



THE UNIVERSITY OF QUEENSLAND

**Vision Software  
for Humanoid Robot Soccer**

By

**Anthony Peters**

The School of Information Technology and  
Electrical Engineering  
The University of Queensland

Submitted for the degree of Bachelor of Engineering (Honours)  
in the division of Software Engineering

29 October 2003

26 October, 2001

Professor Simon Kaplan,  
Head of School,  
School of Information Technology and Electrical Engineering,  
University of Queensland,  
St Lucia QLD 4072.

Dear Professor Kaplan,

In accordance with the requirements of the degree of Bachelor of Engineering in the division of Software Engineering, I present the following thesis entitled “Vision Software for Humanoid Robot Soccer”. This thesis project was conducted under the supervision of Dr Gordon Wyeth.

I declare that the work submitted in this thesis is my own, except as acknowledged in the text, and has not been previously submitted for a degree at the University of Queensland or any other institution.

Yours sincerely,

Anthony Peters

## **Acknowledgements**

God, for helping me get to a stage in my life that I can complete a thesis.

Gordon Wyeth for his supervision and unfaltering vision and direction.

Damien Kee for always having the right advice at the right time.

Mark Chang for his endless and timely advice on the vision hardware.

David Prasser for his assistance with the software for image processing.

John Nguyen and Ravi Nath for constantly keeping things in perspective by reminding me how much further behind they were with their theses.

Lisa Bright, for being my motivation and inspiration, and providing the guiding light I needed when things were looking their bleakest.

My family, whom I have no doubt are tired of hearing about this thesis.

## **Abstract**

This thesis describes the continuation of work on the vision system for The University of Queensland's Humanoid Robotics Project, GuRoo.

The work in this project saw the correction and adjustment of several issues which had not been considered in the test environment. These problems had the potential to disrupt the speed and reliability of the system when used in the target environment. The project also incorporated the cross-development of the software applications which accompany the vision system for Microsoft Windows. The project concluded with the integration of the vision system with GuRoo.

The improvements made to the system allow the streaming of higher resolution images with better colour definition and focus. The focus of the images is corrected through an adjustment of the spacing between the lens and imaging device. The saturating effects of ambient light on the acquired images is eliminated through the prevention of light penetrating the camera housing. This is achieved through the redesign of the camera housing.

The integration of the vision system with GuRoo is achieved, while the project still awaits the arrival of the new head. The attachment of the system is demonstrated through the use of the temporary head, which incorporates the eIMU.

The combination of the software cross-development for Microsoft Windows, combined with the correction of the focal and colour definition issues, will allow more rapid developments to be made to GuRoo's vision system.

# Contents

<b>1 Introduction</b>	<b>1</b>
1.1 GuRoo and the RoboCup Tournament.....	1
1.2 Vision Requirements for RoboCup.....	2
1.3 Thesis Outline.....	4
<b>2 Literature Review</b>	<b>5</b>
2.1 GuRoo Vision Hardware.....	5
2.1.1 Camera Board.....	6
2.1.2 SH4 Vision Board.....	6
2.2 GuRoo Vision Software.....	8
2.2.1 YUV Colour Space.....	8
2.2.2 Lookup Tables.....	10
2.2.3 Morphological Erosion.....	12
2.2.4 Blob Detection.....	13
2.3 Image Analysis and Object Detection.....	14
2.4 Hardware Programming Applications.....	16
2.5 Vision Debug Applications.....	16
<b>3 Problem Specification</b>	<b>17</b>
3.1 Software Redevelopment for Win32.....	18
3.2 Hardware and Software Integration.....	18
3.3 Vision System Integration with GuRoo.....	19

<b>4 Software Developments</b>	<b>20</b>
4.1 Win32 Software.....	21
4.1.1 SH4 Software.....	22
4.1.2 Debugging Software.....	23
4.2 SH4 Improvements.....	25
4.2.1 Resolution Improvements.....	25
4.2.2 Speed Improvements.....	27
<b>5 System Integration</b>	<b>29</b>
5.1 Focus Correction.....	29
5.2 Saturation Suppression.....	32
5.3 Vision System Integration with GuRoo.....	34
<b>6 Project Evaluation and Future Work</b>	<b>36</b>
6.1 Flash for SH4 Vision Board.....	37
6.2 Detection of Additional Objects.....	37
6.3 Redesign of SH4 Vision Board.....	38
<b>7 Conclusion</b>	<b>39</b>
<b>Bibliography</b>	<b>40</b>
<b>A Hardware User Guide</b>	<b>42</b>
<b>B Software User Guide</b>	<b>64</b>
<b>C Additional Code</b>	<b>84</b>
<b>D Head Redesign</b>	<b>105</b>
<b>E Camera Housing Redesign</b>	<b>106</b>

# List of Figures

Figure 1 - Current Vision Hardware .....	5
Figure 2 - Camera Board .....	6
Figure 3 - Hardware Block Diagram .....	7
Figure 4 - CMYK Colour Space .....	9
Figure 5 - YUV Colour Space .....	9
Figure 6 - Normal 2D Lookup Table .....	10
Figure 7 - Bruce's Lookup Table .....	11
Figure 8 - The erosion process.....	12
Figure 9 - Run Length Encoding and Grouping .....	14
Figure 10 - Frame Status Protocol .....	24
Figure 11 - Initial Resolution (256 x 64) vs Improved Resolution (128 x 128).....	26
Figure 12 – Height-to-width comparison - 1:1 vs 2:1 .....	27
Figure 13 - Focus - Old Spacers .....	30
Figure 14 - 1st and 2nd Principal Points.....	31
Figure 15 - Flange Distance.....	31
Figure 16 - Saturated Image.....	32
Figure 17 - Old Camera Housing.....	33
Figure 18 - "Make-do" Head Solution .....	34
Figure 19 - GuRoo's Current Head .....	105
Figure 20 - GuRoo's Redesigned Head.....	105
Figure 21 - New Camera Housing .....	106
Figure 22 - Wireframe of New Camera Housing .....	106

# Chapter 1

## Introduction

### 1.1 GuRoo and the RoboCup Tournament

The University of Queensland's humanoid project, named GuRoo, is now 3 years old. Work on the project has seen 24 students undertake undergraduate theses on topics such as distributed motion controllers, gait generation and control algorithms and vision system design, all targeted at a humanoid implementation.

The purpose of the GuRoo Humanoid project is well defined. It has a single purpose; to compete in the RoboCup Robot Soccer Tournament [1]. The RoboCup Competition is an international tournament where teams of robotics researchers come together to showcase the ground-breaking achievements of their research. The competition is held annually, and exhibits the work of several participating universities and research organisations.

The RoboCup Competition offers several classes of events for its participants to compete in. These events include a Dancing Competition, a Rescue Competition, and the Soccer Competition. There is also a Freestyle Competition, where teams can demonstrate any other achievements which are not applicable to the afore-mentioned categories. The competition of interest to the GuRoo project, and this thesis, is the Soccer Competition.

The Soccer Competition at the RoboCup Tournament has a set of rules which clearly define the operating conditions and environment which participating robots must comply with. These



rules address all issues regarding competition, ranging from the structural requirements for a robot to compete, through to the colour of each object within the environment that participants need to be concerned with.

The RoboCup Soccer Competition exposes three categories of competition for the soccer tournament. These are; small sized league, medium sized league, and humanoid league. Each variation of the competition has slightly differing rules to the others, with the main differences appearing in the structural requirements. [2]

## 1.2 Vision Requirements for RoboCup

The humanoid league of the RoboCup Soccer Competition is still very much in its infancy. As a consequence of this, the rules are highly inconclusive with regards to the details of the environment the humanoids will be operating in. This year has, however, seen the inclusion of such details, in draft form, in the humanoid league rules.

Given the importance of these details to the design of a vision system for a participant in this league, certain assumptions have been made that allow the vision system to take form. These assumptions are primarily the set of objects the vision system must be able to detect, along with their designated colours. The objects, along with their stipulated colours are shown below.

OBJECT	COLOUR
Ball	Orange
Goals	1 x Blue, 1 x Yellow
Playing Field	Green
Field Lines	White
Obstacles	Black

**Table 1 - Assumed Object Set with Colours**

In addition to the objects the vision system must be able to detect, there are other environmental factors which play a role in the vision system's design. One of the primary factors is the dynamic nature of the operating conditions for the humanoid. At any given time, the player must be able to detect and interact with at least one opponent, the ball, and both the goals, and at the same time be concerned with its own location within the field. Given the dynamic nature of this environment, the humanoid's vision system must maintain a frame rate that will allow it to deal with the rapid movement of any of the objects it will be tracking.

There are two factors which impact the frame rate of a vision system. The first is the image acquisition time, and the second is the image processing time. The image acquisition time is determined by the image capture hardware. Currently, technology is available that can capture images at frame rates far greater than that of the human eye. This makes the frame rate of a vision system almost entirely dependant on the image processing hardware and algorithms.

The image processing hardware and algorithms must be designed with speed and accuracy in mind. Speed, however, is the more important attribute. The reason for this is that if a high enough frame rate is maintained, errors in a single frame will be quickly compensated for by the arrival of the following frame. The longer a frame takes to process, the less relevant that frame becomes.

A vision system designed for use in the RoboCup Humanoid League should be designed with the goal of offering real-time vision. However, due to the processing power and algorithm optimisation required to achieve this, such a requirement need not be met for a vision system to be of use to the humanoid.

## **1.3 Thesis Outline**

**Introduction:** A brief outline of the history and purpose of The University of Queensland's Humanoid project. It also outlines the requirements imposed on the humanoid's vision system by the RoboCup initiative.

**Literature Review:** A look at the current state of GuRoo's vision system. Both hardware and software elements are discussed.

**Problem Specification:** A detailed description of the requirements of GuRoo's vision system, along with what the goals of this thesis contribute towards achieving these requirements.

**Software Developments:** A description and rationale behind the software developments made in this thesis, including both the re-writing of the current software for the Win32 API, and the SH4.

**System Integration:** An insight into the steps taken to integrate the current hardware and software elements of the vision system, and the vision system's integration with GuRoo.

**Future Work:** Suggestions and discussion of what steps to take next to bring the vision system closer to achieving its purpose.

**Conclusion:** Results and improvements of the project are summarised.

# Chapter 2

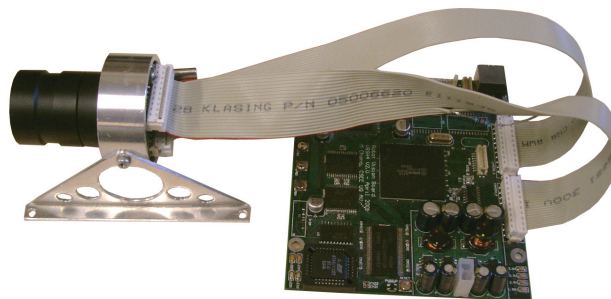
## Literature Review

### 2.1 GuRoo Vision Hardware

#### 2.1.1 Current Vision Hardware

Previous work on the development of a vision system for the GuRoo humanoid includes the selection of a camera, processor, and memory components, and the implementation of their respective electrical interfaces.

The vision hardware for the GuRoo project consists heavily of the interaction between two sub-systems. These sub-systems are the image acquisition system, and the image processing system. These systems are implemented independently in the form of the camera board and the SH4 vision board. The camera board is solely responsible for powering the camera chip, and controlling data transfer between the camera chip and the SH4 vision board. The SH4 vision board is a custom built board that contains various processing components required for the acquisition and analysis of data obtained from the camera board.



**Figure 1 - Current Vision Hardware**

### 2.1.1 Camera Board

The camera chip used is an OmniVision OV7620. This camera chip was chosen based on several of its features which make it ideal for local vision systems. These features include a frame rate of up to 60 Hz, a variable bit data output, RGB and YUV image output, and on-chip windowing. The default resolution is 640x480, with a maximum resolution of 664x492.

The other important component of the image acquisition system is the lens. The lens chosen for GuRoo's vision system is an AVENIR SSV0358. This lens is a vari-focal lens, with a focal length of 3.5 – 8.0mm, and an aperture of 1.4. One of the more important features of the selected lens is its 80 degree horizontal field of view. This allows a much wider area to be seen in a single image than other 1/3" image format lenses.

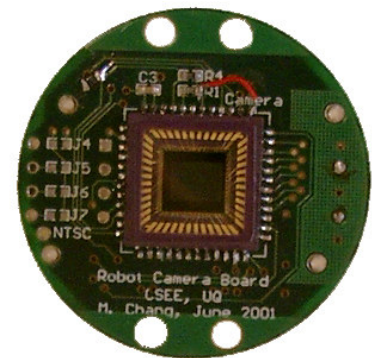


Figure 2 - Camera Board

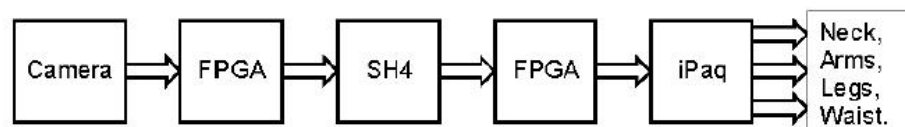
### 2.1.2 SH4 Vision Board

The SH4 vision board was designed by Mark Chang for use in The University of Queensland's ViperRoos robotic soccer team. The board has been designed for the various vision requirements for the ViperRoos, which are similar in almost every aspect to that of those required by GuRoo. The components featuring on the board include an Hitachi SH4 microprocessor, which is used for the primary image processing functions, a Xilinx Spartan II FPGA for secondary processing, 512KB SRAM, and 16MB SDRAM. [4]

The Hitachi SH4 is a 32-bit RISC processor belonging to the Hitachi SH7750 Series of microprocessors [3]. It has an operating frequency of 200MHz, with a performance rating of 360 MIPS. The instruction set uses a fixed length instruction format for improved performance. The processor also offers an instruction cache and an operand cache to further improve the processor's performance.

A 32-bit data bus is used for the transfer of image information and control signals between the SH4 vision board and the camera board. This is convenient due to the SH4's internal data bus also being 32-bit.

The flow of information in the current GuRoo vision hardware is demonstrated in the block diagram shown in Figure 3. (Prasser, 2001:16)



**Figure 3 - Hardware Block Diagram**

The SH4 vision board also supplies facilities for hardware and software debugging. These facilities take various forms, including status LEDs and serial and Universal Serial Bus (USB) connections to a PC. Currently, however, the electronic components required for the USB connection have not been added to the board, leaving the serial interface the only connection available.

There are several status LEDs found on the SH4 board in various locations. These LEDs are used to show the current input voltage, and the state of each of the primary elements of the board, including the CPU and FPGA. The most important set of status LEDs on the vision board are the “boot LEDs”. The boot LEDs comprise of four LEDs, with different combinations showing the current state of the system. During the boot stage, the boot LEDs perform the function of displaying the current phase of the boot process. In the event of a problem during the boot sequence, these LEDs allow the identification of the problematic stage.

The boot LEDs are also used during program execution to communicate such information as the current frame rate and confirmation that the program hasn't crashed.

## **2.2 GuRoo Vision Software**

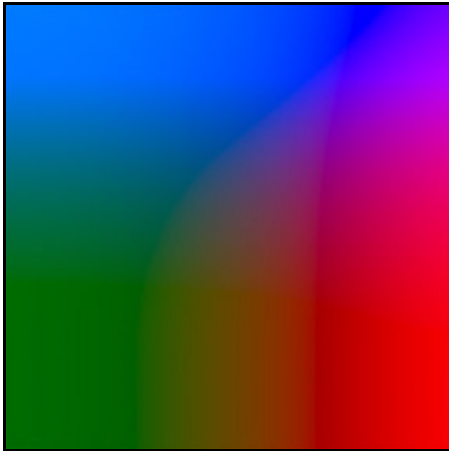
The software pipeline required for robotic vision is quite well established [4]. This pipeline commences with a segmentation process, where the acquired image is reduced into different regions containing the objects of interest. This resulting set of regions lends itself to a more complex set of functions which can then be used to reduce noise in the image, along with the classification of object types and the determination of the relevance of each of the objects found within the image.

Work on the software for GuRoo's vision system has seen the development of algorithms for the segmentation, erosion and classification of objects within images which closely model those which will be captured using the vision system.

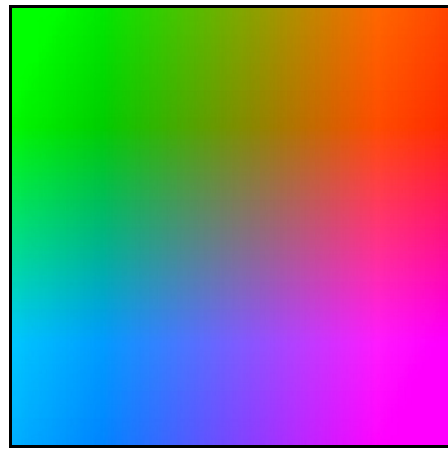
### **2.2.1 YUV Colour Space**

There are various ways of representing the colours we see. These methods of representing colours are defined by the building blocks used to construct all other colours in the model, or the primary colours.

The most common method is known as RGB. This model uses red, green and blue as the primary colours, and is the method used in such visual devices as televisions and computer monitors. Another popular method is titled CMYK. This method uses cyan, magenta, yellow and black as its primary colours. It is heavily used in printing processes, where it makes sense to introduce black as a primary colour.



**Figure 4 - CMYK Colour Space**



**Figure 5 - YUV Colour Space**

There are other models, such as HSI (Hue, Saturation, Brightness), but the model that is of particular interest for vision systems is YUV. This model is the luminance-bandwidth-chrominance model. The Y component represents the brightness of the image, while the U and V components represent the blue and red chrominance respectively. This method is more ideal for robotic vision applications due to its ability to mask the variance in colour due to ambient lighting conditions. A change in ambient light will, theoretically, only affect the Y component. All other colours are then represented as a particular mixture of red and blue [5].

One of the selection criteria for a camera to be used in GuRoo's vision system was the ability to output images in YUV format. This reduces the processing required to detect objects within the image.



2.2.2 Lookup Tables

An image represented in the YUV colour space is only of practical value for the elimination of lighting effects. Once this image has been acquired, the remainder of the process involves the preparation of the image for object detection.

The RoboCup Rules [2] clearly define the specifications for the environment in which GuRoos must operate. Fortunately, each of the relevant objects in the environment are colour coded.

The next step in the image processing pipeline is the conversion of a given pixel from the YUV colour space to the RGB colour space to allow a direct matching of a colour in the image to the colour of an object. The most basic method involves a simple 2-dimensional lookup table which uses the U and V components to map the x and y axes. An example of this is shown in Wong (2002:19).

U7	G	G						
U6	G	G						
U5	G				B	B		
U4								R
U3			Y				R	R
U2		Y	Y	Y		R	R	R
U1		Y	Y			R	R	R
U0	Y	Y			R	R	R	R
	V0	V1	V2	V3	V4	V5	V6	V7

Figure 6 - Normal 2D Lookup Table (Wong, 2002:19)

A more complex, but more memory efficient method of colour lookup is the use of Bruce’s Lookup Table [6]. This method uses a series of n-bit vectors, where n is the number of colours, for each U and V value. The resulting two vectors (one for U and one for V) are bit

ANDed together. The result will be an n-bit vector, which will contain all 0's apart from a single 1. The position of this 1 dictates the colour of the pixel.

R	1	1	1	1	1	0	0	0
B	0	0	0	0	0	1	0	0
Y	1	1	1	1	0	0	0	0
G	0	0	0	0	0	1	1	1
	U0	U1	U2	U3	U4	U5	U6	U7

R	0	0	0	0	1	1	1	1
B	0	0	0	0	1	1	0	0
Y	1	1	1	1	0	0	0	0
G	1	1	0	0	0	0	0	0
	V0	V1	V2	V3	V4	V5	V6	V7

Figure 7 - Bruce's Lookup Table (Wong, 2002:20)

Upon completion of the colour lookup phase, a colour coded image is left that contains areas of all the colours used in the colour lookup phase. The purpose of the colour coded image is two-fold. The first reason is to reduce the amount of information that needs to be processed. The second, and more important, reason is to reduce the image to the colours that concern the object detection algorithms. These colours are simply yellow, blue, orange, green, white and black. All colours in the RGB colour space of the real world are reduced to these six colours. This result allows the easy detection of objects within the image [5].

There is, however, one problem with a colour coded image. This is the introduction of noise. The accuracy of the colour lookup stage is limited by its discrete nature. The introduction of noise into the image could result in major anomalies in the detection of objects. For example, a spot on the lens may result in a section of the ball appearing black, preventing the object detection algorithm from detecting any round objects in the image.

### 2.2.3 Morphological Erosion

The solution to the noise problem previously described lies in a technique called morphological erosion [7]. Morphological erosion is a mathematical technique commonly used in the processing of images. The technique treats an image as a set of pixels which make up a two dimensional coordinate space.

There are two basic operations used in the process. These are erosion and dilation. Erosion causes a group of pixels to become smaller, while dilation has the opposite effect. Both of these functions have two parameters, a group of pixels, and a structuring element. The structuring element dictates the effect the operation has on the group of pixels.

Morphology theory also exposes two complex operations. These are opening and closing. An opening is an erosion followed by a dilation, and a closing is effectively the opposite. The erosion and dilation operations used in an opening or a closing use the same structuring element.

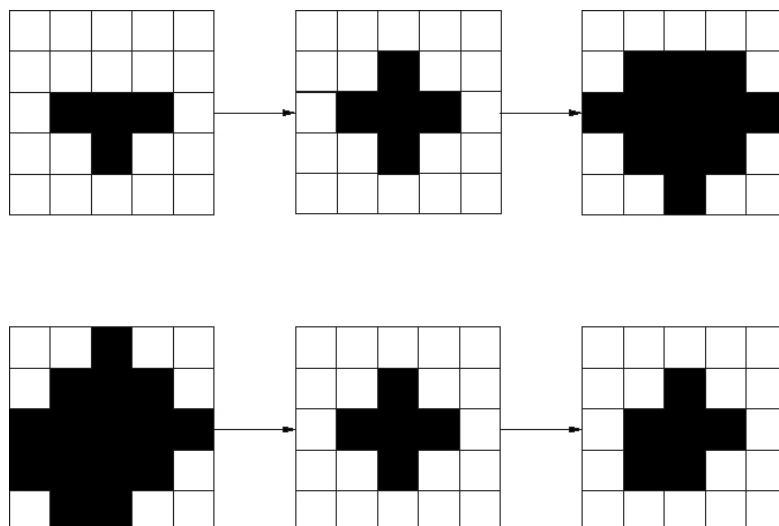


Figure 8 - The erosion process

The process is demonstrated in Figure 8. In the figure, the area to be eroded is the leftmost array, with the structuring element represented in the centre array. The result of the erosion process is shown in the right-most array.

The currently implemented morphological operation for use in GuRoo's vision system is an erosion with a 1 x 3 structuring element. This is relatively successful, particularly in removing small amounts of red noise [5].

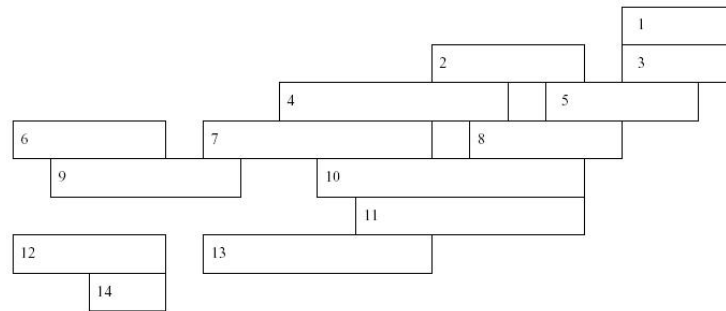
#### **2.2.4 Blob Detection**

The first step to be taken is the recognition of areas within the image that are the same colour. Given the constraints on the operating environment, an area of a specific colour in our image is certain to be a given object.

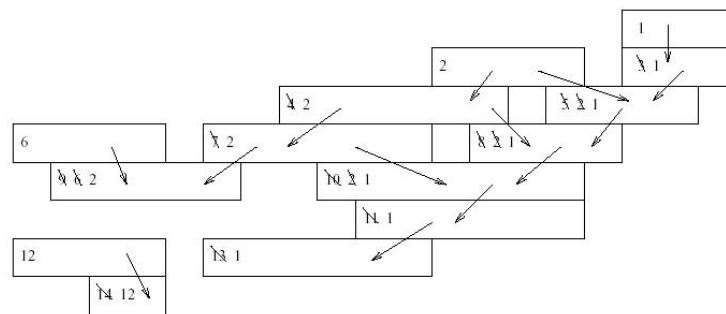
The process of object detection begins with the location of "runs" of the same colour on a single line [5]. Each run of a specific colour is given a unique "run number" and is referred to as a run element. The image is analysed, starting from the top left, with the coordinates of the start and end of each run element stored to memory.

The process of blob detection (grouping) is the final phase of the image processing pipeline before object detection can commence. Blob detection is the process of grouping adjacent run elements of the same colour.

Run elements are analysed in a top-down approach, with a comparison of the overlapping run elements on the rows above and below the current row. If the run elements are the same colour, they are added to a blob. A blob is a group of run elements four-connected which are of the same colour [5].



(a) Run length encoded data before grouping.



(b) Propagation of tag numbers through run elements.

**Figure 9 - Run Length Encoding and Grouping (Prasser, 2001:30)**

Upon the completion of the grouping process, the image is ready for analysis to determine the location of objects within the image.

## 2.3 Image Analysis and Object Detection

The historical background of digital image processing dominates the field of computer vision with regards to object recognition [8]. Digital image processing originated for use in static remote sensing, where the intensive evaluation of single images prevails.

This simple principle of evaluating single images, particularly when combined with state information in image sequences, can produce very useful information for object detection and tracking. For example, using a single image, it is possible to locate any given object within it, however with a substantial processing overhead. Knowing the location of the object in the image, we can now use this information to reduce the search space in the following image, if we are required to locate the same object again. This method can be used to reduce the processing required to locate objects within an image.

With regards to actually locating objects within an image, it is imperative that a prior knowledge of the search objects be known. This helps the detection algorithm recognise an object based on specific properties which can be used to distinguish them.

The basic idea is to extract features from the image [4]. These features are simple properties such as area, perimeter and colour. Each image segment is analysed by assigning the values of its *feature vector*. The resulting feature vector is then matched to a predefined set of feature vectors, which correspond to the search objects. The identity of the search object can then be discovered.

The greatest problem with this method is obtaining enough information to ensure the correct placement of the decision boundaries [4]. For example, if the feature vectors of multiple search objects are similar within a certain threshold, a small error in the classification of an image segment may result in identical feature vectors for quite distinct objects in the image. Other methods may not expose this possible complication. The solution lies in the acquisition of a suitable quantity of information about an image segment and the careful selection of feature elements.

One of the main problems presenting itself with machine vision is the presentation of 2D projections of a 3D reality. The problem lies, as mentioned above, with feature selection. It is exceedingly difficult to derive feature measurements which use object properties which are invariant to the relative image positioning, in addition to distance and lighting factors.

## **2.4 Hardware Programming Applications**

The software used to control the hardware and its appropriate functions are written in C. A serial uploading program called ploader has been written by Mark Chang to accompany his SH4 vision board. The program communicates with the SH4 via the available SCI interface. Details of how to use ploader (both the Linux and Win32 versions) can be found in Appendix A.

In order to compile the software required for the SH4, the GNU SH cross compiler is used. This is a cross compiler built on the GCC framework, and requires the Linux operating environment to function. Details on how to build such a cross development environment can be found in an article titled “Building A Cross Development Environment Targeting SH4 Systems”. [9]

## **2.5 Vision Debug Applications**

Apart from the hardware debugging methods previously mentioned, an application has been developed, VDebug, which displays on a PC what the vision system is currently “seeing”. This program communicates directly with the SH4 vision board, and requests the relevant information through the SCI interface.

VDebug is written according to the ANSI C standard, and makes extensive use of the OpenGL graphics libraries for display functionality. The program displays the relevant information for the current frame in three windows. The first window is used to display the image, the second shows a YUV Map of the image, and the final window shows an RGB Histogram of the image. Each of these windows are rendered as RGB textures in OpenGL and mapped to their appropriate windows.

# Chapter 3

## Problem Specification

Independently, both the hardware and software subsystems have been implemented and their functionality verified experimentally. However, no attempt has been made to integrate these systems with GuRoo, and demonstrate their functionality in conjunction with other GuRoo behaviours.

The scope of this project involved the collaboration of the existing subsystems and verifying their ability to function cooperatively, along with the integration of the vision system with GuRoo.

Before this integration could be attempted, the various debugging applications used in conjunction with the vision system, which were initially written for a Linux execution environment, needed to be ported to the Microsoft Windows environment.

In addition to the redevelopment of the debugging applications, various add-ons required development for these applications which allowed the visual analysis and verification of the image processing functions.



### **3.1 Software Redevelopment for Win32**

The existing software for GuRoo's vision system, including both the embedded software and system debugging software, had been developed exclusively for the Linux operating environment. This enforced knowledge of the Linux operating system as a prerequisite for any work on this vision system. This restriction has in the past caused setbacks and interruptions to development of this vision system. This project was no exception.

This project goal was added after initial problems with the procurement of a Linux system with the required software for use in this project. These hindrances to progress prompted the addition of the goal of redeveloping the existing software for use under a Microsoft Windows environment, in an attempt to reduce the likelihood of such obstacles manifesting themselves in the future.

The primary objective when redeveloping the software for use under Windows was to allow a seamless transition between Windows and Linux. This allows changes to be made to the functionality of the embedded software and/or the debugging applications using the operating environment of choice without inflicting the need to make the changes again under the alternate operating environment.

### **3.2 Hardware and Software Integration**

The hardware used for GuRoo's vision system is currently being used to capture frames using the image capture subsystem, and then using the image processing subsystem to transmit these frames to the PC as requested. This functionality is well below that of the intended use of the vision system, and makes no use of the current image processing algorithms which have been developed for use by the system.

The image processing algorithms have been written and tested on the SH4 processor; however, these algorithms have only been tested on static images, which have been uploaded to the SH4 for testing purposes. No real-time analysis of images has been attempted.

This project objective required the integration of the image processing algorithms with the image processing subsystem to allow the real-time analysis of frames taken from the image capture subsystem.

As a consequence of this project requirement, further development of additional functionality in the debugging applications was required to allow the debugging of the image processing functions.

### **3.3 Vision System Integration with GuRoo**

The primary reason for this project objective was to demonstrate the ability of the current vision system to function in the dynamic environment which GuRoo has been built to operate in. The objective entailed correcting several aspects of the vision system which have, thus far, warranted no further attention. These aspects were factors such as the effects of ambient light on the captured images, and its effects on colour segmentation, along with the correction of various focal issues which present themselves when working in a dynamic environment.

The reason these aspects have not been considered before is due to the fact that the system has not been tested in a real-time environment. Working with static images allows the selection of images which lend themselves better to their intended use. Unfortunately, a dynamic environment such as that which GuRoo will be operating in does NOT lend itself to such luxuries.

# Chapter 4

## Software Developments

The software developments required for this project fall into two distinct categories. These areas are the vision debugging applications, and the hardware control software for the SH4.

Due to the nature of this project objective, a relatively in-depth understanding of both the vision debugging applications and the hardware control software was required. For this reason, it was undertaken as a task for this project to produce documentation for the relevant applications, as at the commencement of the project, no such documentation had been produced.

The applications were documented in a non-formal manner, and user's guides were produced which will give the user a detailed insight into the application logic and data flow used in each of the applications. This style of documentation allows a developer to determine where additional functionality should be inserted into the existing program logic, and also allows the easy identification of the location of program settings in the event that changes are required. These user guides are included in "Appendix A: Hardware User Guide" and "Appendix B: Software User Guide".

## **4.1 Win32 Software**

The software developed by Mark Chang for the SH4, along with its accompanying debugging applications, have quite a unique file architecture. The technique used is called “common compile-time libraries”. This technique allows the sharing of data structures and definitions among multiple applications without requiring redefinition. The file structure adopts a minimalist approach to application design, allowing an entire suite of programs to share a single header file. This header file contains settings and definitions which are common to several applications. An example of this in the application system in question is the output resolution. This is a setting that is common to both the SH4 software and the debugging application. To ensure that the settings are the same, the appropriate values are stored in a library which is referenced externally by both applications. The architecture has its roots in Linux software development, and as such offers most of its benefits to an implementation in such an environment.

In accordance with this project’s objective of enabling developments to be made on the software for GuRoo’s vision system under either a Linux or Win32 environment, it is necessary to preserve this file architecture when redeveloping the applications for Win32. Fortunately, it is a trivial task to work with such an architecture under Win32. The software environment chosen for the development of the Windows versions of the software is Microsoft Visual C++. Project workspaces for each of the required applications in GuRoo’s vision system application suite are created over the top of the existing architecture, making reference to the common header files, allowing the existing system to be used in the same manner under both Linux and Win32.

### **4.1.1 SH4 Software**

In order to enable further development of the SH4 software under Windows, it was necessary to use a cross-compiler for the SH4 which was compatible with Windows. Most of the commercially available cross-compilers are targeted at the Linux environment; however, there are numerous products available for Windows.

Renesas, the owners of Hitachi's semiconductor electronics division, makes available a product called "SuperH RISC Engine C/C++ Compiler Package", which includes a compiler, assembler and linker for the SH4. This product was available for download from Renesas' website ([www.renesas.com](http://www.renesas.com)) under the "Products" section.

An alternative to the official compiler package for the SH4 was the GNU movement's compiler collection, GNU-SH, which is based on GCC. The Win32 version of this compiler collection was distributed by a company called KPIT Cummins ([www.kpit.com](http://www.kpit.com)). To gain access to the "Downloads" section of their website, registration was required.

The object files generated by the compiler can conform to one of several object file formats. The most popular of these object file formats are ELF (Executable and Linking Format) and COFF (Common Object File Format) [10]. Both these file formats are similar in almost all respects, including the functionality they each offer. They both allow the specification of object code (which is generated by the compiler) and executables (which is generated by the linker).

GNU GCC generates files using the ELF file format, and as such, in order to keep file formats consistent, the chosen cross-compiler for Windows should also produce files in the ELF object file format. Fortunately, both the Renesas and GNU cross-compilers for the SH4 are available in versions which produce object files in the ELF and COFF object file formats. The ELF versions have both been trialled, and both are equally capable of being used for this project.

An integrated development environment called High-performance Embedded Workshop (HEW) [11] was developed by Renesas for use with its compiler collection. Fortunately, the GNU cross-compiler was also recognised by this system, allowing either compiler chain to work cooperatively with it. In alternative to HEW are the command line tools which are also shipped with each of the above mentioned compiler chains.

### **4.1.2 Debugging Software**

Porting the debugging application for GuRoo's vision system, VDebug, to Win32 was a trivial task. As mentioned earlier, creating a Microsoft Visual C++ project over the top of the existing file architecture leaves few issues unresolved.

The primary concerns when redeveloping the applications for Windows are the display and communication functionality. The display modules of VDebug make use of the OpenGL graphics libraries. The convenience of this is that the OpenGL libraries are readily available for both the Windows and Linux platforms from the OpenGL website ([www.opengl.org](http://www.opengl.org)). In addition to the standard OpenGL libraries, VDebug employs functionality from a non-standard OpenGL library called the OpenGL Utility Kit (GLUT). GLUT exposes functions which specialise in window creation and manipulation. These form the framework for VDebug. For details on setting up VDebug for Windows, refer to "Appendix B: Software User Guide".

The other major concern was the communication module. Serial communications are handled not too differently in Windows with comparison to Linux, however, changes do need to be made in order for the module to work. Both Linux and Windows refer to the serial device as a file. This allows the standard file I/O functions to be used to access the device. The difference lies in the way the different platforms refer to open files. Linux refers to an open file using a file number, which is declared of type integer. Windows, on the other hand, uses a derived data structure called a handle. This difference prevents the use of the same function definitions in a C header file, inflicting the need to redefine the functions according to the current operating environment.

In order to allow the software to compile with the correct function definitions, it is necessary to incorporate the declaration a global constant which indicates the current operating environment. This is achieved by including the declaration of a constant called “WIN32” in the Visual C++ project settings. Each of the locations where changes are necessary due to the operating environment checks whether this constant is defined, and declares the appropriate functions.

Another area of major work on the debugging application was the addition of a frame status window. The information displayed in this window is that pertaining to the current frame. These details include:

- Frame resolution
- Frame number
- Current frame rate
- Application running time
- Ball and goal locations, sizes and distances
- Ball speed and direction

In order for this information to be transmitted from the SH4, an appropriate protocol needed to be defined. The protocol was designed to be able to transmit information about each of the objects detected, while frame information such as resolution and frame rate can be calculated on the client (or PC) side. The protocol can be seen in Figure 10.

OBJECT	BALL					
PROPERTY	X Centre	Y Centre	Speed	Size	Distance	Direction
FIELD SIZE (bits)	10	10	10	20	10	9
BIT START	0	10	20	30	50	60

OBJECT	BLUE GOAL				YELLOW GOAL			
PROPERTY	X Centre	Y Centre	Size	Distance	X Centre	Y Centre	Size	Distance
FIELD SIZE (bits)	10	10	20	10	10	10	20	10
BIT START	69	79	89	109	119	129	139	159

**Figure 10 - Frame Status Protocol**

This protocol allows the transfer of all relevant image data in a 22 byte header attached to each image. The field sizes were chosen to accommodate the largest possible values in each of the specified areas. For example, using a 10-bit field for the x-centre and y-centre will allow the ball to be located anywhere in an image up to 1024 x 1024. This is well outside the operating range of the current hardware.

## **4.2 SH4 Improvements**

### **4.2.1 Resolution Improvements**

The OmniVision OV7620 camera chip used in the current vision system has a 664 x 492 image array. The size of this image array is what determines the maximum image resolution, as each array element translates to a single pixel in the image captured by the camera. The size of the images captured at this resolution far exceeds that of an image capable of being processed on the SH4, so a lower resolution is required in order for the vision system to function. Windowing is a feature of the camera chip which allows this to be achieved. The windowing process defines a horizontal and vertical boundary for the camera chip to use when determining which portion of the captured image constitutes a frame. The OV7620 allows windowing to be used to reduce the image size from the maximum resolution of 664 x 492 pixels to as small as 4 x 2 pixels.

However, there are also negative consequences of windowing. The field of view captured in the image reduces proportionally to the reduction in the image size. The effects of a reduction in to the field of view are obvious; it may prevent the detection of an object which may otherwise have been detected, simply because it fell outside the windowing range. Hence, ideally, windowing should be kept to a minimum. Unfortunately, restrictions imposed by the current image processing system prevent such advice from being adhered to.



Initially, GuRoo's vision system was designed for use by The University of Queensland's ViperRoos local vision soccer team [12]. Given the nature of these robots, the field of view requirements are significantly different. The ViperRoos' perspective is located approximately 20cm above ground level, reducing the vertical field of view required to locate the objects of interest. This allows a larger proportion of the pixels to be invested into the horizontal field of view. GuRoo, however, has a perspective located around 1m above ground level. This enforces a requirement for a greater vertical field of view, which in turn diminishes the available pixels for a horizontal field of view.

The resolution used by the vision system when implemented for the ViperRoos was 256 x 64. This reflects the availability of greater horizontal resolution due to a sacrificed vertical resolution. The effects of such a resolution on the field of view can be seen in Figure 11.



**Figure 11 - Initial Resolution (256 x 64) vs Improved Resolution (128 x 128)**

In order to improve the field of view available to the image processing functions, an image resolution is required that more evenly distributes the horizontal and vertical field of view. Initially, the resolution was increased to 128 x 128. This was chosen to keep the memory required to store the image constant, however, correct the field of view problem. At this resolution, an interesting trend became apparent. The pixels transmitted by the camera were not square. Further investigations, at higher resolutions, revealed that the pixels were exhibiting a height-to-width ratio of 1:2. The reason this was not investigated further is due to the fact that circumventing the problem was trivial; the vision debug software incorporates a "zoom factor" on each of the horizontal and vertical axes when displaying the image. The horizontal zoom factor was set to twice that of the vertical to enable a more accurate visualisation.



**Figure 12 – Height-to-width comparison - 1:1 vs 2:1**

Using the current image processing algorithms, this “make-do” correction of the problem will have little-to-no effect, as the objects are identified based on the area of the image consumed by the object. This value will be invariant to the zoom factor used at the display end of the data pipeline. However, if the image processing algorithms were to, at a later evolution, apply such constraints as height-to-width ratio in the detection of objects, this problem will need to be investigated further and corrected.

Another solution to the field of view problem is the use of the camera’s QVGA mode. QVGA mode works in a similar manner to interlacing. It halves the output data by ignoring every second line of data, and using full interlaced resolution on each line. This enables the use of the entire image for the analysis and detection of objects, while effectively reducing the memory required to store the image.

### **4.2.2 Speed Improvements**

The overall performance of the vision system is dependent on many factors. There are potential performance bottlenecks at several locations in the data pipeline, including the transmission of frame data across the SCCB (Serial Camera Control Bus) from the camera to the SH4, the processing of the image on the SH4, and the transmission of the frame through the SH4 board’s SCI interface to the PC. In order to achieve the maximum system performance, each of these potential performance bottlenecks should be optimised to function at their highest possible data rate.

Initially, the vision system was functioning at a frame rate of 0.1 Hz at a resolution of 256 x 256, with no image processing. This frame rate would not be very useful in a dynamic environment, for obvious reasons. It was necessary to find the cause of this frame rate, and optimise the bottleneck as required. The process used to achieve this was to trace the data flow back from the PC to the camera, and analyse the operation of each of the possible bottlenecks.

The first bottleneck inspected was the serial link between the SH4 and the PC. This link operated at 115.2 kbps. At a resolution of 256 x 256, with 16 bits/pixel, it is necessary to transmit a total of 1,048,576 bits/frame. This allowed a frame rate of 0.1 Hz through the link. This marked the identification of the first, and primary bottleneck. Unfortunately, the SCI interface on the SH4 does not operate at a higher data rate, making this issue unresolvable.

Although the SH4 was currently not processing the images, but rather buffering them and transmitting them, investigating the processor for bottlenecks was not necessary. The work of Mark Chang however, had demonstrated that performance benefits were possible through the enabling of the SH4's instruction cache. In accordance with instructions supplied by Mark, the instruction cache was enabled. This will allow greater performance of the image processing functions when they are implemented on the vision board.

Finally, the link between the camera board and the SH4 was interrogated. Communication between these devices happens over the serial camera control bus (SCCB). The SCCB is a direct data link which operates at 400 kbps. Again, a prohibiting factor was identified. The SH4 is responsible for requesting frames from the camera chip. The rate of requests from the SH4 was currently set well below that which the OV7620 is capable of responding to. This was set deliberately to enable the SH4 to manage the data coming from the camera. This request rate was not amended, to allow for the implementation of the image processing functions on the SH4.

# **Chapter 5**

## **System Integration**

The final project objective of integrating the vision system with GuRoo required the resolution of several minor issues inherent with the vision system. The apparent issues included the poor focusing of the lens, and the failure of the current camera housing to eliminate the penetration of ambient light to the imaging sensor. These issues had not effected the performance of the vision system when used for testing the hardware and software independently, however, when used concurrently, various calibration issues where inevitably going to arise. It appeared necessary to correct these issues prior to amalgamating the image processing software with the hardware, to ensure the process went smoother.

### **5.1 Focus Correction**

Poor focus has certain benefits in image processing. It reduces the effects of noise in the acquired images, and can, in certain cases, eliminate the need for an erosion phase in the processing pipeline. However, the benefits of poor focus can be compensated for by software functions. The negative effects of poor focus, such as the exclusion of small or distant objects, can not be. Therefore, the more preferable option for a vision system is to have a well focused image which takes marginally longer to process.



**Figure 13 - Focus - Old Spacers**

In order to achieve a correctly focused image from GuRoo's vision system, it was first necessary to gain a better understanding of the operation of the lens. The details of the lens used have not previously been documented, and the properties of the lens needed to be inferred. The details which could be ascertained from the lens are its focal length and aperture. The focal length is 2.8 mm, and the aperture is 1.4. The focal length reflects the centre of curvature for that particular lens, while the aperture represents the ratio between the effective diameter and the focal length. This is what determines the brightness of the image.

Further research into the functionality and characteristics of CCTV lenses revealed that such lenses are available which produce images in various image sizes, for example, 1/2", 1/3" and 1/4" formats. After referring to the documentation for the camera chip used, it was determined that the lens used in the vision system complies with the 1/3" format.

Armed with this new characteristic of the lens, it was then possible to attempt to find a retailer who stocked the lens, and who would be able to provide more detailed information. Fortunately such a stockist exists, and had the required information published on their website [13]. The focal length was published as 3.5 – 8.0 mm.

The initial understanding of how the lens operated was that the imaging sensor could be located anywhere within the range of the focal length, and optimal focus would be achieved. Experimental results proved otherwise. The imaging sensor was placed at a distance of 5 mm, and the image was found to exhibit no better focus than that achieved with the old spacers, which were cut at arbitrary length.

Further tests demonstrated that as the spacing from the lens to the imaging device approached the published maximum of 8.0 mm, the image became more focused. Surprisingly, however, even at 8.0 mm, the focus of the image was not even close to that which was expected. This implied that perhaps the information acquired regarding the characteristics of the lens was inaccurate, or that the understanding of the operation of the lens was flawed. Further investigations were undertaken into the operation of the lens and it was found that the latter were the case.

Video lenses are in fact “lens systems”, which have characteristics that differ quite considerably from that of normal lenses. These characteristics are introduced through the lens’ construction. All video lenses have a 1<sup>st</sup> and 2<sup>nd</sup> Principal Point [14]. The 1<sup>st</sup> Principal Point is the focal point for the first lens. The rays pass through this point and are reflected from the interior of the lens housing. These rays then coincide again at the 2<sup>nd</sup> Principal Point. A second lens is placed at this point which is responsible for creating the image of the desired format (1/2”, 1/3”, etc) at the necessary focal length. This focal length behind the lens is known as the “back focal length”. Another metric worth noting is one called the “flange length”. This is the distance between the contact point between the lens and the camera housing and the focal point. Each of the two standard camera mount types, C and CS, have fixed values for this flange distance. These points are illustrated in Figure 14 and Figure 15.

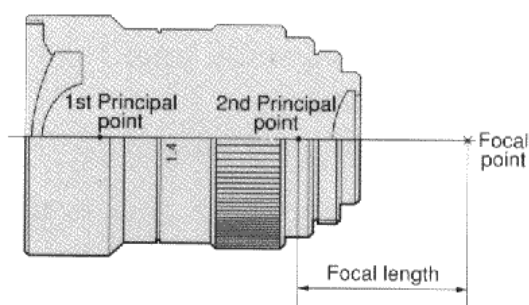


Figure 14 - 1st and 2nd Principal Points [14]

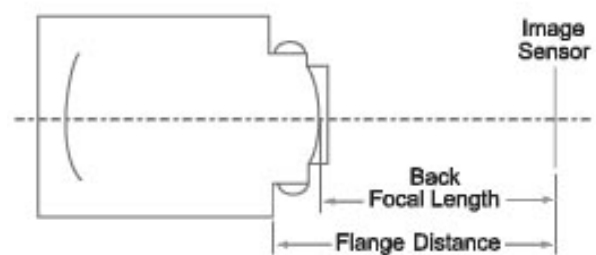


Figure 15 - Flange Distance [15]

This new understanding of video lenses explained the behaviour previously experienced, ie, the image becoming sharper as the spacing approached the upper extreme of the focal range. The lens used in this vision system uses a CS-mount, which has a back flange distance of 12.526 mm.

## **5.2 Saturation Suppression**

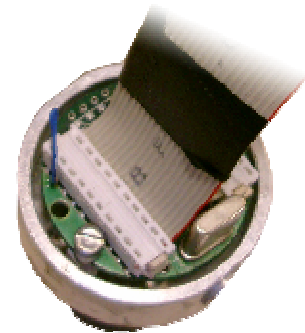
The primary concern which makes itself evident from the effects of an over-concentration of ambient light on an image is the reduction of definition of colour. For example, a particular problem which presented itself in the course of this project was the detection of red objects as a pale shade of red, or pink. This can have extreme effects on the accuracy of the detection of objects, and must be compensated for in the calibration of the lookup tables used in the segmentation process.



**Figure 16 - Saturated Image**

The reason such a high concentration of ambient light was present was due to the poor fit of the camera PCB in its housing. When properly inserted, the PCB leaves a fissure approximately 2mm in width around its edge, as seen in Figure 17.

To achieve the best results from the image processing functions, this problem needed to be corrected. The chosen resolution was a redesign of the camera housing. The new camera housing was designed in cooperation with Damien Kee, who was responsible for the technical drawings along with ordering its manufacture. The new camera housing was designed with two objectives in mind. The first was to eliminate the presence of ambient lighting, and the second was to allow an easier correction to the focal problems, previously described. Details of the new camera housing design can be seen in “Appendix E: Camera Housing Redesign”.



**Figure 17 - Old Camera Housing**

To eliminate the ambient light, the new housing was designed to allow the camera PCB to fit inside with 0.5 mm around it to allow for insertion and removal. The length of the housing was also increased to allow the back of it to be sealed. The problem with sealing the back of the housing is doing it in such a way that the data and power cables used to connect the camera board to the SH4 board may extrude from the housing unobstructed. This was achieved through the use of a flexible backing, which has a hole marginally large enough to accommodate the size of the cables.

To correct the focal problems previously described, spacers were machined to the diameter and length necessary to separate the image sensor from the lens at the precise distance to achieve the desired focus.



### 5.3 Vision System Integration with GuRoo

In order to demonstrate the vision system functioning with GuRoo, certain issues needed to be corrected. The first of these was the mounting of the system on GuRoo. This year saw the conclusion of the work with CSIRO to produce an embedded inertial measurement unit (eIMU) [16]. This device has since been incorporated with GuRoo, and used in such projects as the Active Balance System for a Humanoid Robot [17]. It was decided that the eIMU would be connected in the head region of GuRoo. The existing head design did not allow for such an addition, hence it was decided that GuRoo's head would be redesigned. This head redesign was also to allow for the mounting of two cameras in preparation for stereo vision, along with a place to mount the SH4 vision board. The current head does not make allowance for a secure mounting of the board at all. Details of the new head design are shown in "Appendix D: Head Redesign".

At the completion of this project, the new head was still in the construction stage, and due to the need for the presence of the eIMU, a "make-do" solution has been used, which allows the secure attachment of the eIMU, along with the attachment of the camera. Unfortunately, it was not feasible to securely attach the SH4 vision board at this time.

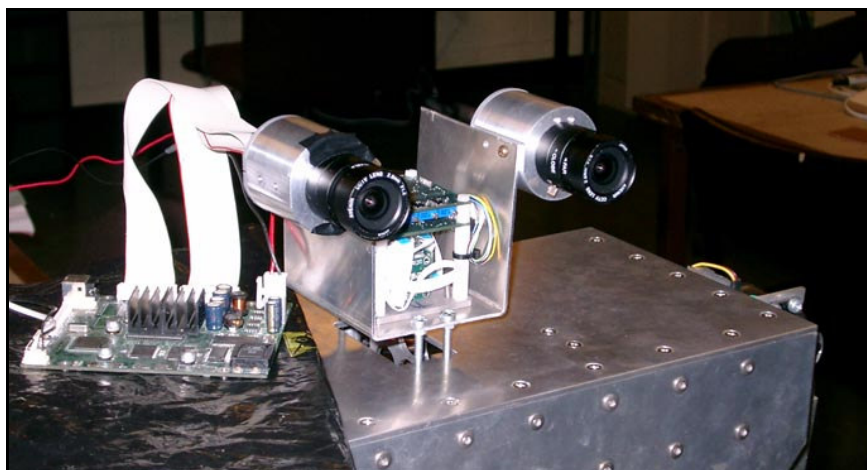


Figure 18 - "Make-do" Head Solution

Another issue which prevented the mounting of the vision system on GuRoo was the existing serial board. Due to an educational background that did not include a great deal of electronics, the diagnosis of a malfunctioning serial circuit did not prove to be the trivial task it should have been. After a great deal of reading, it was found that it was in fact the ICL232 chip which was not functioning correctly. The output of the transmission pin was operating at  $\pm 4.6\text{V}$ , as opposed to the prescribed  $\pm 12\text{V}$ . This was preventing communications taking place over a distance any greater than 10-15 cm. The simplest way to resolve this issue was to construct a new serial board. This was achieved with the assistance of Damien Kee.

## **Chapter 6**

# **Project Evaluation and Future Work**

The vision system discussed in this document is far from being ready for use in the target environment it is being built to operate in. There are large amounts of work left in areas ranging from hardware redesign and optimisation, through to the research and implementation of stereo vision. This project has turned out to be a bridging project, which took the vision system from what was primarily a static test environment, and fine-tuned each of the required aspects necessary to prepare the system for its target, dynamic environment.

Some of the work which was undertaken during the course of this project still remains incomplete, primarily the incorporation of the image processing algorithms with the existing hardware. No significant progress was made towards completing this goal. This can be attributed to the numerous side-steps taken during the project. For example, at the outset of the project, it was not anticipated that such problems as the image focus and lighting issues would need to be corrected in order to allow the image processing functions to operate. These sub-objectives turned out to be more time consuming than initially anticipated.

## 6.1 Flash for SH4 Vision Board

One of the more tedious aspects of the project is the need to reprogram the SH4 board every time it is turned on. This is due to the current use of an EEPROM to store the uploaded program. The work of Mark Chang has seen the incorporation of a flash memory unit into the vision system design, which will result in the need to reprogram the board only when changes are made to the firmware.

While this work was completed prior to the conclusion of this project, time constraints prevented the addition of the flash unit into the vision system. Various software changes need to be made in order for the system to accept the flash unit, which is not a trivial task. The completion of this task would be a great time-saving exercise in the long-term.

## 6.2 Detection of Additional Objects

The existing vision system has a field of view that falls considerably short of the maximum field of view achievable with the current vision hardware. The importance of the field of view to a vision system is incredibly high. This is due to several reasons. Firstly, in order to make GuRoo's vision system comparable to that of the human vision system, which forms the ultimate goal, allowing it to have the same field of view is paramount. While the implementation of stereo vision would be a large step towards achieving this goal, the existing system can be optimised in certain ways to bring the goal closer to realisation.

Secondly, the field of view of the vision system is the one of the greatest determining factors in the object detection process. No calibre of image analysis algorithms can find objects which do not appear in the image to start with. In order to mimic the human visual system, a careful balance between lens and camera must be selected.

One of the challenges which arises during the process of increasing the field of view of a vision system is the inherent increase in resolution. This has certain repercussions on both the hardware and software. The camera chip currently used in GuRoo's vision system, fortunately, offers a solution to this resolution dilemma. This solution is its QVGA mode. This camera mode allows the maximum field of view of the lens to be utilised, while providing a significant reduction to the required resolution to accommodate that field of view. Enabling the QVGA mode halves the resolution of the acquired images by using every second line of the image in a full-interlaced format to create the image. This in turn allows an increased field of view at an incredibly reduced processing cost.

### **6.3 Redesign of SH4 Vision Board**

The SH4 vision board has been designed as a general purpose vision board, which offers many features which aren't currently, nor ever will be, used by GuRoo's vision system. The removal of these hardware components will provide numerous benefits to the vision system. These benefits include a reduction in the cost of producing the boards, along with significant savings in the size of the board. It is anticipated that a vision board which contained only the required components to offer the required functionality would be approximately half the size of the current board. This space saving would have positive repercussions in the head design of GuRoo.

# Chapter 7

## Conclusion

GuRoo's vision system, incorporating the SH4 vision board, has been fine-tuned for use in a dynamic, volatile environment. The system is now capable of streaming higher resolution images with better colour definition and focus. These improvements to the vision system will improve the accuracy and reliability of the image processing functions, featuring the object detection algorithms.

Previous difficulties and delays encountered due to a deficit knowledge of the UNIX operating environment have been prevented through the redevelopment of the vision system's firmware and debugging applications to a Microsoft Windows environment. These packages have been ported in such a way that the underlying architecture has remained intact. This will allow an easy transition from one operating environment to the other without the need to make redundant changes. The vision system's application suite has been documented to allow an easy orientation for future work on the project.

The focus of the images captured by the vision system has been corrected. This was achieved through the adjustment of the spacing between the lens and the imaging device. The saturating effects of ambient light inside the camera housing have been eliminated through the redesign of the camera housing. The new camera housing still complies with all the mounting requirements of GuRoo's up and coming head, while eliminating the penetration of light.

Although GuRoo is still waiting for his new head, the vision system has been mounted onto the "make-do" head (Figure 18), which accommodates the eIMU. The serial board used to communicate between the vision system and the PC used for debugging has been remade, with the assistance of Damien Kee.

# Bibliography

- [1] *RoboCup Website* [Online], 11 Apr, 2003 – last update. Available: <http://www.robocup2003.org> [accessed 16 Apr. 2003].
- [2] *RoboCup Regulations* [Online], 2 Sept, 2002 – last update. Available: <http://www.robocup.org/regulations/4.html> [accessed 16 Apr. 2002]
- [3] Hitachi Ltd. (2002), *SH7750 Series Hardware Manual*
- [4] B. Horn, *Robot Vision*, The MIT Press, Cambridge Massachusetts, 1986.
- [5] Prasser, D. (2001) *Vision Software for a Humanoid Robot*, Undergraduate Thesis, University of Queensland
- [6] Bruce, Balch, Veloso (2000) *Fast and Inexpensive Colour Image Segmentation for Interactive Robots*, Proceedings of the 2000 IEE/RSJ International Conference on Intelligent Robots and Systems
- [7] Gonzalez, R. & Woods, R. (1992) *Digital Image Processing*, Addison-Wesley, Reading Massachusetts
- [8] Jain, A. K. (ed.) (1998) *Real-Time Object Measurement and Classification*, NATO ASI Series (Vol. 42), Springer-Verlag Berlin Heidelberg, Germany

- [9] M. Abe, Building Cross Development Environment Targetting SH4 System, [www.linuxsh.sourceforge.net/docs/abe/2001320-gcc2.97/README\\_E.php3](http://www.linuxsh.sourceforge.net/docs/abe/2001320-gcc2.97/README_E.php3), August, 2001.
- [10] *What are ELF, COFF, and PE COFF?* [Online], 2003 – last update. Available: <http://www.theparticle.com/cs/bc/os/elfpecoff.html> [accessed 19 Sept 2003]
- [11] Hitachi Ltd. (2000), *High-performance Embedded Workshop User's Manual*
- [12] *ViperRoos Homepage* [Online], 2000 – last update. Available: <http://www.itee.uq.edu.au/~chang/ViperRoos/> [accessed 3 Oct 2003]
- [13] *Total Turnkey Solutions* [Online], 2003 – last update. Available: [http://www.turnkey-solutions.com.au/cam\\_avenir\\_varifocal\\_lenses.htm](http://www.turnkey-solutions.com.au/cam_avenir_varifocal_lenses.htm) [accessed 25 Oct 2003]
- [14] *Rapitron: Technical Notes - CCTV Lenses* [Online], 2003 – last update. Available: <http://www.rapitron.it/guidaobE.htm> [accessed 25 Oct 2003]
- [15] *Navitar Video Lenses: Optical Characteristics of Video Lenses* [Online]. Available: [http://www.navitar.com/zoom/cctv\\_op\\_char.htm](http://www.navitar.com/zoom/cctv_op_char.htm) [accessed 25 Oct 2003]
- [16] *CSIRO: Robotics and Automation – eIMU* [Online]. 2003 – last update. Available: <http://www.cat.sciro.au/cmit/automation/commercial/eimu.html>
- [17] Low, T. (2003) *Active Balance System for a Humanoid Robot*, Undergraduate Thesis, University of Queensland



# **Appendix A**

## **Hardware User Guide**

---

**VBSH4**  
**USER GUIDE**

v1.0

October 2003

---

## Preface

This document serves as a guide for setting up and using the VBSH4 vision board designed and implemented by Mark Chang. This is not the definitive guide to all the vision system has to offer. This document reflects the understanding gained during the course of my dealings with the system. Upon the commencement of my project nothing had been documented, and much time was wasted gaining an understanding of how the system operated. This guide is simply an attempt to reduce the time required by students in the future when becoming acquainted with Mark Chang's vision system.

Having said this, the work I did with the system did not make use of many of the features it has to offer. If any feature of the system is not covered in this document, or something covered is found to be incorrect or incomplete, I encourage any additions which may enhance the quality and completeness of this guide.

I hope this guide accomplishes its goal, and you have many fascinating hours playing with the intriguing piece of work that is the SH4 vision board.

Anthony Peters  
October 2003

## Version History

October 2003	Anthony Peters	Initial version
--------------	----------------	-----------------

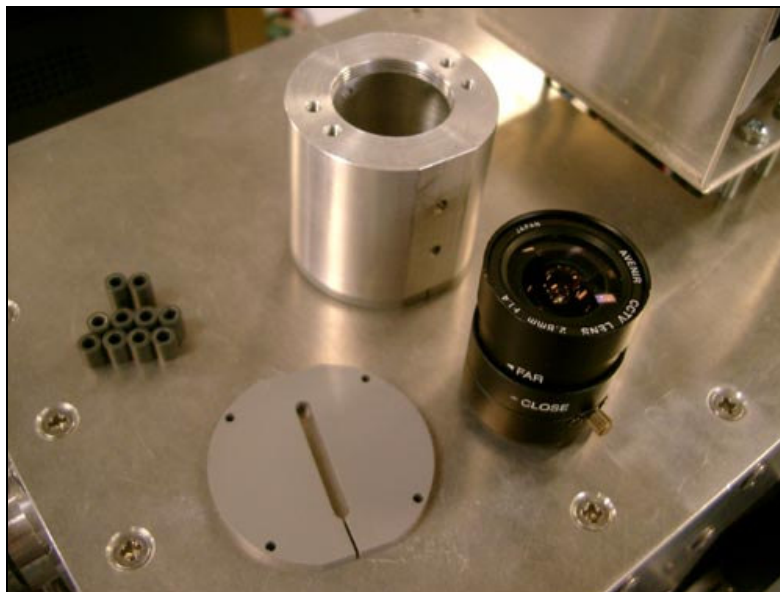
# Table of Contents

1.	Hardware Requirements.....	1
2.	Connecting the Hardware.....	3
3.	Hardware Programming.....	8
3.1	Programming.....	9
3.2	Module Descriptions.....	11
4.	Firmware.....	12

**1**

# **Hardware Requirements**

In components which constitute GuRoo's vision system are various. In addition to the SH4 vision board and camera board, there is also several other components. These are shown in Figure 1.



**Figure 1 - Vision System Components**

The components seen in the above image are the camera board spacers to the left, the camera housing to the top, with the backing to the bottom. The lens is also shown.

Details on how to connect the hardware is given in "Chapter 2: Connecting the Hardware".

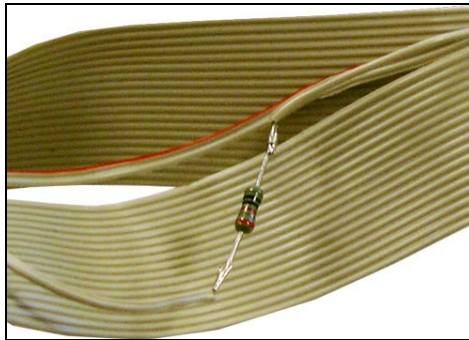
# **2**

## **Connecting the Hardware**

Before preceding with this chapter, please identify the components shown in “Chapter 1: Hardware Requirements”.

In order to set-up the vision system follow the procedure outlined below.

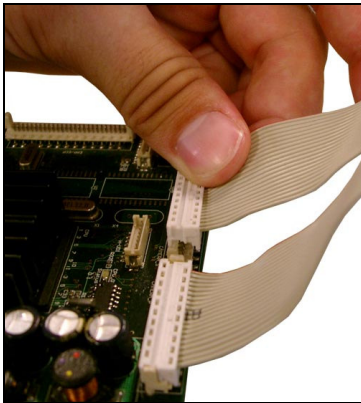
1. Identify the ribbon cables used to connect the camera board to the SH4 board. One of these cables will have a resistor attached to the cable. This is the power cable.



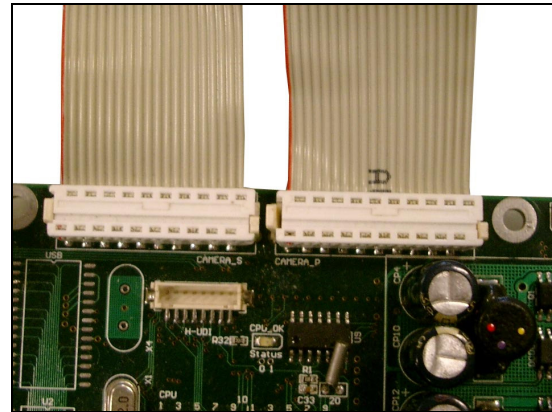
**Figure 2 - Ribbon cables to connect the camera board to the SH4 board**

2. The connectors on both the camera board and the SH4 board are labelled. One cable is for the signal, the other for power. Ensure when connecting these cables that the power cable is connected to the power connector on both boards. Likewise for the signal cable.



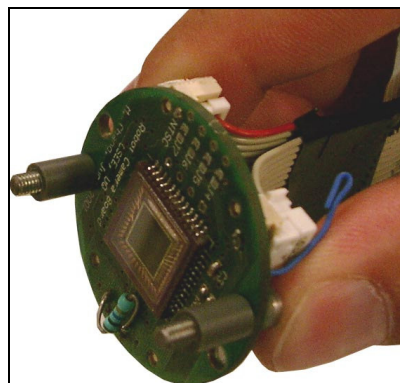


**Figure 3 - Connecting the power and signal cables for the camera**



**Figure 4 - The SH4 and camera board connectors are labelled**

3. Once the camera board is connected to the SH4 board, the camera can be enclosed in the camera housing. Before doing this, place the spacers on the screws as shown. The spacers will allow the lens to focus the image properly.



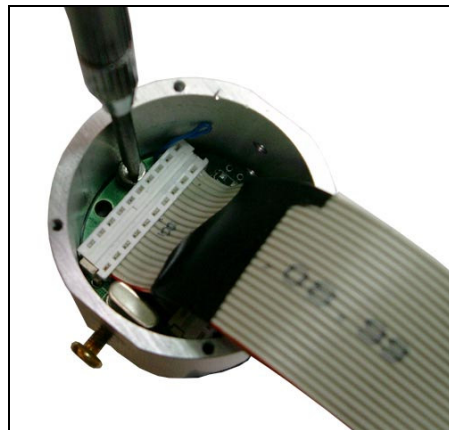
**Figure 5 - The spacers are required to focus the image**

4. Inserting the camera board into the housing requires care, and can become quite tedious if care is not taken. This is due to the spacers falling off the screws. The easiest way to insert the camera board into the housing is to do it on the side, as shown in Figure 6. Ensure to try to line up the screws with their holes before inserting the camera board.



**Figure 6 - Inserting the camera board into the housing**

5. Screw the camera board into place. At least 2 screws are required to hold the board in place. Up to 4 screws may be used, however, in most cases, is not necessary.



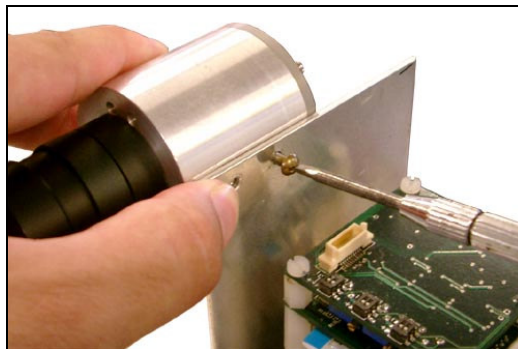
**Figure 7 - Securing the camera board in the housing**

6. Once the camera board is secured in the housing, the housing can be sealed to prevent light from getting in. To attach the seal, simply separate the top and bottom at the split, and slide over the cables. Screw the seal onto the housing as shown in Figure 8.



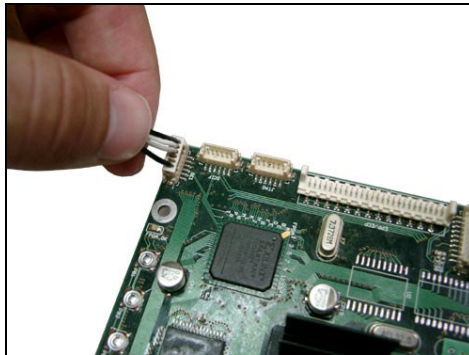
**Figure 8 - Securing the housing seal**

7. Now that the camera housing is sealed, the lens can be attached to the front. Simply screw the lens in as far as possible.
8. Connect the camera housing to GuRoo. Be cautious when doing this. The camera board may be damaged if screws are placed in the front holes. Ensure that the board is not obstructing the holes before attempting to put the screws in.



**Figure 9 - Attaching the camera to GuRoo**

9. Now that the vision system is fully assembled, it can be connected to the PC. To do this, connect the serial cable to the SCI connector on the SH4 board. The serial cable can be connected directly to the PC, or attached to an extension serial cable if desired.



**Figure 10 - Connecting the SH4 to the PC**

# **3**

# **Hardware Programming**

## 3.1 Programming

Using ploader to upload a program to the SH4 is very simple. Before attempting to use this program however, the SH4 must be powered correctly, and must have completed its boot sequence.

The SH4 vision board should be powered between +7 and +12 VDC. The power indicators on the board will indicate if this voltage is being received. If this is correct, the IPL will complete the boot sequence, and the boot status LEDs will show do their “LED-shuffle”. Once complete, only the orange LED will remain on. Now the board may be programmed.



**Figure 11 - The SH4 has booted successfully**

If using Linux, open a terminal window and navigate to the directory where ploader is located. If using Windows, open a Command Prompt window in the similar directory. The command for Linux and Windows is slightly different. The Windows version requires that the serial port to use is specified in the command line. The Linux version will only work with ttyS0, or COM1. The command line for ploader is as follows:

Linux:

Windows:

---

```
./ploader <filename>
```

```
ploader <port name> <filename>
```

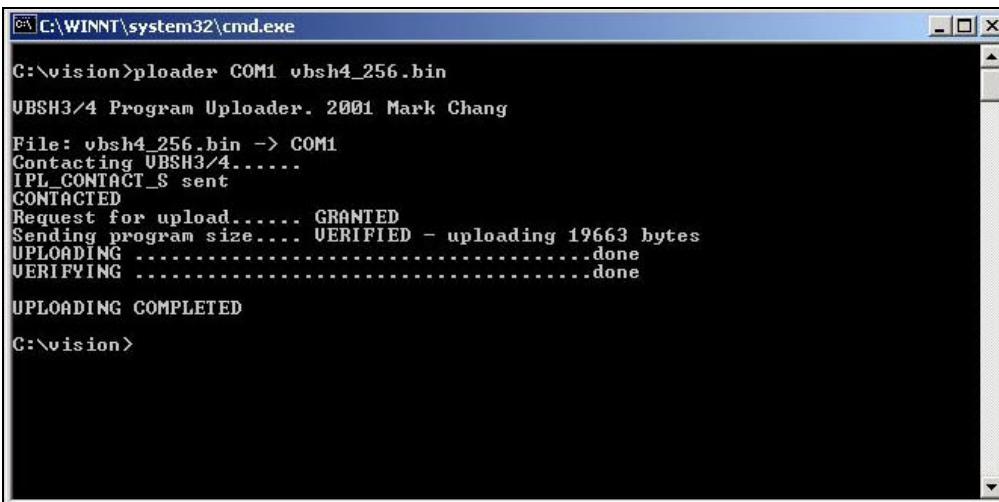
Example:

```
./ploader vbsh4.bin
```

Example:

```
ploader COM1 vbsh4.bin
```

Once the upload is complete, a message similar to that shown in Figure 12 will be displayed.



```
C:\WINNT\system32\cmd.exe
C:\vision>ploader COM1 vbsh4_256.bin
UBSH3/4 Program Uploader. 2001 Mark Chang
File: vbsh4_256.bin -> COM1
Contacting UBSH3/4.....
IPL_CONTACT_S sent
CONTACTED
Request for upload..... GRANTED
Sending program size.... VERIFIED - uploading 19663 bytes
UPLOADING .....done
VERIFYING .....done

UPLOADING COMPLETED

C:\vision>
```

Figure 12 - Successful upload with ploader

In the event that ploader fails when attempting to upload the program, firstly ensure that the board has been powered and that it is waiting for an upload. It is common to attempt to send the program early, before the boot sequence has completed. Should this be the case, simply wait until the boot sequence has finished, then attempt the upload again.

If this was not the problem, ensure that all the cables are connected to the correct ports. It is also quite common to connect the serial cable to the wrong port on the PC or vision board.

## 3.2 Testing the upload

Once the program has been successfully uploaded, the green LED will remain on constantly, the red LED will heartbeat at a constant rate, and the orange LED will toggle with the receipt of each frame from the camera.

If this is not the response from the vision board, the board will be transmitting an error over the SCI interface to the PC. This message can be received by using an appropriate serial communications program on the PC. If under Linux, use `minicom`, on Windows, HyperTerminal.

To use `minicom`, type `minicom -s` and adjust the settings to ensure that all modem settings are cleared (initialisation string, etc), and that the program is listening on the correct device. Save the setup, and re-enter `minicom` by typing `minicom`. The error message being sent from the SH4 should be displayed on the screen.

If using HyperTerminal, create a new connection. Set the baud rate to 115200, and turn off flow control. Make sure the program is listening to the correct port number. Leave all other settings default. Again, the error message begin sent from the SH4 should be displayed on the screen.

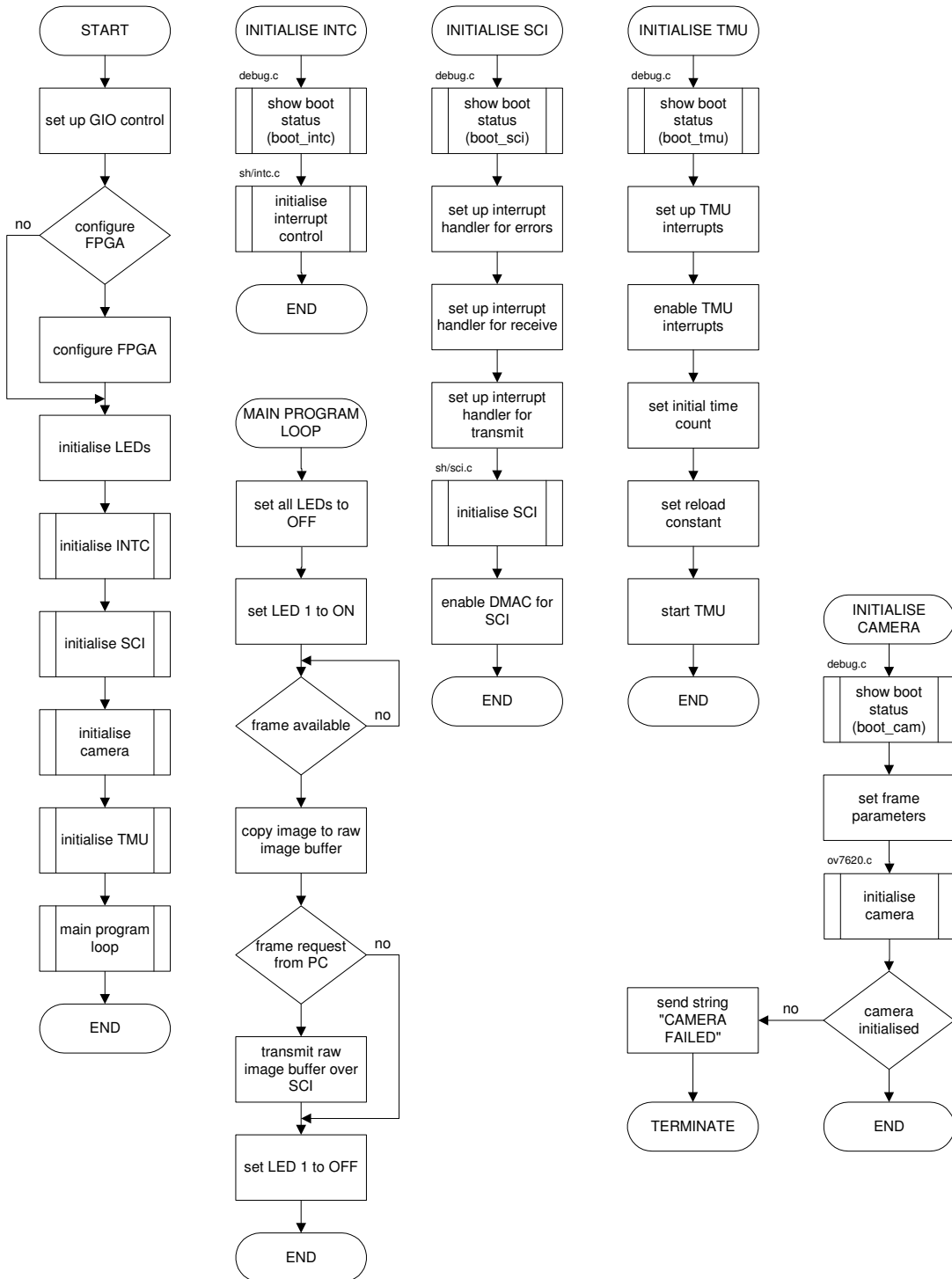


# **4**

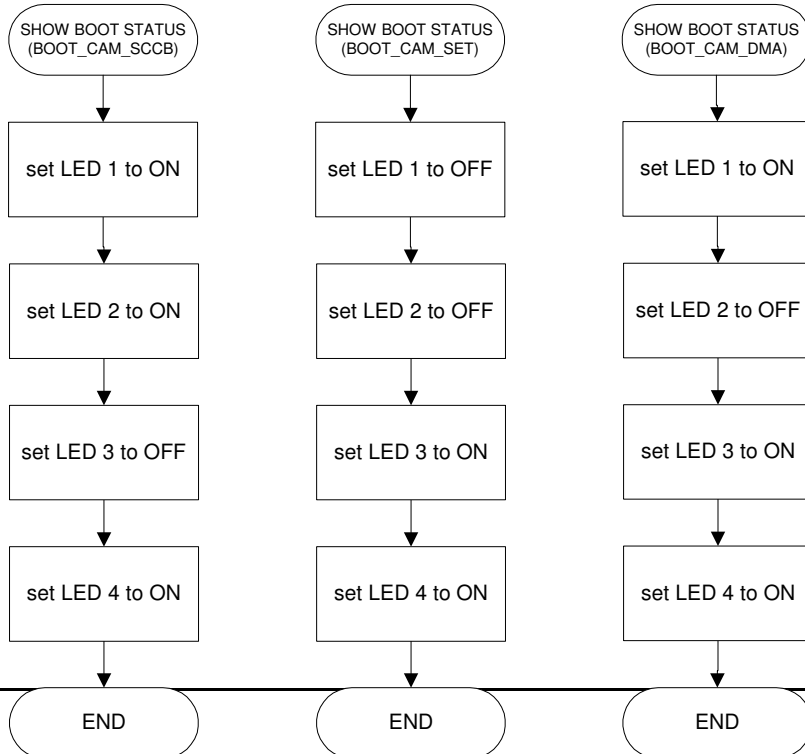
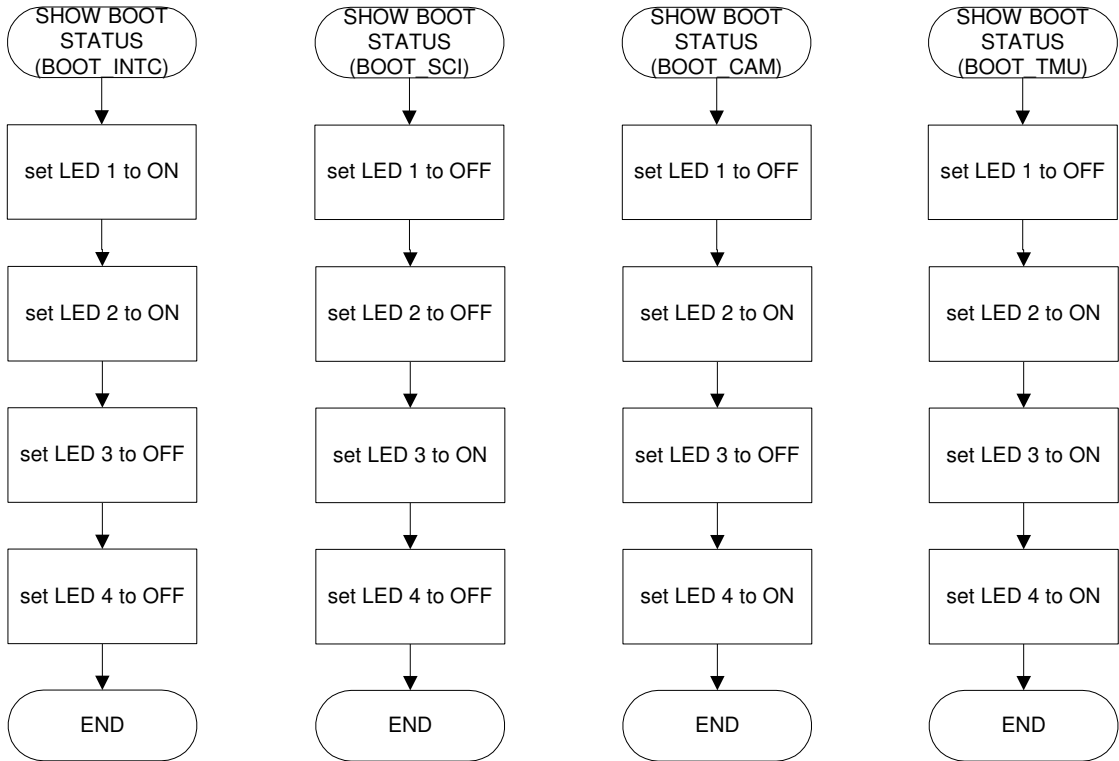
# **Firmware**

## 4.1 Software Flowcharts

File: vbsh4.c



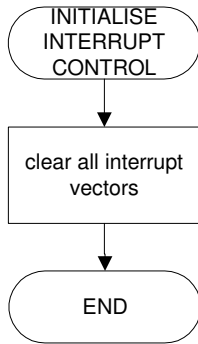
File: debug.c



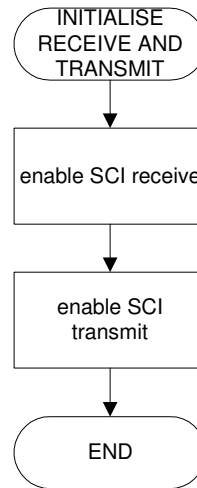
File: ov7620.c



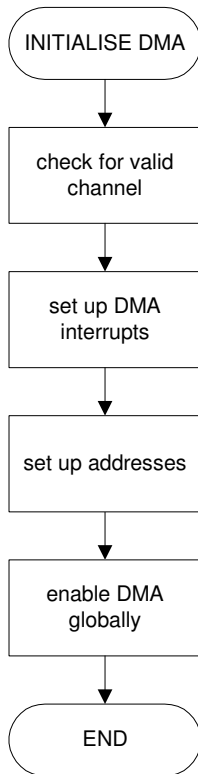
File: sh/intc.c



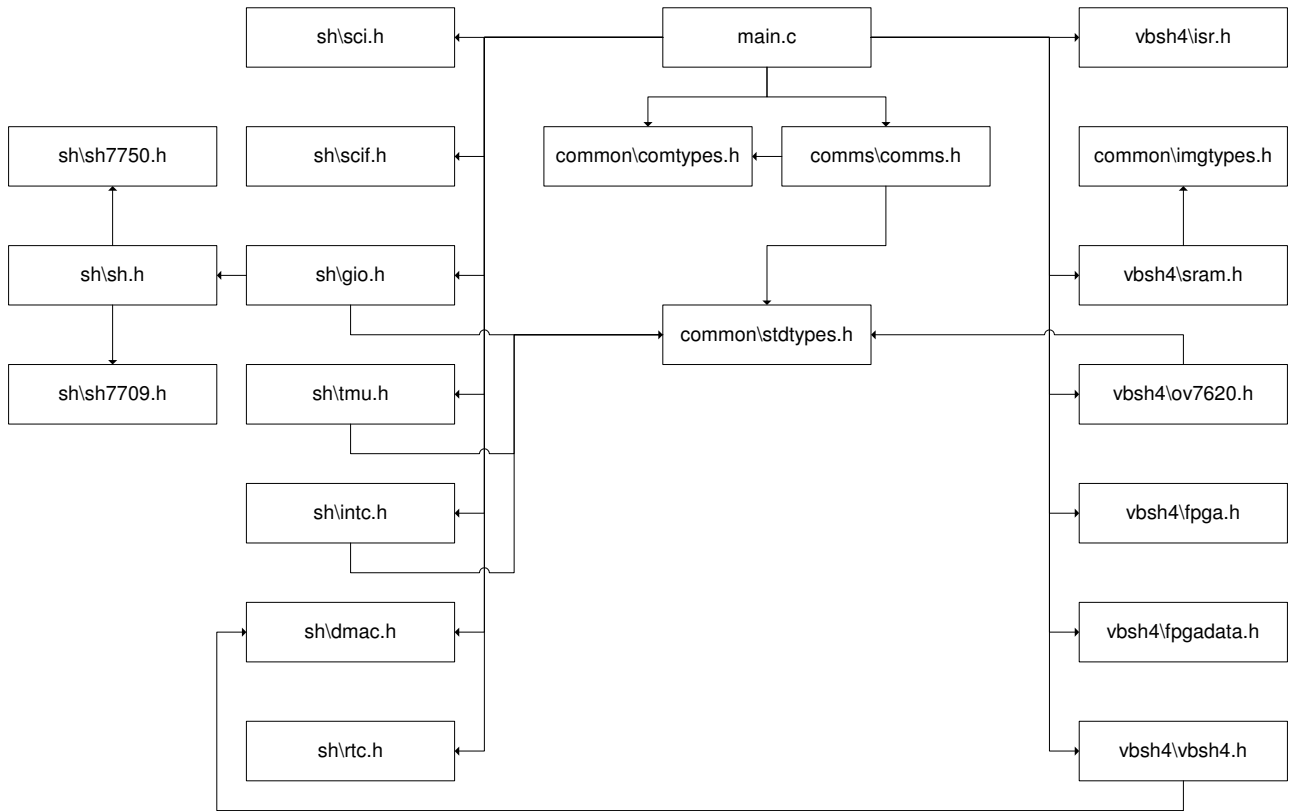
File: sh/sci.c



File: sh/dmac.c



## 4.2 File Dependencies



# **Appendix B**

## **Software User Guide**

---

**VDEBUG**  
**USER GUIDE**

v1.0

October 2003

---



## Preface

This document serves as a guide for installing and using the VDebug vision debug software which accompanies the SH4 vision board designed and implemented by Mark Chang. This is not the definitive guide to all the software has to offer. This document reflects the understanding gained during the course of my dealings with the software. Upon the commencement of my project nothing had been documented, and much time was wasted gaining an understanding of how the software operated. This guide is simply an attempt to reduce the time required by students in the future when becoming acquainted with the debug software.

Having said this, the work I did with the software did not make use of many of the features it has to offer. If any feature of the software is not covered in this document, or something covered is found to be incorrect or incomplete, I encourage any additions which may enhance the quality and completeness of this guide.

I hope this guide accomplishes its goal, and you have many fascinating hours playing with the intriguing piece of work that is the SH4 vision system.

Anthony Peters

October 2003

## Version History

October 2003

Anthony Peters

Initial version

# Table of Contents

1.	Program Overview.....	1
2.	Installation.....	3
2.1	Linux Installation.....	5
2.2	Windows Installation.....	5
3.	Technical Operation.....	6
3.1	File Dependencies.....	7
3.2	Module Descriptions.....	8
4.	Software Flowcharts.....	11

**1**

# **Program Overview**

VDebug is a program written by Mark Chang for the purpose of assisting in the debugging of the SH4 Vision Board used by the University of Queensland Robotics Department.

The program was originally written to allow the user to see a copy of the image captured by the vision system. The program has since been extended to give post-analysis information such as object location and velocities.

Given the nature of the application, the bottleneck for the execution of this program will always be the communication with the SH4. Whether this bottleneck appears due to the communication interface, as is the current situation, or whether the bottleneck appears on the SH4 due to processing requirements, it would contradict the purpose of this application to detract from the performance of the vision system.

# 2

# Installation

Installation of VDebug on any platform requires the installation of the OpenGL drivers. All versions of Windows ship these drivers standard, while most Linux installations will also install them, it may be necessary to manually install them if no games were installed. Amazingly, these are the only Linux programs that depend on the drivers. In the event that the drivers are not installed, the latest drivers are available from [www.opengl.org](http://www.opengl.org).

An extension to the OpenGL libraries, which will not be installed by default on any platform, is the OpenGL Utility Toolkit (GLUT). These drivers are also available from the Open website.

VDebug is included in the "guroo\_vs" application suite. In order to use the software, unzip the GuRoo\_VS.zip file. A directory structure will be created under the guroo\_vs directory, which forms the root directory for the above mentioned guroo\_vs application suite.

## **2.1 Linux Installation**

The program files for VDebug reside in guroo\_vs/vdebug. For a description of the contents of these files, see “Chapter 3: Technical Operation”.

The program is now installed. In order to make changes to the application, open the relevant file in any text editor. To recompile the executable, type make in the guroo\_vs/vdebug directory.

## **2.2 Windows Installation**

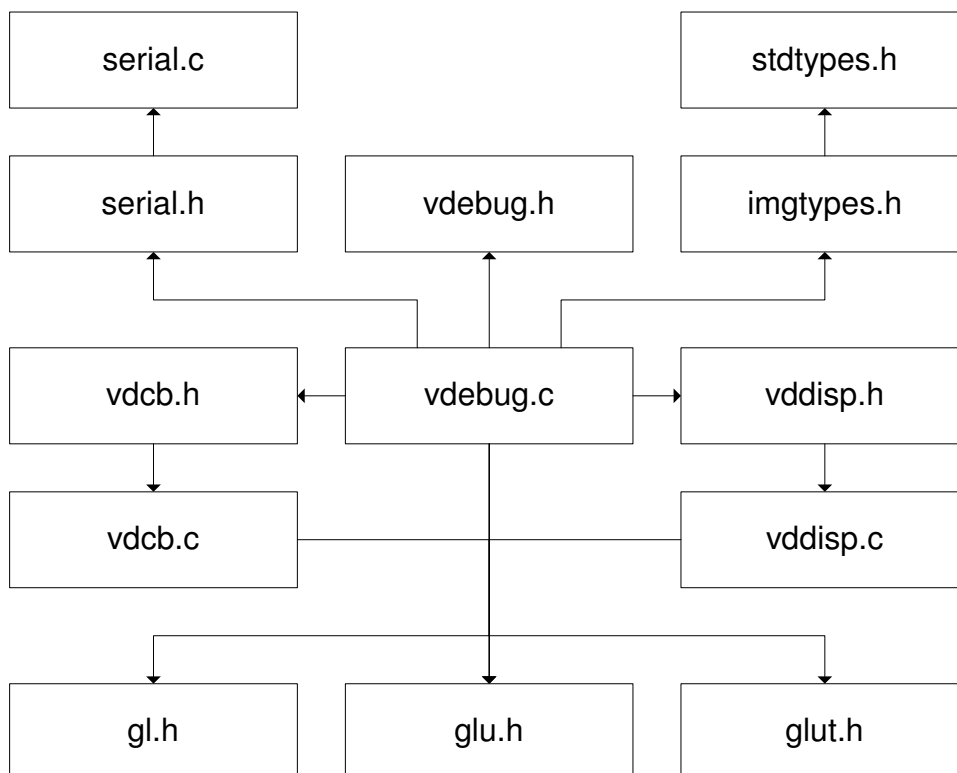
The Windows installation of VDebug is not too different from that of Linux. Once the “guroo\_vs” application suite has been unzipped, the VDebug application files can be found in the guroo\_vs/vdebug directory. Inside this directory is vc directory. This is the location of the Visual C++ project files. The executable for VDebug can be found in the guroo\_vs/vdebug/vc/Debug directory.

# **3**

# **Technical Operation**



### 3.1 File Dependencies



## 3.2 Module Descriptions

### 3.2.1 vdebug.h

This module defines the operating constants for the application. Most of the settings declared in this module should never need to be altered.

#### #includes

```
common\imgtypes.h
```

#### #defines

typedef struct VDebug	HIST_SIZE	glColourRed()
ZOOM	HIST_R	glColourGreen()
WIN_YUV_XSIZE	HIST_G	glColourBlue()
WIN_YUV_YSIZE	HIST_B	glColourlRed()
TEX_MAIN	HIST_Y	glColourlGreen()
TEX_YUV	HIST_U	glColourlBlue()
TEX_HIST	HIST_V	

#### Functions

None

### **3.2.2 vdcb.h**

This module declares the functions used to receive the image data and convert it to a format usable by OpenGL as a texture.

#### **#includes**

None

#### **#defines**

None

#### **Functions**

void IdleFunc (void);

void convert\_image (void);

void time\_event (int value);

### 3.2.3 vddisp.h

This module contains the definitions for the functions which are used to redraw the various program windows. These are declared in the main module as OpenGL callback functions.

#### #includes

None

#### #defines

None

#### Functions

```
void win_main_init (int width, int height);
void win_main_resize (int width, int height);
void win_main_draw (void);
void win_yuv_init (int width, int height);
void win_yuv_resize (int width, int height);
void win_yuv_draw (void);
void win_hist_init (int width, int height);
void win_hist_resize (int width, int height);
void win_hist_draw (void);
```

### 3.2.4 imgtypes.h

This module declares all the program constants which pertain to the image format and size. These settings are used to window sizes on startup.

#### #includes

common\stdtypes.h

#### #defines

IMG_XSIZE	IMG_RGB	YUVMAP_SIZE
IMG_YSIZE	IMG_GREY	YUVMAPIMG_SIZE
IMG_SIZE	IMG_YUV	YUVMAP_TABLE_SUBSAMPLE
IMG_GREY_XSIZE	YUVMAP_XSIZE	
IMG_GREY_YSIZE	YUVMAP_YSIZE	
IMG_GREY_SIZE	YUVMAP_PSIZE	

#### Functions

None

### 3.2.5 stdtypes.h

This module declares a set of standard macros which are used in various places throughout the application.

#### #includes

None

#### #defines

CLIP2MS(t)	FALSE	HALF_PI
TIME2MS(t)	OK	QUARTER_PI
CLIP2SEC(t)	ERR	THREEQUARTER_PI
TIME2MIN(t)	ERROR	TWO_PI
CLIP2MIN(t)	MAX(a,b)	RAD2DEG(r)
TIME2HOUR(t)	MIN(a,b)	DEG2RAD(d)
BASIC_TYPE	ABS(a)	CLIPDEG360(d)
NULL	SGN(a)	
EVER	ROUND(f)	
KBYTE	AVG(a,b)	
MBYTE	CLIP(v,t)	
TRUE	PI	

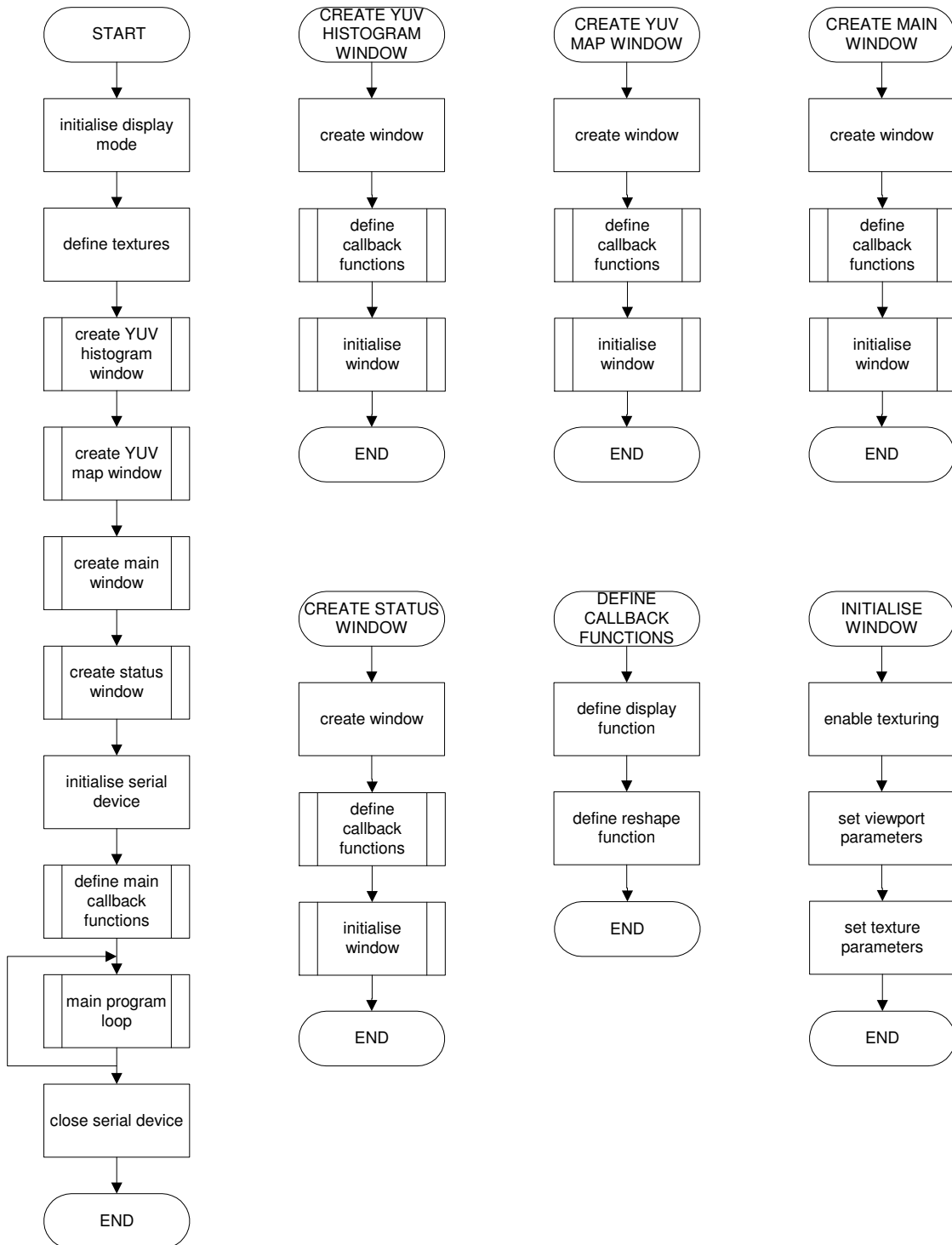
#### Functions

None

# 4

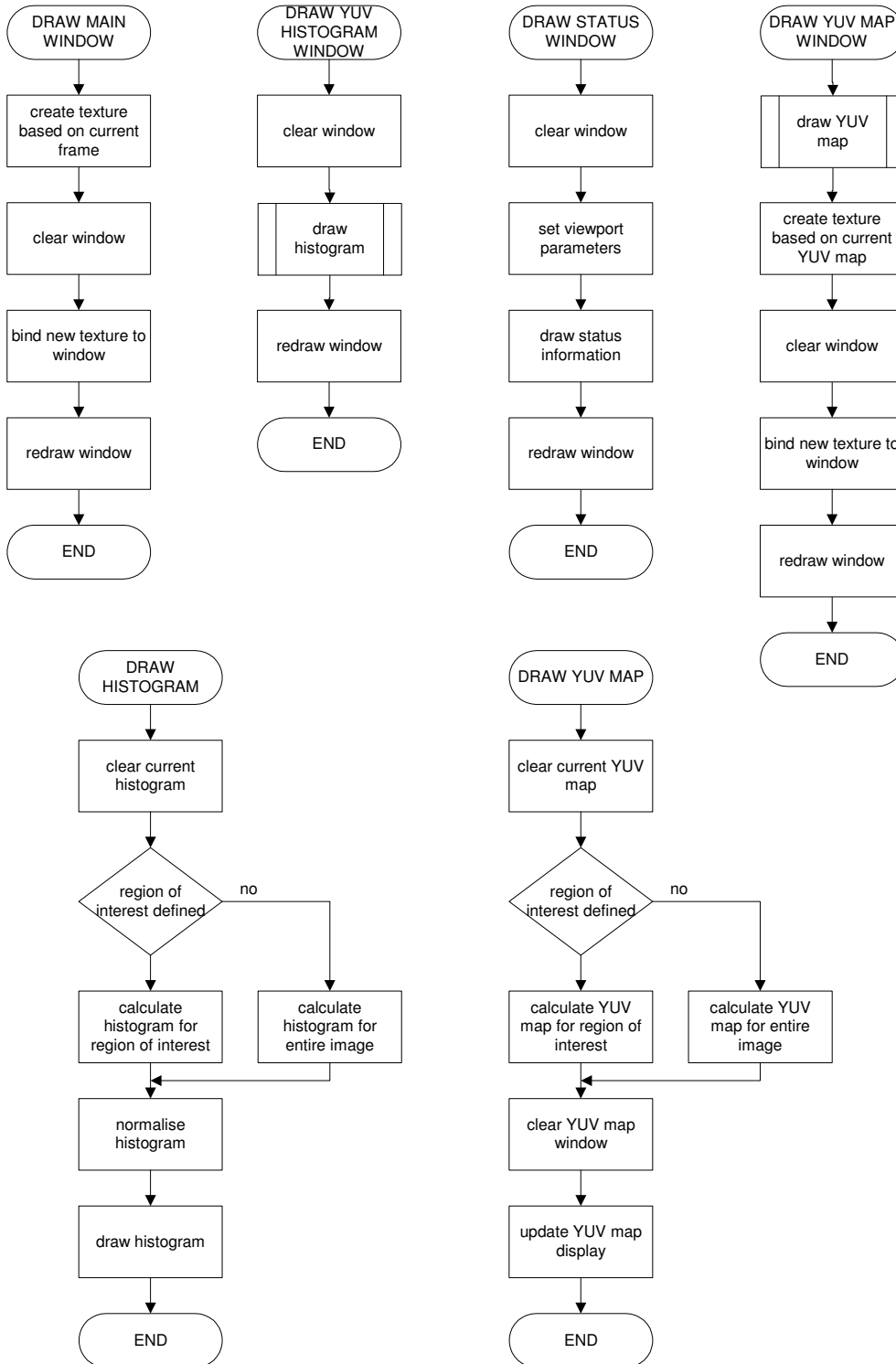
## Software Flowcharts

File: vdebug.c

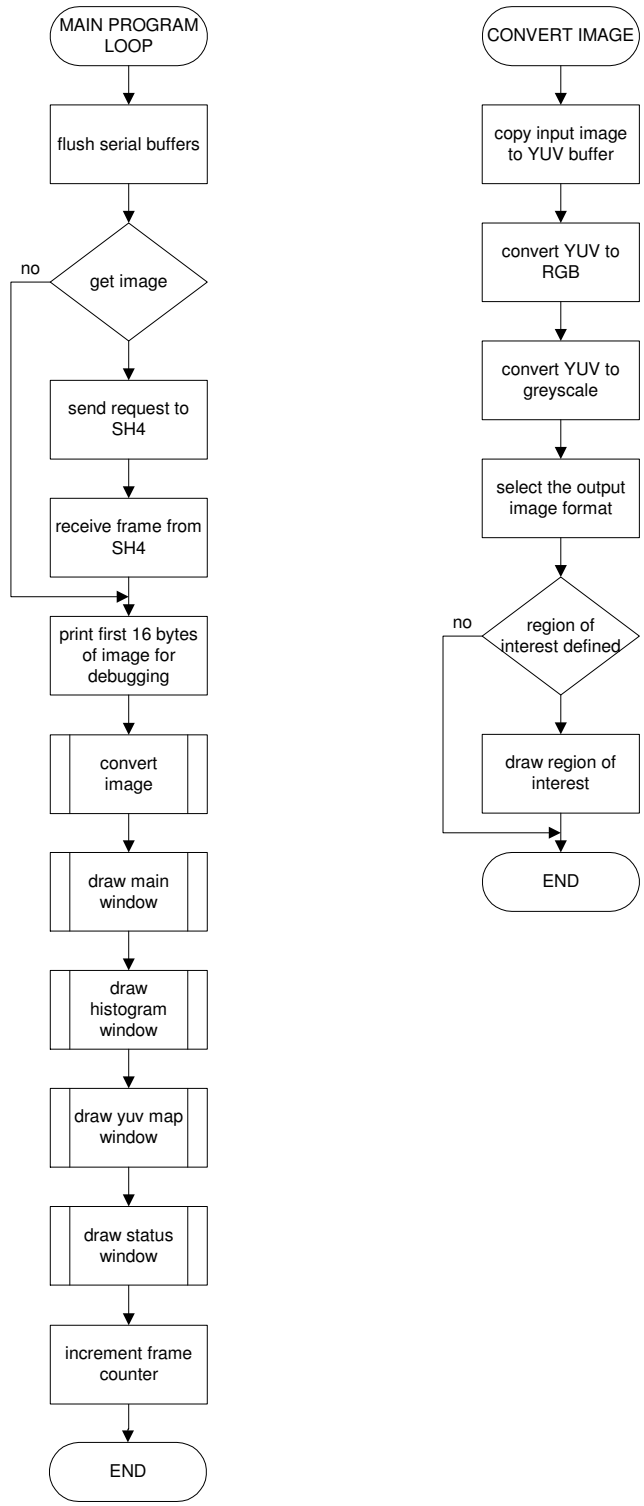




File: vddisp.c



File: vdcb.c



# Appendix C

## Additional Code

The following code is a portion of code found in the `vdebug.c` file. The code was written with the purpose of creating a status window in order to display the current frame information.

```
// *****  
// Status WINDOW  
  
glutInitWindowSize(800, 300);  
glutInitWindowPosition(100, 400);  
g_vdebug->win_status = glutCreateWindow("VDebug Frame Status");  
  
win_status_init(300, 300);  
  
glutDisplayFunc(&win_status_draw);  
glutReshapeFunc(&win_status_resize);  
  
glutIdleFunc(&IdleFunc);  
glutKeyboardFunc(&keyPressed);  
glutMotionFunc(motion_func);  
glutMouseFunc(mouse_func);  
  
// Open Serial Device  
g_vdebug->fd = serial_open(SERIAL_DEVICE);  
serial_init(g_vdebug->fd);  
  
win_main_init(IMG_XSIZE * ZOOM, IMG_YSIZE * ZOOM);  
  
g_vdebug->roi_xs = 0;  
g_vdebug->roi_ys = 0;  
g_vdebug->roi_xe = 0;  
g_vdebug->roi_ye = 0;  
  
g_vdebug->frame = 0;  
g_vdebug->frame_rate = 5;  
g_vdebug->img_type = IMG_RGB;  
g_vdebug->get_image = 1;
```

```

// initial object parameters
g_vdebug->frame_info.b_xloc = 128;
g_vdebug->frame_info.b_yloc = 180;
g_vdebug->frame_info.b_size = 500;
g_vdebug->frame_info.b_dist = 1.5;
g_vdebug->frame_info.b_speed = 5;
g_vdebug->frame_info.b_dir = 90;

g_vdebug->frame_info.bg_xloc = 0;
g_vdebug->frame_info.bg_yloc = 0;
g_vdebug->frame_info.bg_size = 0;
g_vdebug->frame_info.bg_dist = 0;

g_vdebug->frame_info.yg_xloc = 0;
g_vdebug->frame_info.yg_yloc = 0;
g_vdebug->frame_info.yg_size = 0;
g_vdebug->frame_info.yg_dist = 0;

```

The following code is a portion of code found in the `vddisp.c` file. The code was written with the purpose of displaying the status window containing the current frame information.

```

// We call this right after our OpenGL window is created.
void win_status_init(int width, int height) {
    glutSetWindow(g_vdebug->win_status);
    glClearColor(0.0f, 0.0f, 0.0f, 0.0f); // Clear The Background Color To Black

    glMatrixMode(GL_PROJECTION);
    glLoadIdentity(); // Reset The Projection Matrix

    // Calculate The Aspect Ratio Of The Window
    glOrtho(-1.0, 1.0f, -1.0f, 1.0f, 0.0f, 10.0f);
    glMatrixMode(GL_MODELVIEW);
}

/* display a string at the raster position (x,y) */
void displayStr(const char *str, GLfloat x, GLfloat y)
{
    /* choose what font we wish to use */
    void* fontName = GLUT_BITMAP_HELVETICA_12;
    unsigned int index = 0;

    /* where do we want the text to go */
    glRasterPos2f (x, y);

```

```
/* Display the string on the screen character by character.
c strings are null terminated.
*/
while (str[index] != 0 && index < 1000)
{
    glutBitmapCharacter (fontName, str[index]);
    ++index;
}
}

void win_status_resize(int width, int height)
{
    if (height==0) // Prevent A Divide By Zero If The Window Is Too Small
        height=1;

    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();

    // Calculate The Aspect Ratio Of The Window
    glOrtho(-1.0, 1.0f, -1.0f, 1.0f, 0.0f, 10.0f);
    glMatrixMode(GL_MODELVIEW);
}

void win_status_draw(void)
{
    char str[50];

    glutSetWindow(g_vdebug->win_status);
    glClear(GL_COLOR_BUFFER_BIT); // Clear The Screen And The Depth Buffer

    glColor3f(0.0, 1.0, 0.0);

    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho(0, 10, 0, 10, 1, -1);

    /* Show header info */
    displayStr("FRAME INFORMATION", 0.3, 9.3);
    displayStr("_____ ", 0.3, 9.1);

    /* Show IMAGE info */
    displayStr("IMAGE:", 0.5, 8.2);

    sprintf(str, "%ix%i", IMG_XSIZE, IMG_YSIZE);
    displayStr("Resolution:", 0.7, 7.4);
    displayStr(str, 1.7, 7.4);
}
```

```

sprintf(str, "%4.2f fps", g_vdebug->frame_rate);
displayStr("Frame Rate:", 0.7, 6.6);
displayStr(str, 1.7, 6.6);

sprintf(str, "%i", g_vdebug->frame);
displayStr("Frame:", 0.7, 5.8);
displayStr(str, 1.7, 5.8);

sprintf(str, "%4.2f s", glutGet(GLUT_ELAPSED_TIME)/1000.0);
displayStr("Timer:", 0.7, 5.0);
displayStr(str, 1.7, 5.0);

/* Show BALL info */
if (g_vdebug->frame_info.b_xloc == 0 && g_vdebug->frame_info.b_yloc == 0)
    displayStr("BALL: (Not found!)", 3.0, 8.2);
else
    displayStr("BALL:", 3.0, 8.2);

if (g_vdebug->frame_info.b_xloc == 0 && g_vdebug->frame_info.b_yloc == 0)
    sprintf(str, "N/A");
else
    sprintf(str, "(%i,%i)", g_vdebug->frame_info.b_xloc, g_vdebug->frame_info.b_yloc);
displayStr("Location:", 3.2, 7.4);
displayStr(str, 4.0, 7.4);

if (g_vdebug->frame_info.b_size == 0)
    sprintf(str, "N/A");
else
    sprintf(str, "%i pixels", g_vdebug->frame_info.b_size);
displayStr("Size:", 3.2, 6.6);
displayStr(str, 4.0, 6.6);

if (g_vdebug->frame_info.b_dist == 0)
    sprintf(str, "N/A");
else
    sprintf(str, "%i.00 m", g_vdebug->frame_info.b_dist);
displayStr("Distance:", 3.2, 5.8);
displayStr(str, 4.0, 5.8);

if (g_vdebug->frame_info.b_speed == 0)
    sprintf(str, "N/A");
else
    sprintf(str, "%i pixels/frame", g_vdebug->frame_info.b_speed);
displayStr("Speed:", 3.2, 5.0);
displayStr(str, 4.0, 5.0);

if (g_vdebug->frame_info.b_dir == 0)
    sprintf(str, "N/A");

```

```

else
    sprintf(str, "%i degrees", g_vdebug->frame_info.b_dir);
displayStr("Direction:", 3.2, 4.2);
displayStr(str, 4.0, 4.2);

/* Show BLUE GOAL info */
displayStr("BLUE GOAL: (Not found!)", 5.2, 8.2);

displayStr("Location:", 5.4, 7.4);
displayStr("N/A", 6.2, 7.4);

displayStr("Size:", 5.4, 6.6);
displayStr("N/A", 6.2, 6.6);

displayStr("Distance:", 5.4, 5.8);
displayStr("N/A", 6.2, 5.8);

/* Show YELLOW GOAL info */
displayStr("YELLOW GOAL: (Not found!)", 7.5, 8.2);

displayStr("Location:", 7.7, 7.4);
displayStr("N/A", 8.5, 7.4);

displayStr("Size:", 7.7, 6.6);
displayStr("N/A", 8.5, 6.6);

displayStr("Distance:", 7.7, 5.8);
displayStr("N/A", 8.5, 5.8);

glutSwapBuffers();
}

```

The following is portion of code taken from `vdc.c`, and forms the main program loop. It defines the protocol used to receive the frame information for the SH4.

```

void IdleFunc(void)
{
#define BUFFER_SIZE 8

    int img_size;
    int img_size_r;
    int img_size_x;
    int img_size_y;
    int status;

    int i, j;

```

```

unsigned char read_buffer[BUFFER_SIZE];

serial_flush(g_vdebug->fd);

#ifdef WIN32
    Sleep(3);
#endif

if (g_vdebug->get_image == 1)
{
    i = 0;

    serial_writebyte(g_vdebug->fd, g_vdebug->img_type);
    printf("<WRITE>\n");

    img_size = IMG_SIZE;
    img_size_r = IMG_SIZE + IMG_YSIZE;
    img_size_x = IMG_XSIZE;
    img_size_y = IMG_YSIZE;

    /* Read the frame information */
    // Added Anthony Peters
    serial_readbyte(g_vdebug->fd, read_buffer);
    g_vdebug->frame_info.b_xloc = read_buffer[0];
    serial_readbyte(g_vdebug->fd, read_buffer);
    g_vdebug->frame_info.b_yloc = read_buffer[0];
    serial_readbyte(g_vdebug->fd, read_buffer);
    g_vdebug->frame_info.b_size = read_buffer[0];
    serial_readbyte(g_vdebug->fd, read_buffer);
    g_vdebug->frame_info.b_dist = read_buffer[0];
    serial_readbyte(g_vdebug->fd, read_buffer);
    g_vdebug->frame_info.b_speed = read_buffer[0];
    serial_readbyte(g_vdebug->fd, read_buffer);
    g_vdebug->frame_info.b_dir = read_buffer[0];

    /* Ensure that the read buffer is REALLY empty */
    memset(read_buffer, 0x00, sizeof(read_buffer)); // Added Anthony Peters

    while (i < img_size)
    {
        status = serial_readbyte(g_vdebug->fd, read_buffer);

        if (status > 0)
        {
            for (j = 0; j < status; j++)
            {
                g_vdebug->img_org[i] = read_buffer[j];
            }
        }
    }
}

```



```

        i++;
    }
}

}

// print the first 16 byte for debug purpose
for (i = 0; i < 16; i++)
{
    printf("%3u ", g_vdebug->img_org[i]);
    if (i % 4 == 3)
        printf(" | ");
    if (i % 16 == 15)
        printf("\n");
}
printf("\n");

convert_image();
win_main_draw();
win_hist_draw();
win_yuv_draw();
win_status_draw();

g_vdebug->frame_rate = (float)g_vdebug->frame * 1000 / (float)glutGet(GLUT_ELAPSED_TIME);

g_vdebug->frame++;

}

```

The following portion of code is the contents of serial.h. This file is responsible for defining the serial functions appropriate for the current operating system.

```

#ifndef __SERIAL_H
#define __SERIAL_H

#ifdef WIN32
#define BAUDRATE        CBR_115200 //Changed by Prasser
#else
#define BAUDRATE        115200 // Added by Anthony Peters
#endif

#ifdef WIN32 /* Wlindows serial definitions */

#include <windows.h> //Added by Prasser
HANDLE serial_open(char* port_name);

```

```

void serial_close(HANDLE fd);
void serial_init(HANDLE fd);
int serial_flush(HANDLE fd);
int serial_readbyte(HANDLE fd, unsigned char *ch);
int serial_writebyte(HANDLE fd, unsigned char ch);
int serial_writebstring(HANDLE fd, const char *str, int len);
//int serial_getcodedbyte (HANDLE fd, unsigned char *c);
int serial_readcodedbyte (HANDLE fd, unsigned char *c);
//int serial_putcodedbyte (HANDLE fd, unsigned char c);
int serial_writecodedbyte (HANDLE fd, unsigned char c);

#else /* UNIX serial definitions */

int serial_open(char* port_name);
void serial_close(int fd);
void serial_init(int fd);
int serial_readbyte(int fd, unsigned char *ch);
int serial_writebyte(int fd, unsigned char ch);
int serial_writebstring(int fd, const char *str, int len);
int serial_getcodedbyte (int fd, unsigned char *c);
int serial_putcodedbyte (int fd, unsigned char c);
int serial_flush(int fd);

#endif

#endif

```

The following portion of code is the contents of serial.c. This file is responsible for defining the serial functions appropriate for the current operating system.

```

/* Created by Mark Chang
 * Changes required by Windows made by David Prasser
 * UNIX and Windows functions merged and cleaned up by Anthony Peters
 */

#include <stdio.h> /* Standard I/O definitions */
#include <string.h> /* String function definitions */
#include <fcntl.h> /* File control definitions */
#include <errno.h> /* Error number definitions */
#include <comms/serial.h>

#ifdef WIN32

#include <io.h> //Added by Prasser
#include <windows.h> //Added by Prasser.

```

```

#else

#include <unistd.h>          /* UNIX standard function definitions */
#include <termios.h>        /* POSIX terminal control definitions */

#endif

#include <common/rftable.h>
#ifndef _POSIX_SOURCE
#define _POSIX_SOURCE 1    /* POSIX compliant source */
#endif

#ifdef WIN32 // Windows serial functions

HANDLE serial_open(char* port_name)
{
    HANDLE fd; // Prasser File descriptor

    // Prasser makes file for generic writing, 0 sharing, NULL security
    fd = CreateFile(port_name, GENERIC_WRITE|GENERIC_READ, 0,
                   NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);
    if (fd == INVALID_HANDLE_VALUE)
    {
        // unable to open port
        fprintf(stderr, "ERROR (serial_open) : Unable to open %s - %d\n",
               port_name, GetLastError());
    }
    return (fd);
}

// All changed by Prasser.
void serial_close(HANDLE fd)
{
    if (fd != INVALID_HANDLE_VALUE)
        return;

    CloseHandle(fd);
    fd = INVALID_HANDLE_VALUE;
}

void serial_init(HANDLE fd) //Prasser fd used to be an int
{
    //struct termios options;
    DCB options;
    COMMTIMEOUTS timeouts;

    // Get current options for the serial port
    //tcgetattr(fd, &options); //Prasser makes it...

```

```

if (!GetCommState(fd, &options)) {
    printf("Error getting current comms data\n");
};

GetCommTimeouts(fd, &timeouts);
timeouts.ReadIntervalTimeout = 500;
timeouts.ReadTotalTimeoutConstant = 500;
timeouts.ReadTotalTimeoutMultiplier = 500;
timeouts.WriteTotalTimeoutConstant = 300;
timeouts.WriteTotalTimeoutMultiplier = 300;
SetCommTimeouts(fd, &timeouts);

// Set the baud rate for input and output
options.BaudRate = BAUDRATE; //prasser

// CONTROL OPTIONS
options.fOutxCtsFlow = FALSE; //prasser

options.fBinary = TRUE;          // Member must be true Prasser
options.fOutxDsrFlow = FALSE;
options.fDtrControl = DTR_CONTROL_DISABLE;
options.fDsrSensitivity = FALSE;
options.fOutX = FALSE;
options.fInX = FALSE;
options.fRtsControl = RTS_CONTROL_DISABLE;

options.fParity = FALSE;
options.Parity = NOPARITY;

options.StopBits = ONESTOPBIT;
options.ByteSize = 8;

FlushFileBuffers(fd);
SetCommState(fd, &options);
}

int serial_flush(HANDLE fd)
{
    COMSTAT          comStat;
    DWORD            dwErrors;
    int              nread, len;
    unsigned char    buf;

    /*
     * Get number of bytes in the read buffer
     */
    if (!ClearCommError(fd, &dwErrors, &comStat)) {
        return 0;
    }
}

```

```
    }

    len = comStat.cbInQue;

    /*
     * If data in buffer, read it all
     */
    while (len > 0) {
        if (ReadFile(fd, &buf, 1, &nread, NULL) == 0)
            break;
        len -= nread;
    }
    return len;
}

int serial_readbyte(HANDLE fd, unsigned char *ch)
{
    int bytesread;

    ReadFile(fd, ch, 1, &bytesread, NULL);

    if (bytesread == 1)
        return 1;
    else
        return 0;
}

int serial_writebyte(HANDLE fd, unsigned char ch)
{
    int byteswritten, status;
    unsigned char c;

    c = ch;

    status = WriteFile(fd, &c, 1, &byteswritten, NULL);

    if (status && (byteswritten != 1))
    {
        fprintf(stderr, "ERROR (serial_writebyte) : failed writing single byte to port.");
        fprintf(stderr, " - written %d, code %d\n", byteswritten, GetLastError());
        return 0;
    }
    else
        return 1;
}

int serial_writestring(HANDLE fd, const char *str, int len)
```

```
{
    int byteswritten, status;

    status = WriteFile(fd, str, len, &byteswritten, NULL);

    if (status && (byteswritten < len))
    {
        fprintf(stderr,
            "ERROR (serial_writebyte) : failed writing %i characters to port.", len);
        fprintf(stderr, " - written %d, code %d\n", byteswritten, GetLastError());
        return 0;
    }
    else
        return 1;
}

int serial_readcodedbyte (HANDLE fd, unsigned char *c)
{
    unsigned char ch, cl, tmp;
    int sh, sl;

    // Get high byte
    sh = serial_readbyte(fd, &ch);
    // Get low byte
    sl = serial_readbyte(fd, &cl);

    // Decode byte
    tmp = RFdecodingTable[ch];
    tmp = (tmp << 4) & 0xF0;
    *c = tmp;
    tmp = RFdecodingTable[cl];
    *c |= (tmp & 0x0F);

    return (sh && sl);
}

int serial_writecodedbyte (HANDLE fd, unsigned char c)
{
    unsigned char ch, cl, tmp;
    int sh, sl;
    unsigned int i;
    unsigned long delay;           // do delay to allow slow boot up flash

    // Encode byte
    tmp = c & 0xF0;
    tmp = tmp >> 4;
    ch = RFencodingTable[tmp];
```

```

    tmp = c & 0x0F;
    cl = RFencodingTable[tmp];

    // Send high byte
    sh = serial_writebyte(fd, ch);
    for (delay = 0; delay < 0x1FFF; delay++)
        ;
    // Send low byte
    sl = serial_writebyte(fd, cl);
    for (delay = 0; delay < 0x1FFF; delay++)
        ;

    for (i = 0; i < 0xFFF; i++)
        tmp = (unsigned char) sh;

    return (sh && sl);
}

#else // UNIX serial functions

/*
 * TCSANOW -          Make changes now without waiting for data to complete
 * TCSADRAIN -       Wait until everything has been transmitted
 * TCSAFLUSH -       Flush input and output buffers and make the change
 */

/*
// BAUDRATE : bps rate
// CRTSCTS  :      output hardware flow control
// CS8      :          8n1 (8bit, no parity, 1 stopbit)
// CLOCAL   :          local connection, no modem control
// CREAD    :          enable receiving character
// IGNPAR   :          ignore bytes with parity errors
// ICRNL    :          map CR to NL
// ICANON   :          enable canonical input
//
//                                disable all echo functionality and don't send
//                                signals to calling program

// Using Asynchronous Non-Canonical Serial Input Concept, because
//                                Canonical receives input with line transfer
//                                Non-Canonical receives input with fixed size transfer
*/
int
serial_open(char* port_name)
{
    int fd;        // File descriptor

```

```
// O_NOCTTY - not controlling terminal
// O_NDELAY - does not care the state of DCD line
// O_NONBLOCK - use instead of the old O_NDELAY
fd = open(port_name, O_RDWR | O_NOCTTY | O_NONBLOCK);

if (fd < 0)
{
    // unable to open port
    fprintf(stderr, "ERROR (serial_open) : Unable to open %s - \n", port_name);
}
else
    fcntl(fd, F_SETFL, 0);

return (fd);
}

void
serial_close(int fd)
{
    if (fd < 0)
        return;

    close(fd);
    fd = -1;
}

void
serial_init(int fd)
{
    struct termios options;

    // Get current options for the serial port
    // tcgetattr(fd, &options);

    // clear options for serial port
    bzero(&options, sizeof(options));

    // Set the baud rate for input and output
    cfsetispeed(&options, BAUDRATE);
    cfsetospeed(&options, BAUDRATE);

    // CONTROL OPTIONS
    options.c_cflag |= BAUDRATE;           // baudrate
    options.c_cflag |= CLOCAL;            // Local mode
    options.c_cflag |= CREAD;             // Enable receiver
    options.c_cflag &= ~CRTSCTS;          // Disable hardware flow control (old)
    options.c_cflag &= ~PARENB;           // No Parity
}
```



```

options.c_cflag &= ~CSTOPB;          // One stop bit only
options.c_cflag &= ~CSIZE;           // Mask character size bits
options.c_cflag |= CS8;              // Select 8 data bits

// LOCAL OPTIONS
options.c_lflag &= ~ICANON;          // Disable Canonical input (use raw input)
options.c_lflag &= ~ECHO;            // Disable echoing of input characters
options.c_lflag &= ~ECHOE;          // Disable echoing of erase character
options.c_lflag &= ~ISIG;           // Disable Signals
options.c_lflag = 0;                 // no local processing

// INPUT OPTIONS
options.c_iflag = 0;                 // no input processing

// OUTPUT OPTIONS
options.c_oflag = 0;                 // no output processing

// CONTROL CHARACTERS
options.c_cc[VMIN] = 0;              // blocking read until ?? character arrives
options.c_cc[VTIME] = 0;             // timeout in 0.1s unit

/*
options.c_cc[VINTR]                  = 0;          // Ctrl - c
options.c_cc[VQUIT]                  = 0;          // Ctrl - \ //
options.c_cc[VERASE]                  = 0;          // del
options.c_cc[VKILL]                   = 0;          // @
options.c_cc[VEOF]                    = 4;          // Ctrl - d
options.c_cc[VSWTC]                   = 0;          // '\0'
options.c_cc[VSTART]                   = 0;          // Ctrl - q
options.c_cc[VSTOP]                    = 0;          // Ctrl - s
options.c_cc[VSUSP]                    = 0;          // Ctrl - z
options.c_cc[VEOL]                     = 0;          // '\0'
options.c_cc[VREPRINT]                 = 0;          // Ctrl - r
options.c_cc[VDISCARD]                 = 0;          // Ctrl - u
options.c_cc[VWERASE]                  = 0;          // Ctrl - w
options.c_cc[VLNEXT]                   = 0;          // Ctrl - v
options.c_cc[VEOL2]                    = 0;          // '\0'
*/

tcflush(fd, TCIFLUSH);

// Set the new options for the serial port
tcsetattr(fd, TCSANOW, &options);
}

int serial_flush(int fd)
{
    tcflush(fd, TCIFLUSH);
}

```

```
    return 0;
}

int
serial_readbyte(int fd, unsigned char *ch)
{
    return (read(fd, ch, 1));
}

int
serial_writebyte(int fd, unsigned char ch)
{
    int status;
    unsigned char c;

    c = ch;

    status = write(fd, &c, 1);
    if (status <= 0)
    {
        fprintf(stderr, "ERROR (serial_writebyte) : failed writing single byte to port.\n");
        return 0;
    }
    else
        return 1;
}

int
serial_writestring(int fd, const char *str, int len)
{
    int status;

    status = write(fd, str, len);
    if (status < len)
    {
        fprintf(stderr,
            "ERROR (serial_writebyte) : failed writing %i characters to port.\n", len);
        return 0;
    }
    else
        return 1;
}
```

```
int
serial_readcodedbyte (int fd, unsigned char *c)
{
    unsigned char ch, cl, tmp;
    int sh, sl;

    // Get high byte
    sh = serial_readbyte(fd, &ch);
    // Get low byte
    sl = serial_readbyte(fd, &cl);

    // Decode byte
    tmp = RFdecodingTable[ch];
    tmp = (tmp << 4) & 0xF0;
    *c = tmp;
    tmp = RFdecodingTable[cl];
    *c |= (tmp & 0x0F);

    return (sh && sl);
}

int
serial_writecodedbyte (int fd, unsigned char c)
{
    unsigned char ch, cl, tmp;
    int sh, sl;
    unsigned int i;
    unsigned long delay;          // delay to allow slow boot up flash

    // Encode byte
    tmp = c & 0xF0;
    tmp = tmp >> 4;
    ch = RFencodingTable[tmp];
    tmp = c & 0x0F;
    cl = RFencodingTable[tmp];

    // Send high byte
    sh = serial_writebyte(fd, ch);
    for (delay = 0; delay < 0x1FFF; delay++)
        ;

    // Send low byte
    sl = serial_writebyte(fd, cl);
    for (delay = 0; delay < 0x1FFF; delay++)
        ;
}
```

```

    for (i = 0; i < 0xFF; i++)
        tmp = (unsigned char) sh;

    return (sh && sl);
}

#endif

```

The following code is the Visual C++ makefile for the VDebug application.

```

# Microsoft Developer Studio Generated NMAKE File, Based on vdebug.dsp
!IF "$(CFG)" == ""
CFG=vdebug - Win32 Debug
!MESSAGE No configuration specified. Defaulting to vdebug - Win32 Debug.
!ENDIF

!IF "$(CFG)" != "vdebug - Win32 Release" && "$(CFG)" != "vdebug - Win32 Debug"
!MESSAGE Invalid configuration "$(CFG)" specified.
!MESSAGE You can specify a configuration when running NMAKE
!MESSAGE by defining the macro CFG on the command line. For example:
!MESSAGE
!MESSAGE NMAKE /f "vdebug.mak" CFG="vdebug - Win32 Debug"
!MESSAGE
!MESSAGE Possible choices for configuration are:
!MESSAGE
!MESSAGE "vdebug - Win32 Release" (based on "Win32 (x86) Console Application")
!MESSAGE "vdebug - Win32 Debug" (based on "Win32 (x86) Console Application")
!MESSAGE
!ERROR An invalid configuration is specified.
!ENDIF

!IF "$(OS)" == "Windows_NT"
NULL=
!ELSE
NULL=nul
!ENDIF

!IF "$(CFG)" == "vdebug - Win32 Release"

OUTDIR=.\\Release
INTDIR=.\\Release
# Begin Custom Macros
OutDir=.\\Release
# End Custom Macros

ALL : "$(OUTDIR)\\vdebug.exe"

CLEAN :
    -@erase "$(INTDIR)\\serial.obj"
    -@erase "$(INTDIR)\\vc60.idb"
    -@erase "$(INTDIR)\\vdcb.obj"
    -@erase "$(INTDIR)\\vddisp.obj"
    -@erase "$(INTDIR)\\vdebug.obj"
    -@erase "$(OUTDIR)\\vdebug.exe"

"$(OUTDIR)" :
    if not exist "$(OUTDIR)/$(NULL)" mkdir "$(OUTDIR)"

CPP=c1.exe
CPP_PROJ=/nologo /ML /W3 /GX /O2 /D "WIN32" /D "NDEBUG" /D "_CONSOLE" /D "_MBCS"
/Fp"$(INTDIR)\\vdebug.pch" /YX /Fo"$(INTDIR)\\\" /Fd"$(INTDIR)\\\" /FD /c

.c{$(INTDIR)}.obj::

```

```

$(CPP) @<<
$(CPP_PROJ) $<
<<

.cpp{$(INTDIR)}.obj::
$(CPP) @<<
$(CPP_PROJ) $<
<<

.cxx{$(INTDIR)}.obj::
$(CPP) @<<
$(CPP_PROJ) $<
<<

.c{$(INTDIR)}.sbr::
$(CPP) @<<
$(CPP_PROJ) $<
<<

.cpp{$(INTDIR)}.sbr::
$(CPP) @<<
$(CPP_PROJ) $<
<<

.cxx{$(INTDIR)}.sbr::
$(CPP) @<<
$(CPP_PROJ) $<
<<

RSC=rc.exe
BSC32=bscmake.exe
BSC32_FLAGS=/nologo /o"$(OUTDIR)\vdebug.bsc"
BSC32_SBR= \

LINK32=link.exe
LINK32_FLAGS=kernel32.lib user32.lib gdi32.lib winspool.lib comdlg32.lib advapi32.lib
shell32.lib ole32.lib oleaut32.lib uuid.lib odbc32.lib odbccp32.lib kernel32.lib user32.lib
gdi32.lib winspool.lib comdlg32.lib advapi32.lib shell32.lib ole32.lib oleaut32.lib uuid.lib
odbc32.lib odbccp32.lib /nologo /subsystem:console /incremental:no /pdb:"$(OUTDIR)\vdebug.pdb"
/machine:I386 /out:"$(OUTDIR)\vdebug.exe"
LINK32_OBJS= \
    "$(INTDIR)\serial.obj" \
    "$(INTDIR)\vdcb.obj" \
    "$(INTDIR)\vddisp.obj" \
    "$(INTDIR)\vdebug.obj"

"$(OUTDIR)\vdebug.exe" : "$(OUTDIR)" $(DEF_FILE) $(LINK32_OBJS)
    $(LINK32) @<<
    $(LINK32_FLAGS) $(LINK32_OBJS)
<<

!ELSEIF "$(CFG)" == "vdebug - Win32 Debug"

OUTDIR=.\\Debug
INTDIR=.\\Debug
# Begin Custom Macros
OutDir=.\\Debug
# End Custom Macros

ALL : "$(OUTDIR)\vdebug.exe"

CLEAN :
-@erase "$(INTDIR)\serial.obj"
-@erase "$(INTDIR)\vc60.idb"
-@erase "$(INTDIR)\vc60.pdb"
-@erase "$(INTDIR)\vdcb.obj"
-@erase "$(INTDIR)\vddisp.obj"
-@erase "$(INTDIR)\vdebug.obj"
-@erase "$(OUTDIR)\vdebug.exe"
-@erase "$(OUTDIR)\vdebug.ilc"

```

```

        -@erase "$(OUTDIR)\vdebug.pdb"

"$ (OUTDIR) " :
    if not exist "$(OUTDIR)/$(NULL)" mkdir "$(OUTDIR)"

CPP=cl.exe
CPP_PROJ=/nologo /MLd /W3 /Gm /GX /ZI /Od /D "WIN32" /D "_DEBUG" /D "_CONSOLE" /D "_MBCS"
/Fp"$ (INTDIR)\vdebug.pch" /YX /Fo"$ (INTDIR)\\" /Fd"$ (INTDIR)\\" /FD -I /GZ -I ../../include /c

.c{$ (INTDIR)}.obj::
    $(CPP) @<<
    $(CPP_PROJ) $<
<<

.cpp{$ (INTDIR)}.obj::
    $(CPP) @<<
    $(CPP_PROJ) $<
<<

.cxx{$ (INTDIR)}.obj::
    $(CPP) @<<
    $(CPP_PROJ) $<
<<

.c{$ (INTDIR)}.sbr::
    $(CPP) @<<
    $(CPP_PROJ) $<
<<

.cpp{$ (INTDIR)}.sbr::
    $(CPP) @<<
    $(CPP_PROJ) $<
<<

.cxx{$ (INTDIR)}.sbr::
    $(CPP) @<<
    $(CPP_PROJ) $<
<<

RSC=rc.exe
BSC32=bscmake.exe
BSC32_FLAGS=/nologo /o"$ (OUTDIR)\vdebug.bsc"
BSC32_SBR= \

LINK32=link.exe
LINK32_FLAGS=kernel32.lib user32.lib gdi32.lib winspool.lib comdlg32.lib advapi32.lib
shell32.lib ole32.lib oleaut32.lib uuid.lib odbc32.lib odbccp32.lib kernel32.lib user32.lib
gdi32.lib winspool.lib comdlg32.lib advapi32.lib shell32.lib ole32.lib oleaut32.lib uuid.lib
odbc32.lib odbccp32.lib /nologo /subsystem:console /incremental:yes
/pdb:"$(OUTDIR)\vdebug.pdb" /debug /machine:I386 /out:"$(OUTDIR)\vdebug.exe" /pdbtype:sept
LINK32_OBJS= \
    "$(INTDIR)\serial.obj" \
    "$(INTDIR)\vdcb.obj" \
    "$(INTDIR)\vddisp.obj" \
    "$(INTDIR)\vdebug.obj"

"$ (OUTDIR)\vdebug.exe" : "$ (OUTDIR) " $(DEF_FILE) $(LINK32_OBJS)
    $(LINK32) @<<
    $(LINK32_FLAGS) $(LINK32_OBJS)
<<

!ENDIF

!IF "$(NO_EXTERNAL_DEPS)" != "1"
!IF EXISTS("vdebug.dep")
!INCLUDE "vdebug.dep"
!ELSE
!MESSAGE Warning: cannot find "vdebug.dep"
!ENDIF
!ENDIF

```

```
!IF "$(CFG)" == "vdebug - Win32 Release" || "$(CFG)" == "vdebug - Win32 Debug"
SOURCE=..\serial.c

"$(INTDIR)\serial.obj" : $(SOURCE) "$(INTDIR)"
    $(CPP) $(CPP_PROJ) $(SOURCE)

SOURCE=..\vdcb.c

"$(INTDIR)\vdcb.obj" : $(SOURCE) "$(INTDIR)"
    $(CPP) $(CPP_PROJ) $(SOURCE)

SOURCE=..\vddisp.c

"$(INTDIR)\vddisp.obj" : $(SOURCE) "$(INTDIR)"
    $(CPP) $(CPP_PROJ) $(SOURCE)

SOURCE=..\vdebug.c

"$(INTDIR)\vdebug.obj" : $(SOURCE) "$(INTDIR)"
    $(CPP) $(CPP_PROJ) $(SOURCE)

!ENDIF
```

# Appendix D

## Head Redesign

The head redesign by Damien Kee for GuRoo was made in an attempt to improve the overall aesthetic look of GuRoo, while being a little more “future-ready”. The main element, which is also the improved element to note on the new head design is the ability to accommodate stereo vision. While the current vision system is still a fair way from achieving coordinated stereo vision, the new head design supports such a development when it inevitably does happen. The other key feature to note is the position of the CSIRO eIMU in the centre of the head. This is intended to give the impression of a brain, as it is currently the closest GuRoo has to one. The green region towards the back of the head is where the vision boards will be mounted.

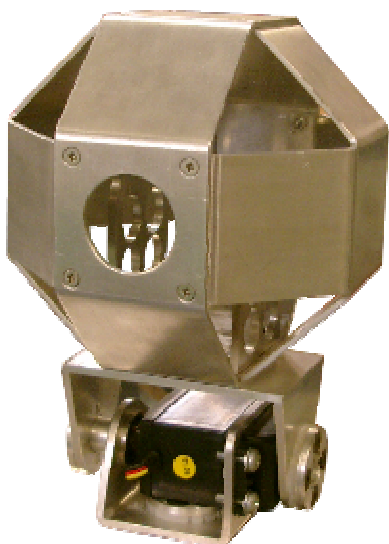


Figure 19 - GuRoo's Current Head

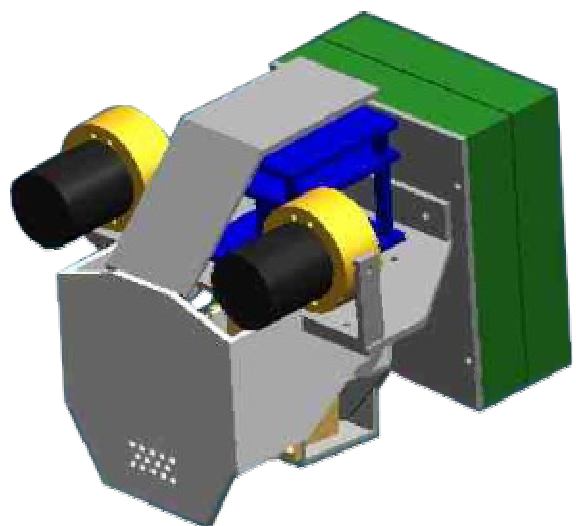
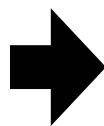


Figure 20 - GuRoo's Redesigned Head



## Appendix E

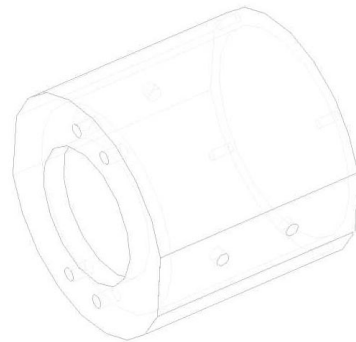
# Camera Housing Redesign

The objective of redesigning the camera housing is two-fold. The first, and primary reason, is to eliminate the ambient light that the old housing allowed to saturate the images captured by the vision system. The second purpose was to offer a greater depth for the spacers, which was required to obtain a correctly focused image.

The key points to note on the new camera housing are the extended depth, and the back seal. The technical work for the housing was completed by Damien Kee.



**Figure 21 - New Camera Housing**



**Figure 22 - Wireframe of New Camera Housing**