 --> Enables PPI0 to drive VID_OUT
* JP6: NOT USED
* JP7: NOT USED
* JP8: JP8.1/3 & JP8.2/4 --> Selects PPI0 as source
* JP8.7/8  --> Enables VID_OUT bus sync
* JP9: JP9.1/3 --> Connect AD7179 reset to reset flag
* JP10: NOT USED
*
*********************************************************************************/

/*********************************************************************************
*
*           External Connections
*
*********************************************************************************
*
* Connect a monitor to the A-V Extender card video-out connector. The video connectors are
* the bank of 6 RCA-style jacks on the A-V Extender card labelled as J7.
*
*   J7  +--------------------------------------------------+
*       |     O      O < Video out       O      | (white)
*
*       |     O      O                   O      | (red)
*       +--------------------------------------------------+
*
*********************************************************************************/
```

```

/*****************************************************************************
*
* Enumerations and defines
*
*****************************************************************************/
#define ENCODER_PPI (0) // ADSP-BF537 has only 1 PPI called PPI0
#define NUM_BUFFERS (30) // Colour Change Rate = (NUM_BUFFERS/30)/second
// Colour Patterns
static u8 black[] = {0x80,0x10,0x80,0x10}; // Black pixel YCbCr format
static u8 blue[] = {0xF0,0x29,0x6E,0x29}; // Blue pixel YCbCr format
static u8 red[] = {0x5A,0x51,0xF0,0x51}; // Red pixel YCbCr format
static u8 magenta[] = {0xCA,0x6A,0xDE,0x6A}; // Magenta pixel YCbCr format
static u8 green[] = {0x36,0x91,0x22,0x91}; // Green pixel YCbCr format
static u8 cyan[] = {0xA6,0xAA,0x10,0xAA}; // Cyan pixel YCbCr format
static u8 yellow[] = {0x10,0xD2,0x92,0xD2}; // Yellow pixel YCbCr format
static u8 white[] = {0x80,0xEB,0x80,0xEB}; // White pixel YCbCr format

/*****************************************************************************
*
* Static data
*
*****************************************************************************/
// Create two areas in SDRAM that will each hold a 656 Frame
static u8 PingFrame[ADI_ITU656_NTSC_LINE_WIDTH * ADI_ITU656_NTSC_HEIGHT];
static u8 PongFrame[ADI_ITU656_NTSC_LINE_WIDTH * ADI_ITU656_NTSC_HEIGHT];

ADI_DEV_2D_BUFFER PingBuffer[NUM_BUFFERS]; // Create two buffer chains.
ADI_DEV_2D_BUFFER PongBuffer[NUM_BUFFERS];

// DMA Manager data (base memory + memory for 1 DMA channel)
static u8 DMAMgrData[ADI_DMA_BASE_MEMORY + (ADI_DMA_CHANNEL_MEMORY
* 1)];
// Deferred Callback Manager data (memory for 1 service plus 4 posted callbacks)
static u8 DCBMgrData[ADI_DCB_QUEUE_SIZE + (ADI_DCB_ENTRY_SIZE)*4];

// Device Manager data (base memory + memory for 3 devices)
// Memory for 3 devices is required because usage of a 717x device results in the usage of the
// PPI and SPI devices.
static u8 DevMgrData[ADI_DEV_BASE_MEMORY + (ADI_DEV_DEVICE_MEMORY *
3)];
ADI_DEV_DEVICE_HANDLE AD7179DriverHandle; // Handle to the ADV7179 Driver

/*****************************************************************************
*
* ExceptionHandler - An Exception error should never happen but just in case if one occurs all
*                    the LEDs will light up.
*
*****************************************************************************/
static ADI_INT_HANDLER(ExceptionHandler) // Exception Handler
{
  ezErrorCheck(1);
  return(ADI_INT_RESULT_PROCESSED);
}

/*****************************************************************************
```

```
*
* HWErrorHandler - A Hardware error should never happen but just in case if one occurs all
*                  the LEDs will light up.
*
****************************************************************************/
static ADI_INT_HANDLER(HWErrorHandler) // Hardware Error Handler
{
  ezErrorCheck(1);
  return(ADI_INT_RESULT_PROCESSED);
}

/****************************************************************************
*
* Callback - Callback occurs when the PPI has completed processing of the last buffer in the
*            Ping & Pong Buffer chains.
*
****************************************************************************/
static void Callback(void *AppHandle,u32  Event,void *pArg)
{
  ADI_DEV_BUFFER *pBuffer; // Pointer to the Buffer that was processed

  switch (Event)
  { // Case Of (event type)
    case ADI_DEV_EVENT_BUFFER_PROCESSED: // CASE (buffer processed)
      // When the buffer chain was created, the CallbackParameter value for the buffer that was
      // generating the callback was set to be the address of the first buffer in the chain.
      // So here in the callback that value is passed in as the pArg parameter.
      pBuffer = (ADI_DEV_BUFFER *)pArg;
      break;

    case ADI_DEV_EVENT_DMA_ERROR_INTERRUPT: // Case (an Error)
    case ADI_PPI_EVENT_ERROR_INTERRUPT:
      ezTurnOnAllLEDs(); // Turn on all LEDs and wait for help
      while (1) ;
  }
}

void main(void)
{
  // Table of PPI driver configuration values
  ADI_DEV_CMD_VALUE_PAIR ConfigurationTable [] =
  {
   {ADI_DEV_CMD_SET_DATAFLOW_METHOD,
    (void*)ADI_DEV_MODE_CHAINED_LOOPBACK},
    {ADI_PPI_CMD_SET_CONTROL_REG, (void *)0x0082},
    {ADI_PPI_CMD_SET_LINES_PER_FRAME_REG,
    (void*)ADI_ITU656_NTSC_HEIGHT},
    {ADI_DEV_CMD_SET_STREAMING, (void *)TRUE},
    {ADI_DEV_CMD_END, NULL},
  };

  ADI_DCB_HANDLE DCBManagerHandle; // Handle to the Callback Service Manager
  ADI_DMA_MANAGER_HANDLE DMAManagerHandle; // Handle to the DMA Manager
  ADI_DEV_MANAGER_HANDLE DeviceManagerHandle; // Handle to the Device Manager

  u32 ResponseCount; // Response Counter
```

```
int i = 0; // Counter

ezInit(1); // Initialise the EZ-Kit
ezTurnOffAllLEDs();// Turn off all LEDs

// Initialise the Interrupt Manager and hook the exception and hardware error interrupts
ezErrorCheck(adi_int_Init(NULL, 0, &ResponseCount, NULL));
ezErrorCheck(adi_int_CECHook(3, ExceptionHandler, NULL, FALSE));
ezErrorCheck(adi_int_CECHook(5, HWErrorHandler, NULL, FALSE));

// Initialise the Deferred Callback Manager and setup a queue
ezErrorCheck(adi_dcb_Init(&DCBMgrData[0],
                          ADI_DCB_QUEUE_SIZE,
                          &ResponseCount,
                          NULL));

ezErrorCheck(adi_dcb_Open(14,
                            &DCBMgrData[ADI_DCB_QUEUE_SIZE],
                            (ADI_DCB_ENTRY_SIZE)*4,
                            &ResponseCount,
                            &DCBManagerHandle));

// Initialise the flag service, memory is not passed because callbacks are not being used
ezErrorCheck(adi_flag_Init(NULL, 0, &ResponseCount, NULL));

for (i = EZ_FIRST_LED; i < EZ_NUM_LEDS; i++) // Enable all LEDs
{
  ezInitLED(i);
}

ezErrorCheck(adi_dma_Init(DMAMgrData, // Initialise the DMA Manager
             sizeof(DMAMgrData),
             &ResponseCount,
             &DMAManagerHandle,
             NULL));

ezErrorCheck(adi_dev_Init(DevMgrData, // Initialise the Device Manager
                          sizeof(DevMgrData),
                          &ResponseCount,
                          &DeviceManagerHandle,
                          NULL));

// Initialise the two frames and make them both different colours
adi_itu656_FrameFormat (PingFrame,ADI_ITU656_NTSC_PR);
adi_itu656_FrameFormat (PongFrame,ADI_ITU656_NTSC_PR);
adi_itu656_FrameFill (PingFrame,ADI_ITU656_NTSC_PR,white); // WHITE
adi_itu656_FrameFill (PongFrame,ADI_ITU656_NTSC_PR,blue); // BLUE

ezEnableVideoEncoder(); // Enable video encoder (7179)
ezDelay(300); // Give the encoder time to sync

// Open the AD7179 Driver for Output
ezErrorCheck(adi_dev_Open(DeviceManagerHandle, // Handle controlling the Device
                        &ADIADV7179EntryPoint, // Address of Entry Point
                        ENCODER_PPI,  // Number identifying which Device is Opened
                        NULL, // No Client Handle
```

```
                              &AD7179DriverHandle, // Handle Address
                              ADI_DEV_DIRECTION_OUTBOUND, // Data Direction
                              DMAManagerHandle, // Handle to DMA Manager
                              DCBManagerHandle, // Handle to Callback Manager
                              Callback)); // Callback

    // Set PPI Device Number
    ezErrorCheck(adi_dev_Control(AD7179DriverHandle, // Handle identifying Device
                              // Command Identifier
                              ADI_ADV717x_CMD_SET_PPI_DEVICE_NUMBER,
                              (void*)0)); // PPI Device Number

    // Open PPI Device
    ezErrorCheck(adi_dev_Control(AD7179DriverHandle, // Handle identifying Device
                              ADI_ADV717x_CMD_SET_PPI_STATUS, // Command Identifier
                              // Address of Command Specific Parameter
                              (void*)ADI_ADV717x_PPI_OPEN));

    // Create a buffer chain that points to the PingFrame. Each buffer points to the same PingFrame
    // so the PingFrame will be displayed NUM_BUFFERS times. NUM_BUFFERS is sized to
    // keep the display busy for 1 second. Place a callback on only the last buffer in the chain.
    // Make the CallbackParameter (the value that gets passed to the callback function as the pArg
    // parameter) point to the first buffer in the chain. This way, when the callback goes off, the
    // callback function can requeue the whole chain if the loopback mode is off.

    for (i = 0; i < NUM_BUFFERS; i++) // Populate the PingBuffer
    {
      PingBuffer[i].Data = PingFrame; // Point to PingFrame Data
      PingBuffer[i].ElementWidth = 2;
      PingBuffer[i].XCount = (ADI_ITU656_NTSC_LINE_WIDTH/2);
      PingBuffer[i].XModify = 2;
      PingBuffer[i].YCount = ADI_ITU656_NTSC_HEIGHT;
      PingBuffer[i].YModify = 2;
      PingBuffer[i].CallbackParameter = NULL;
      PingBuffer[i].pNext = &PingBuffer[i + 1];
    }

    PingBuffer[NUM_BUFFERS - 1].CallbackParameter = &PingBuffer[0];
    PingBuffer[NUM_BUFFERS - 1].pNext = NULL;

    for (i = 0; i < NUM_BUFFERS; i++) // Populate the PongBuffer
    {
      PongBuffer[i].Data = PongFrame; // Point to PongFrame Data
      PongBuffer[i].ElementWidth = 2;
      PongBuffer[i].XCount = (ADI_ITU656_NTSC_LINE_WIDTH/2);
      PongBuffer[i].XModify = 2;
      PongBuffer[i].YCount = ADI_ITU656_NTSC_HEIGHT;
      PongBuffer[i].YModify = 2;
      PongBuffer[i].CallbackParameter = NULL;
      PongBuffer[i].pNext = &PongBuffer[i + 1];
    }

    PongBuffer[NUM_BUFFERS - 1].CallbackParameter = &PongBuffer[0];
    PongBuffer[NUM_BUFFERS - 1].pNext = NULL;

    // Configure the AD7179 Dataflow Method
    ezErrorCheck(adi_dev_Control(AD7179DriverHandle, // Handle identifying Device
                 ADI_DEV_CMD_SET_DATAFLOW_METHOD, // Command Parameter
                 (void *)ADI_DEV_MODE_CHAINED_LOOPBACK)); // Outbound Loopback

    // Give the device the Ping and Pong buffer chains
    ezErrorCheck(adi_dev_Write(AD7179DriverHandle, // Handle identifying Device
                               ADI_DEV_2D, // 2D Buffer
                               (ADI_DEV_BUFFER *)&PingBuffer)); // Point to PingBuffer

    ezErrorCheck(adi_dev_Write(AD7179DriverHandle, // Handle identifying Device
                               ADI_DEV_2D, // 2D Buffer
                               (ADI_DEV_BUFFER *)&PongBuffer)); // Point to PongBuffer

    // Enable data flow
    ezErrorCheck(adi_dev_Control(AD7179DriverHandle, // Handle identifying Device
                                 ADI_DEV_CMD_SET_DATAFLOW// Command Parameter
                                 (void *)TRUE)); // Turn on Dataflow
     while(1);
   }
```

Appendix F - Application Source Code

ezkitutilities.h - Note: See *Appendix E – ezkitutilities.h*

adi_itu656.h - Note: See *Appendix E – adi_itu656.h*

CAN.h - Note: See *Appendix E - CAN_Test.h*. *CAN.h* and *CAN_Test.h* (*Appendix E*) are identical except that Line 51 of *CAN_Test.h* is omitted from *CAN.h*

ezkitutilities.c - Note: See *Appendix E – ezkitutilities.c*

adi_itu656.c - Note: See *Appendix E – adi_itu656.c*

adv7179.c - Note: See *Appendix E – adv7179.c*

adv717x.c - Note: See *Appendix E – adv717x.c*

CAN_Init.c.

```
/*****************************************************************************
*
* Device: ADSP-BF537
* Osc: SCLK = 120MHz
* File Name: "CAN_Init.c"
* Author: Dominick O' Brien
* Date: 19-Jan-07
* Version 1.00
*
*****************************************************************************/
#include " CAN.h" // CAN Utilities

/*****************************************************************************
*
* Init_CAN_Port – Sets up the Ports for CAN use and configured the PFx pins for access to the
*                         on-board LEDs.
*
*****************************************************************************/
void Init_CAN_Port ()
{
  short temp_fix;
  // Configure CAN RX and CAN TX pins on GPIO Port
```

```
    temp_fix = *pPORT_MUX;
    ssync();

    *pPORT_MUX = PJCE_CAN; // Enable CAN Pins On Port J
    ssync();
    *pPORT_MUX = PJCE_CAN; // #22 work-around: write it a few times
    ssync();
    *pPORT_MUX = PJCE_CAN; // #22 work-around: write it a few times
    ssync();

    temp_fix = *pPORT_MUX; // #22 work-around: read PORT_MUX after writing
    ssync();

    // Configure Port F pins for LED access
    *pPORTFIO_DIR   = 0x0FC0; // Enable PF6-11 As Outputs (LEDs)
    ssync();
} // End Init_Port ()

/*****************************************************************************
*
* Init_CAN_Timing – Sets up the CAN_TIMING & CLOCK Registers
*
*****************************************************************************/
void Init_CAN_Timing()
{
  //        =================================================
  //        BIT TIMING:
  //
  //        CCLK 600 MHz
  //        SCLK 120 MHz
  //
  //        CAN_CLOCK  : Prescaler (BRP)
  //        CAN_TIMING : SJW = 2, TSEG2 = 3, TSEG1 = 5
  //
  //        tBIT = TQ x (1 + (TSEG1 + 1) + (TSEG2 + 1))
  //        2e-6 = TQ x (1 + (5 + 1) + (3 + 1))
  //        TQ = 1.82e-7
  //
  //        TQ = (BRP+1) / SCLK
  //        1.82e-7 = (BRP+1) / 120e6
  //        (BRP+1) = 21.84
  //        BRP = 20.84 ~ 21
  //        =================================================
  //        Set Bit Configuration Registers ...
  //        =================================================
  *pCAN_TIMING = 0x0235;
   *pCAN_CLOCK  = 21; // [0x15] 500kHz CAN Clock :: tBIT = 2us
  ssync();
} // End Init_CAN_Timing()

/*****************************************************************************
*
* Init_CAN_Mailboxes – Configures Mailbox 24 to transmit a specific message ID with a
*                      message length of 8 bytes. Configures Mailbox 6 and 7 to each receive
*                      a specific message ID
*
```

```
*******************************************************************************/
void Init_CAN_Mailboxes()
{
  short msgID;                  // Variable for Mailbox 24 ID
  short msgID_OnB;          // Variable for Mailbox 7 ID
  short msgID_SPI;           // Variable for Mailbox 6 ID

  volatile char mbID;
  volatile char mbID_OnB; // Variable for Mailbox # for ON_B
  volatile char mbID_SPI; // Variable for Mailbox # for SPI
  // Mailbox 24 Will Transmit ACK  to the Network via ID 0x007
  msgID = 0x007;
  mbID  = 24;

  *(pCAN_MB_ID1(mbID)) = msgID << 2; // ID1, mask disabled, remote frame disable, 11 bit
                                                                    // identifier
  *(pCAN_MB_ID0(mbID)) = 0; // ID0 = all 0's
  *(pCAN_MB_LENGTH(mbID)) = 8; // DLC = 8 bytes

  // Mailbox 7 will Receive CAN Command from Network via ID 0x411
  // Mailbox 6 will Recieve CAN Command from Network via ID 0x189
  msgID_OnB = 0x411; // ID = dec 1041
  msgID_SPI = 0x189; // ID = dec 393
  mbID_OnB  = 7; // Mailbox 7
  mbID_SPI = 6; // Mailbox 6

  *(pCAN_MB_ID1(mbID_OnB)) = msgID_OnB << 2; // ID1, mask disabled, remote frame
                                                                              // disable, 11 bit identifier
  *(pCAN_MB_ID0(mbID_OnB)) = 0; // ID0 = all 0's
  *(pCAN_MB_ID1(mbID_SPI)) = msgID_SPI << 2; // ID1, mask disabled, remote frame
                                                                            // disable, 11 bit identifier
  *(pCAN_MB_ID0(mbID_SPI)) = 0; // ID0 = all 0's
  *(pCAN_MB_LENGTH(mbID_SPI)) = 8; // DLC = 8 bytes
} // End Init_CAN_Mailboxes()

/*******************************************************************************
*
* Init_Interrupts – Assigns interrupt priorities for CAN TX and CAN RX.
*
*******************************************************************************/
void Init_Interrupts()
{
  // Configure Interrupt Priorities
  *pSIC_IAR0 = 0x77717777; // PPI DMA IRQ : 1 = IVG8
  *pSIC_IAR1 = 0x47777777; // CAN RX IRQ : 4 = IVG11
  *pSIC_IAR2 = 0x77777775; // CAN TX IRQ : 5 = IVG12
  *pSIC_IAR3 = 0x77777777;

  // Register Interrupt Handlers and Enable Core Interrupts
  register_handler(ik_ivg11, CAN_RCV_HANDLER);
  register_handler(ik_ivg12, CAN_XMT_HANDLER);

  // Enable SIC Level Interrupts
  *pSIC_IMASK |= (IRQ_CAN_RX|IRQ_CAN_TX);
} // End Init_Interrupts
```

CAN_Functions.c

```
/*****************************************************************************
*
* Device: ADSP-BF537
* Osc: SCLK = 120MHz
* File Name: "CAN_Functions.c"
* Author: Dominick O' Brien
* Date: 22-Jan-07
* Version 1.00
*
*****************************************************************************/
#include "CAN.h" // CAN Utilities

/*****************************************************************************
*
* CAN_Setup_Interrupts – Enables Mailbox Interrupts for Mailboxes Used
*
*****************************************************************************/
void CAN_Setup_Interrupts()
{
  *pCAN_MBIM1 = 0x00C0; // Enable Interrupts for Mailbox 7 and Mailbox 6
  *pCAN_MBIM2 = 0x0100; // Enable Interrupt for Mailbox 24
  ssync();
} // End CAN_Setup_Interrupts

/*****************************************************************************
*
* CAN_Enable – Writes Mailbox Direction and Enables Registers before issuing a CAN
*                Configuration Request and waiting for a CAN Configuration acknowledge
*                before continuing.
*
*****************************************************************************/
void CAN_Enable()
{
  // Set Mailbox Direction
  *pCAN_MD1 = CAN_RX_MB_LO; // No Low Mailboxes (MB 0-15) Are RX
  *pCAN_MD2 = CAN_TX_MB_LO; // Mailbox 24 Enabled For TX

  // Enable Mailboxes
  *pCAN_MC1 = CAN_RX_MB_LO; // Enables Mailbox 7 and Mailbox 6
  *pCAN_MC2 = CAN_TX_MB_HI; // Enables Mailbox 24
  ssync();

  *pCAN_CONTROL &= ~CCR; // Enable CAN Configuration Mode (Clear CCR)

while(*pCAN_STATUS & CCA); // Wait for CAN Configuration Acknowledge (CCA)

} // End CAN_Enable
```

CAN_ISR.c

```
/*****************************************************************************
*
* Device: ADSP-BF537
* Osc: SCLK = 120MHz
* File Name: "CAN_ISR.c"
* Author: Dominick O' Brien
* Date: 24-Jan-07
* Version 1.00
*
******************************************************************************/
#include "CAN_Test.h" // CAN Utilities

/******************************************************************************
*
* CAN_RCV_HANDLER – This ISR checks for the highest priority RX Mailbox with an
*                    active interrupt and clears it.
*
******************************************************************************/
EX_INTERRUPT_HANDLER(CAN_RCV_HANDLER)
{
 char  highMB; // Which CAN Registers Should Be Used (1 or 2)
 // short data type is 16 bits
 short mbim_status; // Temp Location for Interrupt Status
 short bit_pos = 0; // Offset Into MBxIF Registers

 mbim_status = *pCAN_MBRIF2;

 if (mbim_status == 0) // If High 16 MBoxes Have No Active IRQ
 {
   mbim_status = *pCAN_MBRIF1; // Check Low 16 MBoxes
   highMB = 0; // Clear High/Low* Indicator
 }

 else // Otherwise, Active High MBox IRQ Found
 {
   highMB = 1; // Set High/Low* Indicator
 }

 while (!(mbim_status & 0x8000)) // Scan Status Register For Highest MB IRQ
 {
   mbim_status <<= 1;
   bit_pos++; // bit_pos Contains Offset from MB31
 }

 if (highMB)
 {
   *pCAN_MBRIF2 = (1 << (15 - bit_pos));
 }

else // Low Mailbox Interrupt
 {
   if(bit_pos = = 0x8) // if Mailbox7 IRQ
   {
     if((*(pCAN_MB_DATA3(7)) <= 127))
     {
       clr_screen = 0; // Display BLACK
     }

     if((*(pCAN_MB_DATA3(7)) >= 128) && (*(pCAN_MB_DATA3(7)) <= 255))
```

```
                {
                  clr_screen = 1; // Display BLUE
                }

                if((*(pCAN_MB_DATA3(7)) >= 256) && (*(pCAN_MB_DATA3(7)) <= 383))
                {
                  clr_screen = 2; // Display RED
                }

                if((*(pCAN_MB_DATA3(7)) >= 384) && (*(pCAN_MB_DATA3(7)) <= 511))
                {
                  clr_screen = 3; // Display MAGENTA
                }

                if((*(pCAN_MB_DATA3(7)) >= 512) && (*(pCAN_MB_DATA3(7)) <= 639))
                {
                  clr_screen = 4; // Display GREEN
                }

                if((*(pCAN_MB_DATA3(7)) >= 640) && (*(pCAN_MB_DATA3(7)) <= 767))
                {
                  clr_screen = 5; // Display CYAN
                }

                if((*(pCAN_MB_DATA3(7)) >= 768) && (*(pCAN_MB_DATA3(7)) <= 895))
                {
                  clr_screen = 6; // Display YELLOW
                }

                if(*(pCAN_MB_DATA3(7)) >= 896)
                {
                  clr_screen = 7; // Display WHITE
                }
              } // end if Mailbox 7

              if(bit_pos = = 0x9) // if Mailbox 6 IRQ
              {
                // Place Received Commands Into CAN TX Mailbox
                *(pCAN_MB_DATA3(24)) = *(pCAN_MB_DATA3(6));
                *(pCAN_MB_DATA2(24)) = *(pCAN_MB_DATA2(6));
                *(pCAN_MB_DATA1(24)) = *(pCAN_MB_DATA1(6));
                *(pCAN_MB_DATA0(24)) = *(pCAN_MB_DATA0(6));

                // Issue CAN Transmit Request for Mailbox 24
                *pCAN_TRS2 = CAN_TX_MB_HI;
                ssync();
              } // end if Mailbox 6

              *pCAN_MBRIF1 = (1 << (15 - bit_pos)); // Write-1-to-Clear RX IRQ
            } // end Low Mailbox Interrupt
          } // end CAN_RCV_HANDLER

          /*****************************************************************************
          *
          * CAN_ XMT_HANDLER – This ISR checks for the highest priority TX Mailbox with an
          *                                     active interrupt and clears it.
```

```
*
******************************************************************************/
EX_INTERRUPT_HANDLER(CAN_XMT_HANDLER)
{
  char  highMB; // Which CAN Registers Should Be Used (1 or 2)
  short mbim_status; // Temp Location for Interrupt Status
  short bit_pos = 0;              // Offset Into MBxIF Registers

  mbim_status = *pCAN_MBTIF2; // Check High Mailboxes First
  if (mbim_status == 0) // If No High MB Interrupts
  {
    mbim_status = *pCAN_MBTIF1; // Check Low MB Interrupts
    highMB = 0; // Clear High/Low* Mailbox Indicator
  }

  else highMB = 1; // Set High/Low* Mailbox Indicator

  while (!(mbim_status & 0x8000)) // Find Highest Mailbox W/ Active IRQ
  {
    mbim_status <<= 1;
    bit_pos++;
  } // Interrupting Mailbox Found

  if (highMB) // Process High Mailbox IRQ
  {
    *pCAN_MBTIF2 = (1 << (15 - bit_pos));
  }

  else // Else, Process Low Mailbox IRQ
  {
    *pCAN_MBTIF1 = (1 << (15 - bit_pos));
  }
ssync();

} // End CAN_XMT_HANDLER
```

main.c

```
/******************************************************************************
*
* Device: ADSP-BF537
* Osc: SCLK = 120MHz
* File Name: "main.c"
* Author: Dominick O' Brien
* Date: 30-Jan-07
* Version 1.00
* Modified version of Analog Device's "ezkitutilities.c" found in VisualDSP++4.5
* References: "..\Blackfin\Examples\ADSP-BF533 EZ-Kit Lite\Drivers\PPI\Streaming.."
* Target Processor: ADSP-BF537
* Target Tools Revision: ADSP VisualDSP++ v4.5 (September 2006 Update)
*
******************************************************************************/
#include <services\services.h> // System Services
#include <drivers\adi_dev.h> // Device Manager Includes
#include <drivers\ppi\adi_ppi.h> // PPI Driver Includes
```

```
#include <defBF537.h> // Include all MMR's and bit defs
#include <drivers\encoder\adi_adv717x.h> // 7179 Device Driver Includes
#include "ezkitutilities.h" // EZ-Kit Utilities
#include "adi_itu656.h" // ITU656 Utilities
#include "CAN.h" // CAN Utilities

/*********************************************************************************
*
* ADSP-BF537 Switch Settings
*
*********************************************************************************
*
* SW1: ALL OFF
* SW2: ALL ON
* SW3: ALL OFF
* SW4: OFF, ON, OFF, ON
* SW5: ALL ON
* SW6: ALL ON
* SW7: ALL ON
* SW8: ON, ON, OFF, OFF, OFF, OFF
*
*********************************************************************************/

/*********************************************************************************
*
* A/V Extender Board Jumper Settings
*
*********************************************************************************
*
* JP1: NOT USED
* JP2: NOT USED
* JP3: JP3.5/7 & JP3.6/8 --> Processor's TWI
* JP4: JP4.1/2 & JP4.3/4 --> 27MHz A V extender card onboard clock to source PPI CLK
* JP5: JP5.3/4 --> Enables PPI0 to drive VID_OUT
* JP6: NOT USED
* JP7: NOT USED
* JP8: JP8.1/3 & JP8.2/4 --> Selects PPI0 as source
* JP8.7/8  --> Enables VID_OUT bus sync
* JP9: JP9.1/3 --> Connect AD7179 reset to reset flag
* JP10: NOT USED
*
*********************************************************************************/

/*********************************************************************************
*
*         External Connections
*
*********************************************************************************
*
* Connect a monitor to the A-V Extender card video-out connector. The video connectors are
* the bank of 6 RCA-style jacks on the A-V Extender card labelled as J7.
*
*   J7  +--------------------------------------------------+
*       |      O      O < Video out        O       | (white)
*
*       |      O      O                    O       | (red)
*        +------------------------------------------------+
*
*****************************************************************************/

/*****************************************************************************
*
* Enumerations and defines
*
*****************************************************************************/
#define ENCODER_PPI (0) // ADSP-BF537 has only 1 PPI called PPI0
#define NUM_BUFFERS (1) // Colour Change Rate = (NUM_BUFFERS/30)/second
// Colour Patterns
static u8 black[] = {0x80,0x10,0x80,0x10}; // Black pixel YCbCr format
static u8 blue[] = {0xF0,0x29,0x6E,0x29}; // Blue pixel YCbCr format
static u8 red[] = {0x5A,0x51,0xF0,0x51}; // Red pixel YCbCr format
static u8 magenta[] = {0xCA,0x6A,0xDE,0x6A}; // Magenta pixel YCbCr format
static u8 green[] = {0x36,0x91,0x22,0x91}; // Green pixel YCbCr format
static u8 cyan[] = {0xA6,0xAA,0x10,0xAA}; // Cyan pixel YCbCr format
static u8 yellow[] = {0x10,0xD2,0x92,0xD2}; // Yellow pixel YCbCr format
static u8 white[] = {0x80,0xEB,0x80,0xEB}; // White pixel YCbCr format

/*****************************************************************************
*
* Static data
*
*****************************************************************************/
// Create two areas in SDRAM that will each hold a 656 Frame
static u8 PingFrame[ADI_ITU656_NTSC_LINE_WIDTH * ADI_ITU656_NTSC_HEIGHT];
static u8 PongFrame[ADI_ITU656_NTSC_LINE_WIDTH * ADI_ITU656_NTSC_HEIGHT];

ADI_DEV_2D_BUFFER PingBuffer[NUM_BUFFERS]; // Create two buffer chains.
ADI_DEV_2D_BUFFER PongBuffer[NUM_BUFFERS];

// DMA Manager data (base memory + memory for 1 DMA channel)
static u8 DMAMgrData[ADI_DMA_BASE_MEMORY + (ADI_DMA_CHANNEL_MEMORY
* 1)];
// Deferred Callback Manager data (memory for 1 service plus 4 posted callbacks)
static u8 DCBMgrData[ADI_DCB_QUEUE_SIZE + (ADI_DCB_ENTRY_SIZE)*4];

// Device Manager data (base memory + memory for 3 devices)
// Memory for 3 devices is required because usage of a 717x device results in the usage of the
// PPI and SPI devices.
static u8 DevMgrData[ADI_DEV_BASE_MEMORY + (ADI_DEV_DEVICE_MEMORY *
3)];
ADI_DEV_DEVICE_HANDLE AD7179DriverHandle; // Handle to the ADV7179 Driver

/*****************************************************************************
*
* Global data
*
*****************************************************************************/
ADI_ITU656_FRAME_TYPE Frame; // ITU Frame Type
short clr_screen = 0;

/*****************************************************************************
*
```

```
* ExceptionHandler - An Exception error should never happen but just in case if one occurs all
*                        the LEDs will light up.
*
*******************************************************************************/
static ADI_INT_HANDLER(ExceptionHandler) // Exception Handler
{
  ezErrorCheck(1);
  return(ADI_INT_RESULT_PROCESSED);
}

/*******************************************************************************
*
* HWErrorHandler - A Hardware error should never happen but just in case if one occurs all
*                        the LEDs will light up.
*
*******************************************************************************/
static ADI_INT_HANDLER(HWErrorHandler) // Hardware Error Handler
{
  ezErrorCheck(1);
  return(ADI_INT_RESULT_PROCESSED);
}

/*******************************************************************************
*
* Callback - Callback occurs when the PPI has completed processing of the last buffer in the
*                Ping & Pong Buffer chains.
*
*******************************************************************************/
static void Callback(void *AppHandle,u32  Event,void *pArg)
{
  ADI_DEV_BUFFER *pBuffer; // Pointer to the Buffer that was processed

  switch (Event)
  {
    case ADI_DEV_EVENT_BUFFER_PROCESSED: // CASE (buffer processed)
      // When the buffer chain was created, the CallbackParameter value for the buffer that was
      // generating the callback was set to be the address of the first buffer in the chain.
      // So here in the callback that value is passed in as the pArg parameter.
      pBuffer = (ADI_DEV_2D_BUFFER *)pArg;

      switch(clr_screen) // Update data buffer with new colour
      {
        case 0: // Fill frame with BLACK colour
          adi_itu656_FrameFill (pBuffer->Data,Frame,black);
          break;

        case 1: // Fill frame with BLUE colour
          adi_itu656_FrameFill (pBuffer->Data,Frame,blue);
          break;

        case 2: // Fill frame with RED colour
          adi_itu656_FrameFill (pBuffer->Data,Frame,red);
          break;

        case 3: // Fill frame with MAGENTA colour
          adi_itu656_FrameFill (pBuffer->Data,Frame, magenta);
```

```
            break;

         case 4: // Fill frame with GREEN colour
            adi_itu656_FrameFill (pBuffer->Data,Frame,green);
            break;

        case 5: // Fill frames with CYAN colour
            adi_itu656_FrameFill (pBuffer->Data,Frame,cyan);
            break;

         case 6: // Fill frame with YELLOW colour
            adi_itu656_FrameFill (pBuffer->Data,Frame,yellow);
            break;

         default: // Fill frame with WHITE colour
            adi_itu656_FrameFill (pBuffer->Data,Frame,white);
            break;
      }

      break;

      // CASE (an error)
      case ADI_DEV_EVENT_DMA_ERROR_INTERRUPT:
      case ADI_PPI_EVENT_ERROR_INTERRUPT:
         ezTurnOnAllLEDs();// Turn on all LEDs and wait for help
         while (1) ;
  }
}

void main(void)
{
  // Table of PPI driver configuration values
  ADI_DEV_CMD_VALUE_PAIR ConfigurationTable [] =
  {
    {ADI_DEV_CMD_SET_DATAFLOW_METHOD,
     (void*)ADI_DEV_MODE_CHAINED_LOOPBACK},
     {ADI_PPI_CMD_SET_CONTROL_REG, (void *)0x0082},
     {ADI_PPI_CMD_SET_LINES_PER_FRAME_REG,
     (void*)ADI_ITU656_NTSC_HEIGHT},
     {ADI_DEV_CMD_SET_STREAMING, (void *)TRUE},
     {ADI_DEV_CMD_END, NULL},
  };

  ADI_DCB_HANDLE DCBManagerHandle; // Handle to the Callback Service Manager
  ADI_DMA_MANAGER_HANDLE DMAManagerHandle; // Handle to the DMA Manager
  ADI_DEV_MANAGER_HANDLE DeviceManagerHandle; // Handle to the Device Manager

  u32 ResponseCount; // Response Counter
  int i = 0; // Counter
  Frame = ADI_ITU656_NTSC_PR; // Frame Type

  ezInit(1); // Initialise the EZ-Kit
               // - Configure Async Memory
               // - Configure Power & SDRAM Parameters
               // - Configure Clock, CCLK = 600MHz, SCLK = 120MHz

  Init_CAN_Port(); // Initialise CAN Ports
  Init_CAN_Timing(); // Setup CAN Timing Parameters
  Init_CAN_Mailboxes(); // Initialise CAN Mailboxes' Registers
  CAN_Setup_Interrupts(); // Configure CAN Mailbox Interrupts
  CAN_Enable(); // Enable CAN

  ezTurnOffAllLEDs(); // Turn off all LEDs

  // Initialise the Interrupt Manager and hook the exception and hardware error interrupts
  ezErrorCheck(adi_int_Init(NULL, 0, &ResponseCount, NULL));
  ezErrorCheck(adi_int_CECHook(3, ExceptionHandler, NULL, FALSE));
  ezErrorCheck(adi_int_CECHook(5, HWErrorHandler, NULL, FALSE));

  // Initialise the Deferred Callback Manager and setup a queue
  ezErrorCheck(adi_dcb_Init(&DCBMgrData[0],
                            ADI_DCB_QUEUE_SIZE,
                            &ResponseCount,
                            NULL));

  ezErrorCheck(adi_dcb_Open(14,
                              &DCBMgrData[ADI_DCB_QUEUE_SIZE],
                              (ADI_DCB_ENTRY_SIZE)*4,
                              &ResponseCount,
                              &DCBManagerHandle));

  // Initialise the flag service, memory is not passed because callbacks are not being used
  ezErrorCheck(adi_flag_Init(NULL, 0, &ResponseCount, NULL));

  for (i = EZ_FIRST_LED; i < EZ_NUM_LEDS; i++) // Enable all LEDs
  {
     ezInitLED(i);
  }

  ezErrorCheck(adi_dma_Init(DMAMgrData, // Initialise the DMA Manager
               sizeof(DMAMgrData),
               &ResponseCount,
               &DMAManagerHandle,
               NULL));

  ezErrorCheck(adi_dev_Init(DevMgrData, // Initialise the Device Manager
                            sizeof(DevMgrData),
                            &ResponseCount,
                            &DeviceManagerHandle,
                            NULL));

  // Initialise the two frames and make them both BLACK in colour
  adi_itu656_FrameFormat (PingFrame, Frame);
  adi_itu656_FrameFormat (PongFrame, Frame);
  adi_itu656_FrameFill (PingFrame,ADI_ITU656_NTSC_PR,black);
  adi_itu656_FrameFill (PongFrame,ADI_ITU656_NTSC_PR,black);

  ezEnableVideoEncoder(); // Enable video encoder (7179)
  ezDelay(300); // Give the encoder time to sync

  // Open the AD7179 Driver for Output
  ezErrorCheck(adi_dev_Open(DeviceManagerHandle, // Handle controlling the Device
                        &ADIADV7179EntryPoint, // Address of Entry Point
                        ENCODER_PPI,  // Number identifying which Device is Opened
                        NULL, // No Client Handle
                        &AD7179DriverHandle, // Handle Address
                        ADI_DEV_DIRECTION_OUTBOUND, // Data Direction
                        DMAManagerHandle, // Handle to DMA Manager
                        DCBManagerHandle, // Handle to Callback Manager
                        Callback)); // Callback

// Set PPI Device Number
ezErrorCheck(adi_dev_Control(AD7179DriverHandle, // Handle identifying Device
                        // Command Identifier
                        ADI_ADV717x_CMD_SET_PPI_DEVICE_NUMBER,
                        (void*)0)); // PPI Device Number

// Open PPI Device
ezErrorCheck(adi_dev_Control(AD7179DriverHandle, // Handle identifying Device
                        ADI_ADV717x_CMD_SET_PPI_STATUS, // Command Identifier
                        // Address of Command Specific Parameter
                        (void*)ADI_ADV717x_PPI_OPEN));

// Create a buffer chain that points to the PingFrame. Each buffer points to the same PingFrame
// so the PingFrame will be displayed NUM_BUFFERS times. NUM_BUFFERS is sized to
// keep the display busy for 1 second. Place a callback on only the last buffer in the chain.
// Make the CallbackParameter (the value that gets passed to the callback function as the pArg
// parameter) point to the first buffer in the chain. This way, when the callback goes off, the
// callback function can requeue the whole chain if the loopback mode is off.

for (i = 0; i < NUM_BUFFERS; i++) // Populate the PingBuffer
{
PingBuffer[i].Data = PingFrame; // Point to PingFrame Data
PingBuffer[i].ElementWidth = 2;
PingBuffer[i].XCount = (ADI_ITU656_NTSC_LINE_WIDTH/2);
PingBuffer[i].XModify = 2;
PingBuffer[i].YCount = ADI_ITU656_NTSC_HEIGHT;
PingBuffer[i].YModify = 2;
PingBuffer[i].CallbackParameter = NULL;
PingBuffer[i].pNext = &PingBuffer[i + 1];
}

PingBuffer[NUM_BUFFERS - 1].CallbackParameter = &PingBuffer[0];
PingBuffer[NUM_BUFFERS - 1].pNext = NULL;

for (i = 0; i < NUM_BUFFERS; i++) // Populate the PongBuffer
{
PongBuffer[i].Data = PongFrame; // Point to PongFrame Data
PongBuffer[i].ElementWidth = 2;
PongBuffer[i].XCount = (ADI_ITU656_NTSC_LINE_WIDTH/2);
PongBuffer[i].XModify = 2;
PongBuffer[i].YCount = ADI_ITU656_NTSC_HEIGHT;
PongBuffer[i].YModify = 2;
PongBuffer[i].CallbackParameter = NULL;
PongBuffer[i].pNext = &PongBuffer[i + 1];
}

PongBuffer[NUM_BUFFERS - 1].CallbackParameter = &PongBuffer[0];
PongBuffer[NUM_BUFFERS - 1].pNext = NULL;

// Configure the AD7179 Dataflow Method
ezErrorCheck(adi_dev_Control(AD7179DriverHandle, // Handle identifying Device
          ADI_DEV_CMD_SET_DATAFLOW_METHOD, // Command Parameter
          (void *)ADI_DEV_MODE_CHAINED_LOOPBACK)); // Outbound Loopback

// Give the device the Ping and Pong buffer chains
ezErrorCheck(adi_dev_Write(AD7179DriverHandle, // Handle identifying Device
                         ADI_DEV_2D, // 2D Buffer
                         (ADI_DEV_BUFFER *)&PingBuffer)); // Point to PingBuffer

ezErrorCheck(adi_dev_Write(AD7179DriverHandle, // Handle identifying Device
                         ADI_DEV_2D, // 2D Buffer
                         (ADI_DEV_BUFFER *)&PongBuffer)); // Point to PongBuffer

Init_Interrupts(); // Assign Interrupt priorities for CAN RX/TX

// Enable data flow
ezErrorCheck(adi_dev_Control(AD7179DriverHandle, // Handle identifying Device
                           ADI_DEV_CMD_SET_DATAFLOW// Command Parameter
                           (void *)TRUE)); // Turn on Dataflow
while(1);
}
```

Appendix G - SAE 2007-01-1644

2007-01-1644

Design of a Digital Dash-Panel using a TFT LCD Panel and Blackfin Processor

Dominick O' Brien, Gavin Walsh
Advanced Automotive Electronic Control Group, Waterford Institute of Technology, Ireland

ABSTRACT

The following document discusses the benefits achieved from implementing a digital dash-panel within an automobile. It details the synthesis of the digital display system using the appropriate hardware components and software strategies. The justifications behind the selection of the Blackfin processor for controlling the system are outlined. In addition, a brief discussion of the safety aspects obtained from the digital dash-display is discussed.

INTRODUCTION

At present the automotive industry is seeing an escalating trend in the customer's desire to have new modern-day technologies integrated into the vehicle. This is due to the fact that drivers are increasingly expecting more from their automobiles than just the fundamental idea of travel. These new technologies are being included within the vehicle to satisfy the consumer's request for comfort, convenience, productivity and safety whilst driving. Digital displays are one of the new technological integration trends [1].

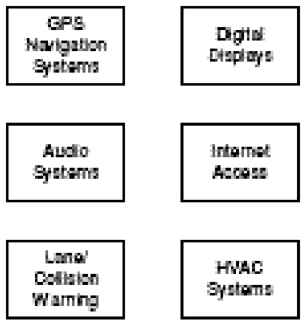


Figure 1: Modern Integrated Technologies

Digital display technologies, such as TFT LCD panels, are being increasingly used within the automobile as they possess a number of benefits, some of which have already been listed. Additional advantages can be obtained from these LCDs if they are used as a digital dash-display, thus replacing the aged analogue dash-panel. The reason behind this is that they can exhibit a larger volume of vehicle information than their analogue predecessors and can do so in a more aesthetically pleasing manner [1] [2].

The use of LCD panels can also lead to streamlined production as the same monitor can be placed into a number of different vehicle models, therefore reducing costs and increasing productivity. With these digital dash-panels manufacturers can introduce a degree of flexibility into the graphics displays. For example, a driver can select that the speedometer presents readings in either mph or km/h. In addition to this, style variations between automobile models can still be maintained by vehicle manufacturers by simply developing different software graphics/displays for each model [3].

Therefore it is proposed to design and synthesize a digital dash-system that graphically illustrates fundamental vehicle information such as speed, fuel level, engine temperature etc. The various vehicle data can be retrieved from a CAN network. A Blackfin processor will then be used to extract this vehicle information from CAN messages. The Blackfin processor would then interpret these messages and use the data to drive the graphics display upon the LCD panel.

SYSTEM DESIGN & SYNTHESIS

The design and synthesis of this digital dash-system can be divided into two main categories – hardware and software.

SYSTEM HARDWARE

The hardware for this particular digital dash-display contains a number of components as seen in Figure 2. First and foremost, in order to generate vehicle data numerous potentiometers are employed to mimic sensors measuring physical parameters such as engine temperature and speed. These potentiometers form part of a number of CAN nodes. The CAN nodes used within this system are implemented using PIC microcontrollers [4]. This is due to the fact that the PIC MCU is relatively straightforward to program, several of its *18F* series contain integrated CAN features and its unit cost is highly competitive.

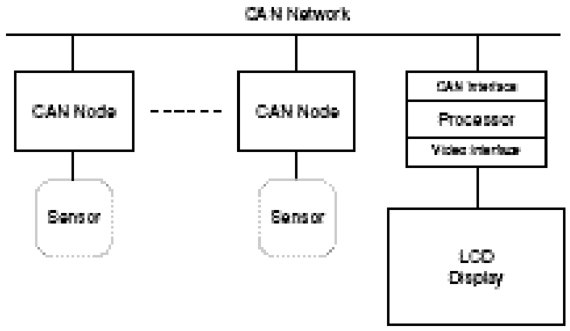


Figure 2: System's Hardware Components

<u>CAN Protocol Overview</u>

CAN is an asynchronous serial-bus protocol with NRZ bit coding designed to efficiently support distributed control systems. It was developed by Bosch in the late 1980's. CAN's popularity within the automotive industry and beyond is owed to its inherent robust ability to transfer data at relatively high speeds within harsh environments [5] [6]. For these reasons its employment within this system, to serve the purpose of transmitting simulated vehicle data, is justified.

<u>Selection of an Appropriate Processor</u>

The selection of a suitable processor within this system is crucial to its implementation as the processor itself will act as the system controller. The system controller within this application has to perform typical functions such as conditional operations, computations and data manipulations/conversions etc. It will also be required to contain CAN processing capabilities. However, with reference to Figure 2, it is implicit that a suitable system controller for this application must also possess EMP functionality. Therefore a number of processor options exist. The two main choices being the following:

1. MCU
2. DSP

MCU

A typical MCU could fulfill the role of the system controller with relative ease as it handles conditional operations and instruction jumps comfortably. What's more, a typical MCU's code is written using a high-level language like C or C++; therefore it is not a necessity to learn a particular MCU's assembler language [7].

MCUs can vary from 8-bit, to 16-bit or even to 32-bit. However, applications requiring EMP functionality usually rule out 8 or 16-bit processors. This is due to the fact that 8-bit processors do not contain the bandwidth and computational power required to process real-time signaling in this category of application. 16-bit processors do have an obvious advantage over their 8-bit counterparts (although they typically cost double), but on the other hand 16-bit processors usually require a complete new code development effort. As a result of this, the majority of developers would rather skip over 16-bit processors and move straight to a 32-bit controller since a new learning process would have to be embarked upon in any case. The leap over 16-bit processors is also justified due to the high level of on-chip peripheral integration associated with 32-bit processors. The use of a 32-bit system leads to improved code density and greater compiler support as a result of greater bit capacity within the data and address register files [7]. While a typical 32-bit MCU can sufficiently act as a system controller in a low-end EMP application, it does not possess the number-crunching capabilities required for more demanding EMP systems.

DSP

To address the average MCU's short-comings DSPs have materialized. DSPs are comprised of an architecture designed explicitly to perform as many MAC operations as possible in a single clock cycle [7]. Optimized DSP code is typically written using assembler algorithms. Thus, a detailed knowledge of a DSP's assembler language is required to develop highly optimized algorithms. In a typical EMP application a stand-alone DSP would not accomplish the role of a system controller adequately as it is primarily focused on completing mathematical algorithms.

System Controller Comprised of a Discrete MCU & DSP

After outlining the advantages and disadvantages of both MCUs and DSPs, one may be inclined to lean towards the concept of utilizing a discrete MCU and DSP to act together collectively as the system controller. Such a model would exploit the benefits of both the MCU and DSP. Additionally, the limitations of each discrete device would be counter-balanced by the other device. The DSP could concentrate on mathematically intensive algorithms, while the MCU could focus its attention upon control functionality [7].

Conversely, this particular conceptual design is not without flaws. If the system seen in Figure 3 was developed, the designer must ensure that adequate partitioning is in place to ensure that the computationally-intensive DSP doesn't steal resources required by the MCU. Also, another limitation would be cost. The cost of implementing a system containing a discrete MCU and DSP, along with their individual software suites, could outweigh the benefits achieved by such a system.

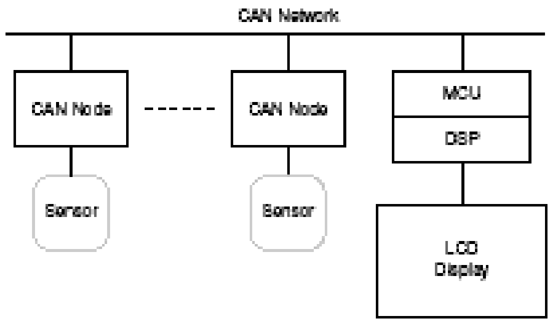


Figure 3: Systems Hardware Components with a Processor Comprised of Discrete MCU & DSP

Blackfin – A Convergent Processor

The Blackfin processor is an example of a convergent processor. It is an ideal system controller for EMP applications as it combines an MCU and DSP onto a single chip [7]. It obviously provides a cheaper and smaller solution than the concept discussed in the previous section. It must be noted that the Blackfin is not a DSP with additional microcontroller features; nor is it a MCU with advanced computational performance. The Blackfin simply merges the best from both worlds. This processor concurrently operates as a 16-bit DSP and 32-bit MCU. Consequently, it means that Blackfin processors contain adequate resources capable of handling both signal processing and control functions, or a combination of both depending on the particular application [8]. This convergent processor allows for the development of efficient coding as MCU routines can be written in a high-level language like C, while more intensive algorithms can be written using assembly language.

The Blackfin ADSP-BF537 processor, situated upon an EZ-Kit Lite development board, was chosen as the system controller in this application's synthesis as it contains adequate on-board features such as:

- CAN (32 Message Buffers)
- DMA
- PPI (Allowing ITU-R 656 Video Data Processing)

The ADSP-BF537 operates at speeds of up to 600MHz and contains 64MB of SDRAM with 4MB of Flash. From this it can be seen that it this is a powerful processing device, and the outlined features again prove its selection for this system's implementation is warranted.

A block diagram of the internal contents of the ADSP-BF537 can be seen in Figure 4 [9].

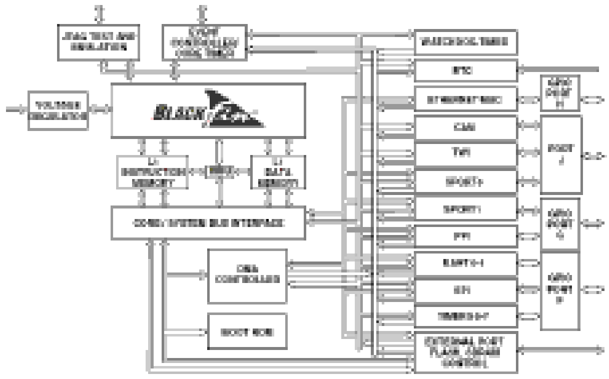


Figure 4: Blackfin ADSP-BF537 Functional Block Diagram

The ADSP-BF537 board also has an audio/visual daughter board which understandably facilities the generation of a video stream for display upon the digital panel. This A/V board contains both video encoders and decoders which can be interfaced with using the Blackfin's PPI. The hardware system employed for the application synthesis can be seen in Figure 5.

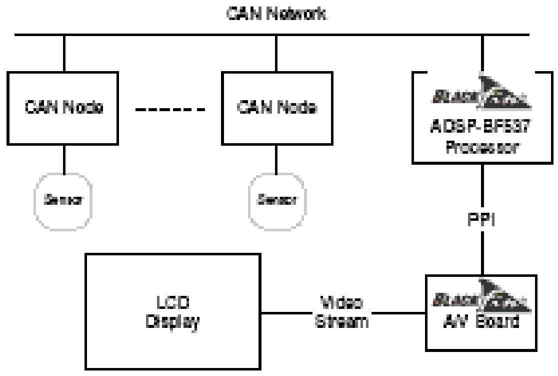


Figure 5: System Design

<u>ITU-R BT.601 & ITU-R BT.656 Digital Video Protocols</u>

The Blackfin's PPI port in conjunction with the A/V daughter board supports the ITU-R BT.656 protocol. BT.656, along with the BT.601 protocol, specifies methods for the digital coding and streaming of uncompressed video data. These protocols employ the YCbCr colour space for efficient use of bandwidth [10].

BT.656 and BT.601 can be used to implement PAL or NTSC streams [11]. In this discussion however, references to the PAL implementation will only be discussed.

BT.656 can be implemented in two formats – bit parallel and bit-serial mode. The bit-parallel mode is used in this system as the Blackfin processor doesn't support the bit-serial mode. This is advantageous as the bit-serial mode, although it uses only one channel, requires complex synchronization. The frame partitioning and data stream characteristics of ITU-R BT.656 can be seen in Figures 6 and 7 respectively [12].

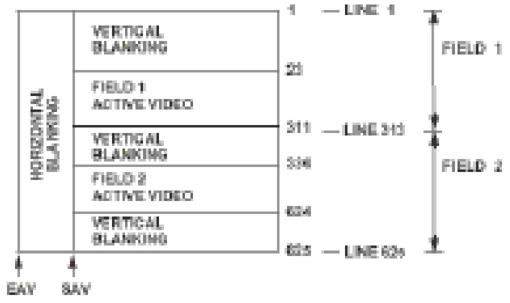


Figure 6: ITU-R BT.656 Frame Partitioning for a PAL System.

Within this protocol, an entire field/frame is composed of active video, horizontal blanking and vertical blanking. Each pixel component can be quantized to either 8 or 10-bits. PAL has 720 pixels of active video per line and its vertical resolution is typically 625 lines [10].

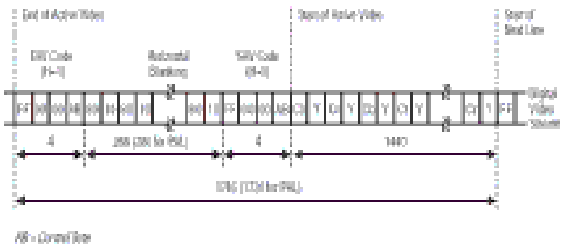


Figure 7: ITU-R BT.656 8-bit Parallel Data Stream for NTSC (PAL)

The SAV and EAV signals are used to indicate the beginning and end of video elements. An SAV occurs during a 1-to-0 transition of the horizontal bit, H, while an EAV occurs upon a 0-to-1 transition of H. As seen in Figure 7, there is a defined pre-amble of three bytes (0xFF, 0x00 and 0x00) for SAV and EAV. This is then followed by a control word containing the horizontal (H), vertical (V) and field (F) bits along with a number of error detection and correction bits [10]. The chrominance and luminance data follows the control byte as seen in Figure 7.

Thus, with a view to this systems realization, specific data received from within CAN messages are manipulated by the Blackfin and A/V board, and inserted into a BT.656 video stream for display upon the LCD in PAL format.

Interfacing PPI Port and Video Encoder

The ADSP-BF537 contains two PPI ports. Both ports are half-duplex and can accommodate 16-bits of data. The highest performance, in terms of PPI throughput, is achieved with 8-bit data since two 8-bit streams can be combined into a single 16-bit word [12].

It is worth pointing out that the PPI port does not explicitly support an ITU-R BT.656 output mode as it does not provide the proper partitioning and framing of preambles. However this can be implemented manually by performing a streaming operation from memory out through the PPI. Data and control codes can be stored in a memory buffer prior to video transmission. The DMA module can then transmit active video data into the buffer on a frame-by-frame basis [12].

The Blackfin's PPI port can interface with a number of devices including video encoders. In this system, one of the Blackfin's PPI ports transmits data to an ADV7179 video encoder chip situated upon the A/V daughter board. The ADV7179 video encoder can be configured to accept this ITU-R BT.656 video stream and convert it into a standard PAL signal upon one of its three DAC outputs [13]. The resulting PAL signal is then displayed upon the LCD panel.

SYSTEM SOFTWARE

The software created for use in this system synthesis can be summarized into two main sections:

1. CAN Implementation
2. Video Graphics Implementation.

CAN Implementation

The CAN network used in this design for the transfer of simulated vehicle data operates at a baud rate of 500kbaud/sec. Standard 11-bit CAN identifiers are employed throughout this system. Before delving further into the processes behind implementing this CAN network, a discussion of how the simulated vehicle data was obtained, in terms of software, is discussed.

10-bit Analog-to-Digital Conversion

Recall that earlier it was indicated that a number of potentiometers were used to mimic sensors measuring vehicle data. These potentiometers form part of a number of CAN nodes. A voltage reading from a particular potentiometer is converted into digitized data by a PIC MCU performing an ADC conversion. Each digitized reading can theoretically represent an automobile sensor measurement; for example speed or oil temperature. The PIC's resulting ADC conversion is a 10-bit binary representation of the applied analogue signal. However, it must be remembered that the PIC is an 8-bit processor.

Consequently the resulting 10-bit ADC result is stored across two 8-bit registers. Within the PIC, the ADC result can be configured to be either right or left-justified. When right-justified, the upper six bits of the most-significant ADC result register read '0'. Therefore the two MSBs of the 10-bit ADC conversion are stored in the upper ADC result register, while the remaining eight bits are stored in the lower ADC result register.

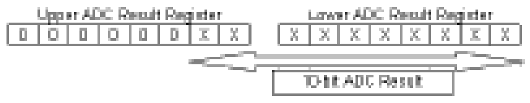


Figure 8: Right Justified 10-Bit ADC Result

As a result of this, the implemented C program must manipulate the ADC data appropriately in order to insert it into 8-bit CAN data bytes without the loss of any information. Taking the size of an integer in a particular compiler to be 16-bits, the 10-bit ADC result can be read back using a function and stored within an integer data variable. In other words, the 10-bit result is now contained within a variable capable of storing 16-bits.

The next step now involves extracting the data into two bytes. Therefore two characters, i.e. two 8-bit data variables, are declared. One character will store the lower eight bits of the conversion result, while another character will store the two upper bits of the ADC. The assigning of the value of the 16-bit data variable to the least significant character results in the eight MSB's of the 16-bit integer being discarded and the eight LSB's of the integer being stored in the least significant character.

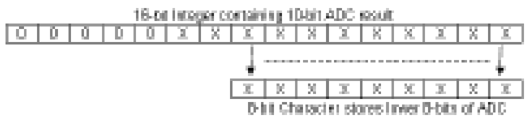

16-bit Integer containing 10-bit ADC result
8-bit Character stores lower 8-bits of ADC

Figure 9: Lower 8-bits of 16-bit Integer assigned to 8-bit Character

In order to assign the two MSB's of the ADC to the other character variable the 16-bit integer data variable must be manipulated by the bitshift-right operator. The contents of the integer variable are shifted eight places to the right. The resulting value is now assigned to the most significant character. Therefore the two MSB's of the ADC conversion are now the two LSB's of the character variable.

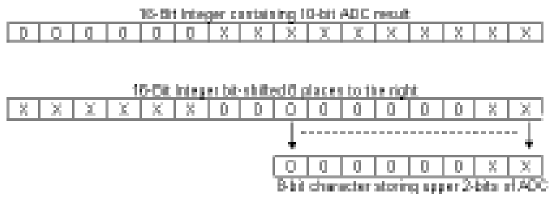

16-Bit Integer containing 10-bit ADC result
16-Bit Integer bit-shifted 8 places to the right
8-bit character storing upper 2-bits of ADC

Figure 10: Manipulation of 16-bit Integer to obtain the two LSBs of ADC result

Once the data from the potentiometers has been appropriately manipulated it can be inserted into CAN data byte registers. Subsequently, it can then be transmitted over the CAN network and used as simulated vehicle measurements for the purpose of illustration upon the digital dash-panel.

Blackfin CAN Interface

The ADSP-BF537 contains thirty-two CAN buffers/mailboxes. Eight of these mailboxes are for transmission; eight others are dedicated for message reception, while the remaining sixteen can be configured for either direction. Each of the thirty-two mailboxes contains eight 16-bit control and data registers [14]. These registers must be programmed accordingly in order to set-up the Blackfin's CAN interface.

The simulated vehicle data transmitted upon the CAN network needs to be received and interpreted correctly by the ADSP-BF537. Hence, numerous CAN buffers are configured appropriately with the aim of receiving messages with specific IDs. As the Blackfin is the system controller within this design it has to perform tasks in addition to CAN reception. Consequently the Blackfin's CAN interface is implemented using interrupts. This is in opposition to the polling methodology employed with the PIC. Polling is sufficient for use with the PIC in this system as the PIC's sole duty is to perform ADC readings upon the potentiometers and transmit the results over the CAN network.

Video Graphics Implementation

On initial power-up of the system various Blackfin components need to be initialized through software. For instance, the video encoder and SDRAM have to be configured, as does the PPI port and DMA. Once this has been carried out the system is ready to support the display of video data.

The CAN messages received by the Blackfin must be transferred from the various mailboxes into SDRAM in order to be converted into an ITU-R BT.656 video stream. Once the DMA and PPI port have been enabled the contents of SDRAM can be transferred to the PPI via DMA. The use of the DMA relieves the Blackfin of this transfer duty thus allowing it to perform additional tasks. As soon as the data has been transferred to the PPI port it is passed onto the video encoder. The ADV7179 video encoder converts the BT.656 video stream into PAL data and this PAL data is then displayed upon the LCD panel. A flow chart summarizing this process can be seen in Figure 11.

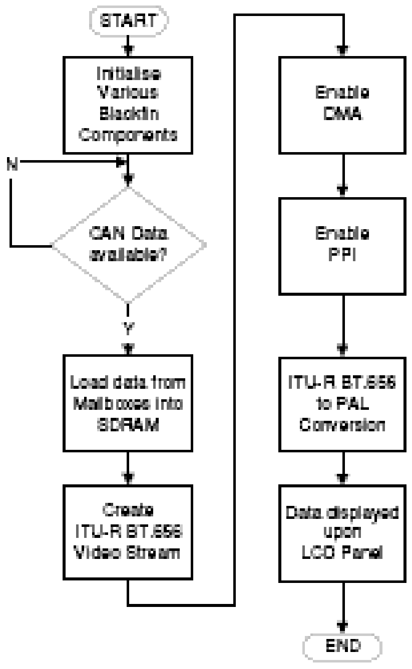

START
Initialise Various Blackfin Components
N
CAN Data available?
Y
Load data from Mailboxes into SDRAM
Create ITU-R BT.656 Video Stream
Enable DMA
Enable PPI
ITU-R BT.656 to PAL Conversion
Data displayed upon LCD Panel
END

Figure 11: Flow Chart of Video Graphics Implementation

ADDITIONAL BENEFITS OF DIGITAL DASH-PANELS

The implemented system discussed in this document leads to benefits that are not easily attainable with the use of analogue dash-displays.

An economies-of-scale can be achieved in employing a digital dash-panel. The same digital dash-system can be inserted into all of a particular manufacturer's vehicle models therefore reducing expenditure. Style variations between automobile models can still be maintained using differing levels of graphical elegance.

As the human eye is more responsive to colour resolution than spatial resolution important vehicle information displayed upon the digital dash-panel can be brought to the driver's attention more quickly [2]. Taking a fuel-level display for example; instead of displaying the typical gauge representative of fuel-levels, the system can display a green fuel tank for ample fuel levels, a blue tank for moderate amounts and a red tank for low levels [2]. The use of icons to represent vehicle data upon the LCD as opposed to using gauges results in additional information being displayed in a smaller area [2].

A safety aspect can also be realized with this digital dash-display system as telematic features such as GPS data could be displayed behind the steering wheel on the LCD as opposed to the centre-console display. Therefore drivers will have to spend less time searching for information as the data could be displayed directly in front of them. Further research can be carried out in this area to evaluate any improvements in driver concentration levels.

CONCLUSION

LCD panels are been used more frequently within the automobile. The construction of a digital dash-panel, implemented using a TFT LCD display, requires an adequate system controller to manage the necessary hardware and software duties. The Blackfin processor is suitable for this purpose as its combined MCU and DSP features are more than capable of adequately supporting the required tasks within this synthesis. The inherent EMP capabilities of the Blackfin allow for the relatively straight-forward interfacing to the LCD panel, while its CAN handling capabilities smoothly handle simulated vehicle data.

REFERENCES

1. Bielby, R., *Automotive FPGAs Offer Innovative Solutions to Automotive Display Challenges*, Xilinx Inc., http://www.techonline.com/community/ed_resource/tech_paper/39309, 2005.

2. Birman, V., Birman, P., *Driver Information Utilizing Flat Panel Display*, SAE Paper 2006-01-0951, 2006.

3. Kleiss, J. A., Enke, G., *Assessing Automotive Audio System Visual Appearance Attributes Using Empirical Methods*, SAE Paper 1999-01-1268, 1999.

4. www.microchip.com , 2006.

5. www.can-cia.org/can/ , 2006.

6. Richards, P., *Understanding Microchip's CAN Module Bit Timing*, Microchip Technology Inc., 2001.

7. Katz, D. J., Gentile, R., *Embedded Media Processing*. Pg17-26, 2006.

8. Katz, D.J., Lam, C., Gentile, R., *Blackfin Processor's Interface Simplifies LCD Connection in Portable Multimedia*, Analog Dialogue 39-01, www.analog.com/analogdialogue , 2005.

9. Analog Devices Inc., *Blackfin Embedded Processor – ADSP-BF536/ADSP-BF537 Preliminary Technical Data*, www.analog.com , 2006.

10. Katz, D. J., Gentile, R., *Embedded Media Processing*. Pg191-206, 2006.

11. Jack, K., *Video Demystified – 4th Edition*, 2004.

12. Analog Device Inc., *ADSP-BF537 Blackfin Processor Hardware Reference – Rev2.0*, Pg341-380, www.analog.com , December 2005.

13. Analog Device Inc., *ADV7174/ADV7179 Data Sheet*, 2004.

14. Analog Device Inc., *ADSP-BF537 Blackfin Processor Hardware Reference – Rev2.0*, Pg513-605, www.analog.com , December 2005.

15. Desroches, P., Trick, L., Nonnecke, B., *The Impact of Navigation Systems on the Perception Time of Young and Older Drivers*, SAE Paper 2006-01-0577, 2006.

CONTACT

Dominick O' Brien B.Sc (Hons), AMIEI

Email: dobrien@wit.ie

Gavin Walsh M.Sc, B.Tech (Hons), MIEEE, MIEE, AMIEI

Email: gwalsh@wit.ie

DEFINITIONS, ACRONYMS, ABBREVIATIONS

TFT: Thin Film Transistor

LCD: Liquid Crystal Display

CAN: Controller Area Network

MCU: Microcontroller

NRZ: Non-Return to Zero

EMP: Embedded Media Processing

DSP: Digital Signal Processor

MAC: Multiply-Accumulate Operations

DMA: Direct Memory Access

PPI: Parallel Peripheral Interface

A/V: Audio/Visual

PAL: Phase Alternation Line. This is a colour encoding system used in the broadcast television sector. It is predominantly used in Europe and South America.

NTSC: National Television System Committee. This is another broadcast system implemented to encode luminance and chrominance. NTSC is mostly used in North America and Asia.

HSYNC: This is the horizontal synchronization signal within a video steam. It indicates the start of active video on each row of a video frame.

VSYNC: This is the vertical synchronization signal within a video steam. It demarcates the start of a new video image.

SAV: Start of Active Video

EAV: End of Active Video

DAC: Digital-to-Analog Converter

ADC: Analog-to-Digital Conversion

MSB: Most Significant Bit

LSB: Least Significant Bit