University of Southern Queensland
Faculty of Health, Engineering & Sciences

Real-Time Collision Avoidance for Programmable Machines from 3D Model Analysis

A dissertation submitted by

Rob Brooker

in fulfilment of the requirements of

ENG4111/2 Research Project

towards the degree of

Bachelor of Mechatronic Engineering (Honours)

Submitted: October, 2019

Abstract

Collision avoidance in programmable machines can reduce programming and setup time, and reduce the likelihood of needing to replace or repair parts during commissioning. While collision avoidance can be accomplished manually by a thorough analysis of the 3D model of the machine, and additional PLC code, this may protect the machine, but costs additional time, and is susceptible to human error.

The proposed system includes a computer program to export the 3D model of the machine, and a custom computer that is attached to the machine which manipulates the 3D model in real-time to detect approaching collisions. This computer signals the machine to stop when a collision is predicted. By performing interference detection on the 3D model of the machine, it effectively eliminates the possibility of human error, and saves the additional time that would otherwise be dedicated to the structural analysis and protection code mentioned above.

This project used a Raspberry Pi 3 Model B+ connected to the machine via a fieldbus link, to read axis positions and speeds from the machine PLC. It sends stop signals directly to servos when a collision is detected. From simulation testing it was determined that model complexity has a large effect on performance, but using a more powerful computer, and developing a better 3D model exporting algorithm could improve performance significantly.

Physical testing demonstrated accuracy and reliability, with reasonable response times. With limited optimization conducted for individual axes during testing, the performance of the test system showed great promise for further development, including an auto-tuning mode which will measure the dynamics of each axis to find the best response parameters for each. Work will continue on this system until a commercial product is realized.

University of Southern Queensland

Faculty of Health, Engineering & Sciences

ENG4111/2 *Research Project*

Limitations of Use

The Council of the University of Southern Queensland, its Faculty of Health, Engineering & Sciences, and the staff of the University of Southern Queensland, do not accept any responsibility for the truth, accuracy or completeness of material contained within or associated with this dissertation.

Persons using all or any part of this material do so at their own risk, and not at the risk of the Council of the University of Southern Queensland, its Faculty of Health, Engineering & Sciences or the staff of the University of Southern Queensland.

This dissertation reports an educational exercise and has no purpose or validity beyond this exercise. The sole purpose of the course pair entitled "Research Project" is to contribute to the overall education within the student's chosen degree program. This document, the associated hardware, software, drawings, and other material set out in the associated appendices should not be used for any other purpose: if they are so used, it is entirely at the risk of the user.

Dean

Faculty of Health, Engineering & Sciences

Certification of Dissertation

I certify that the ideas, designs and experimental work, results, analyses and conclusions set out in this dissertation are entirely my own effort, except where otherwise indicated and acknowledged.
I further certify that the work is original and has not been previously submitted for assessment in any other course or institution, except where specifically stated.

Rob Brooker

Acknowledgments

I would like to thank my employer, Mexx Engineering for providing the resources to develop and test my designs. In particular, Luis De Jesus for allowing me time from my daily duties to conduct experiments in support of my project. And Pedro De Jesus for always being prepared to take time out of his busy day to exchange ideas.

The greatest thanks goes to my wife, Jocelyn, who has provided love and support throughout my studies.

Rob Brooker

Contents

List of Figures

List of Tables

Chapter 1

Introduction

As programmable machines grow in complexity and capability, so too do the possibilities that they can damage themselves through self-collision. Mechanically timed machines protect themselves by their very design. A mechanically timed machine may have multiple axes which occupy the same space at various times, but through the use of gears and cams, it is ensured that they cannot occupy the same space at the same time. Modern programmable machines though, rely on programming to ensure that one axis cannot collide with another, and the fact that it is programmable, means that the program can be changed. This project was undertaken to create a fail-safe system which is able to anticipate an imminent collision and stop the axes responsible before impact. It needed to accomplish this without requiring any human analysis, either before, or during operation.

Programmable machines have advantages over mechanically timed machines in that they are less complex to design, by not needing all axes to be linked to a central source of motive force. They are more flexible, in that changing from one product to another may only require changing a recipe, rather than a tooling change for some products. And they are more tuneable, such that upon commissioning, a programmer can modify motion profiles, timing or even add or remove entire movements by adjusting code, where in a Mechanically timed machine, changing cam profiles or gearing to achieve even a little of this flexibility would be time consuming and costly.

1.1 The Problem

The advantages of programmable machines come at a cost. The machine will be designed with as many axes as are required to deal with the range of products that the customer specifies. These axes will have ranges of motion sufficient to accommodate all products, and their speed and acceleration capabilities will be specified to satisfy the most demanding case that the designer can anticipate. In most cases, this will produce several axes that could collide with one another with enough momentum to cause damage to the machine. Furthermore, because the machine is built to be flexible, the programming is often developed in an iterative way, rather than systematically with the mechanical design. This, and the need to tune on commissioning leads to a lack of built-in protection against collisions. As the responsibility to prevent collision is with the programmer, they must analyse the capabilities of all axes and keep them in mind while generating the motion profiles to be used. As complexity of machines increase, this task becomes more difficult. Also, they must consider what will happen when the same program is run with products of different dimensions, and characteristics.

There are 3 situations where collisions are likely, and they are:

- Initial testing when the program is run for the first time, and any programming mistakes will be discovered
- In production tuning, or new product setup, where changes are made without considering all collision scenarios
- Manual jogging operations by maintenance staff.

The risk of the first 2 situations can be reduced by the programmer studying the 3D model with the designer, to be aware of possible clashes. As the number of axes increases, it becomes increasingly difficult to be aware of all potential collisions at once. The second situation has the added disadvantage of occurring sometime after the machine has been commissioned, and so the programmer may have forgotten some of the potential collisions, or they may be new to the company, and have no prior knowledge of the machine. In the case of manual jogging, one must assume that a person might jog an axis straight into another axis without any consideration of the harm it may do.

1.2 The Solution

As modern machines are designed in 3D modelling software before they are built, these models can be used to determine when collisions will occur. In fact, interference detection is a part of most main-stream 3D modelling packages already. These features are used by designers to validate the capabilities of the machine during the design phase. If the model of the machine could be communicated to the machine such that it could move the various model axes as the machine axes move, then interference detection could be conducted on the model to warn the machine when a collision is about to occur. The advantage of this approach is that it eliminates the human factors that cause the 3 collision situations identified above. Provided the machine is built as an exact copy of the model that resides in the machine, then there should be no collisions at all. Even if the machine is changed, e.g. a revised part or assembly made to improve functionality. Provided the model in the machine is also updated, then the machine will still protect itself. This is particularly important, as physical changes to the machine's geometry may cause issues with the original program that was running before the change.

1.3 Research Questions

Before progressing with development of a particular solution it was necessary to conduct research to answer the following questions:

1. Are there any commercially available systems providing this type of protection, and by what methods?
2. Can the 3D model be exported in such a way that it can be read and manipulated by the machine? The model of the machine must be broken up into fixed and moving parts, and each moving part must be associated with an axis in the real machine.
3. Is there a method of interference detection which is fast enough, accurate and can ignore close proximities between parts that touch but do not collide, such as linear rails and their bearings; or ball-screws and their nuts?
4. The size of the model helps determine the requirements of the system. Can the model be loaded in the memory of the PLC controlling the machine? Or is separate

hardware required? If the model resolution is reduced to reduce the memory consumed, will the system still see collisions with enough precision to allow the machine to function correctly?

5. Response time is a combination of the interference testing time, the communications latency and the time it takes an axis to stop when instructed. What response time is acceptable to keep the collision avoidance system unobtrusive?

6. If processing must be performed on separate hardware, then what method can be found to pass the axis positions and speeds from the Programmable Logic Controller (PLC), to the collision avoidance processor, and status information from processor to PLC.

These questions will be answered in the following section with the view to creating a system complying with the project objectives:

- Self-collisions eliminated at all times while servos are in known positions
- Operation is transparent to PLC program, provided the program does not cause a collision
- Minimal human judgement required for setup and configuration, to avoid human error issues
- Easily adapt to different machines, or design changes of existing machine
- Able to be developed into a commercial product

Chapter 2

Literature Review

2.1 Existing Products

The closest match found to the desired system commercially available is ModuleWorks' software system called Collision Avoidance System (CAS) (ModuleWorks 2019). This system has limited documentation online at: www.moduleworks.com, but provides an interesting video introduction to the system. From the information available it appears that their software provides collision avoidance to Computer Numerical Control (CNC) machines by manipulation of 3D models of spindle adapters, tools, workpieces and fixtures. The software projects the moving parts 800 ms into the future of the CNC program to predict collisions. When the machine is being jogged, it restricts the speed of axes to ensure that collisions do not occur on the 800 ms projection. Communication with the CNC machine controller is achieved through the OPC UA protocol, which is a communication protocol for cross-platform communications in process control applications (Wikipedia 2019*a*).

2.2 Model Export

Siemens provides a Software Development Kit (SDK) which allows the user to write software which interacts with the Solid Edge program, and 3D model directly (Siemens 2019). With the Solid Edge SDK and a suitable programming language, the assembly model of the machine can be separated into moving, and fixed parts. The moving parts can

be associated with specific axes controlled by the PLC and the model parts exported in a suitable format for conducting interference detection. Solid Edge is capable of exporting models in a number of different 3D formats (Siemens 2010). Some of these are open standards which have their specifications freely available for download, while some are proprietary, and so their specification is not freely available. Two of the open standards are STEP, and STL. Both of these are easy to interpret and can be stored as plain text. This makes reading them into a target program simple. While STEP uses an object hierarchy which defines points on coordinates, then lines and vectors from points, then polygons from lines, etc (ISO 1994). STL uses a flat approach which describes the entire model as surface triangles (Wikipedia 2019*b*).

2.3 Interference Detection

Performing Collision Avoidance on 3D models requires finding suitable techniques for defining interference between objects in an efficient manner. (Shen, Jia, Chen, Wang & Sun 2015) detailed a set of vector algebraic formulas which can be used to detect interference between spheres, cylinders and cuboids. They use vector dot product, and normal vectors to object surfaces to determine if a part of one object is inside another. (Huang, Tang, Lou & Xiao 2014) also uses vectors to determine which side of a surface, a point lies on, but they take the determinant of a matrix constructed from 3 vectors to make the determination. For reduced execution time, (Shen et al. 2015) generalized the geometry of components into Oriented Bounding Boxes (OBB) with shapes of sphere, cylinder and cuboid.

(Shen et al. 2015) used a hierarchical decision structure where a pair of collision candidate objects are assessed to determine if the distance between their centre points is greater than the sum of their geometric dimensions. If so, then a collision is not possible, and analysis stops. Similarly, (Kwak & Park 2009) use the distance between centre points to identify one of 3 states (no chance of contact, some chance of contact, or certain contact). Neither of the 2 outer cases require further computation, but the middle case will require the detection of surface triangle interference.

(Huang et al. 2014) reduce processing time by only considering object pairs where at least one of the pair is currently moving. They also improved efficiency by employing a hier-

archical Axially Aligned Bounding Box (AABB) space subdivision binary tree structure, which benefits from the efficiency of AABB, but provides greater accuracy by recursive space subdivision where each level of bounding box is sub divided into 2 smaller boxes, which follow part geometry more closely. (Xing, Liu & Xu 2010) used sorted lists instead of binary trees. They found that the temporal coherence of most physical environments meant that taking lists of projections of AABBs onto each Cartesian axis and then applying a diminishing increment sort, meant that processing time was reduced by quickly finding projections that overlap on all three axes. (Zhiliang & Desheng 2011) used a hash function to map space grid cells to a hash table, and then perform tests when objects are both mapped to the same index in the hash table.

The final analysis in (Huang et al. 2014) is to detect the intersection of triangles, as surface elements of the 3D model. Where Huang et al (2014) positioned their space subdivision plane along the longest axis of the AABB such that there were an equal number of triangle centroids on either side of the plane, (Xing et al. 2010) chose the mid-point of the axis to position the plane, and the mid-point of the base of the triangle to determine which side of the plane it belonged to. (Kwak & Park 2009) divide geometry into small cuboid elements to ensure the accuracy of their collision detection, but they also group the elements that are rigidly connected to identify them as unable to collide, and so not assessed.

(Kwak & Park 2009) also performed cycle time analysis applying variations on their techniques to determine what was cost effective. They found that excluding analysis pairs that were both parts of a rigid assembly provided a small saving in cycle time, but detecting guaranteed collisions, and eliminating impossible collisions by the use of centre distances had a much more profound effect on cycle time.

An alternative to cuboid Bounding Boxes is to use spherical Bounding Boxes to greatly simplify the calculations to determine interference (Ouyang & Zhang 2012). They used these in combination with Octree structure which allows a cube to be recursively split into 8 equal child cubes. Each of these are then bounded by inner and outer spheres which are used for the interference detection.

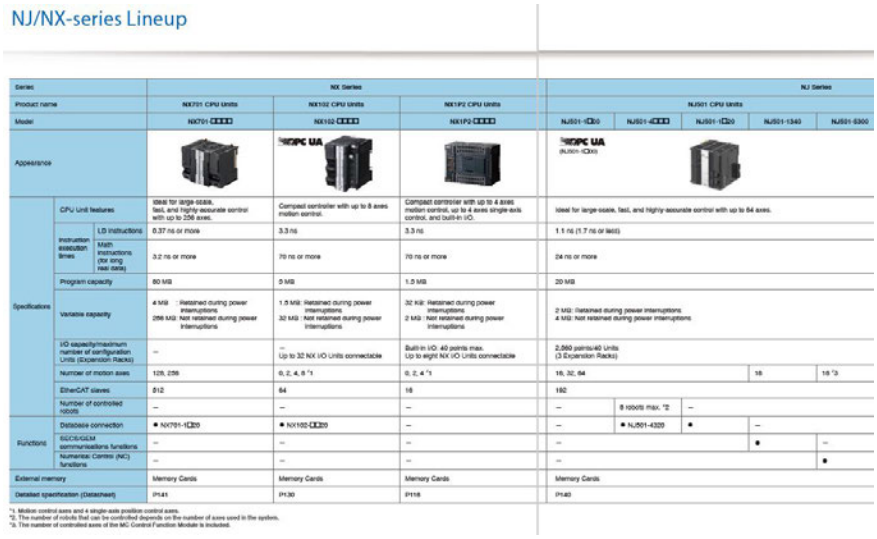
All of the preceding have used discrete time samples to detect collisions, but (Ping & Guang-long 2011), (Zhang & Liu 2015*a*), (Zhang & Liu 2015*b*) used interval interpolation to join discrete samples in order to detect collisions that may have occurred between samples. Their methods share some mathematical similarities to the vector dot product

method of (Shen et al. 2015), except that (Ping & Guang-long 2011), and (Zhang & Liu 2015*a*) construct their vectors from functions of time. This results in a polynomial in t, and the existence of a root in the examined time interval verifies that the 2 surfaces have at some point made contact. Solving for the root will identify the exact time that contact was (or will be,) made. To further improve efficiency (Zhang & Liu 2015*a*) used Taylor models to approximate the range of the polynomial in t over the period being assessed. This allows them to reduce the time taken to identify the existence of roots, and therefore the occurrence of a collision. Sturm Theorem was also used in (Zhang & Liu 2015*b*) as another method of detecting the presence of roots without having to solve a cubic equation. (Ping & Guang-long 2011) also used bounding boxes to reduce processing time, but after testing bounding boxes from the start and end of an interval, they then define a path bounding box which is the normal bounding box elongated along the path travelled during the detection period.

2.4 Memory Requirements

The original intent of this project was to have real-time analysis conducted within the PLC, so the system required no dedicated hardware, except an SD card containing the exported model. Early testing found a number of reasons why this would not be feasible. The first was the lack of memory available in the PLC, figure 2.1 below show excepts from the Performance specification sheets for Omron NJ501 PLC, which is the middle of the range product, and the NX701 PLC which is the top of the range. Note that while the NX701 provides 256 MB of variable storage for non-retained variables, the NJ501 has only 4 MB (Omron 2019). During early model export testing, an existing machine was exported as separate STL files for each axis, and one for the fixed parts of the machine. The average size for the moving assemblies was 104,000 triangles when exported with a 1mm resolution. As each triangle consists of 3 double precision floating point numbers, this accounts for 1.24 MB. And the entire machine model consisted of over $1.3\text{x}10^6$ triangles, or approx. 14.9 MB, it can be seen that, while the NX701 could have the entire machine loaded in memory, the NX501 would need to load, and unload model parts dynamically during execution. This would greatly increase processing time, and therefore reduce the effectiveness of the system. The next barrier to conducting the interference detection in the PLC was the lack of any ability to dynamically allocate memory during execution. Where in PC programming languages it is possible to allocate,

and free up memory as required, the PLC has its memory allocated at build time. This is a problem for the interference detection methods discussed above. Without the ability to dynamically allocate memory, it would be necessary to invent a complex coding system which allows a large chunk of memory to be allocated at build time, and then use a coding system which parcels out smaller subsections of the memory chunk to dynamically created objects within the analysis program. While this is possible, the work involved would be too much as an addition to the current project.



NJ/NX-series Lineup

Series			NX Series			NJ Series				
Product name			NX701 CPU Units	NX102 CPU Units	NX1P2 CPU Units	NJ501 CPU Units				
Model			NX701-□□□□	NX102-□□□□	NX1P2-□□□□	NJ501-1□20	NJ501-4□□□	NJ501-1□20	NJ501-1340	NJ501-5300
Appearance										
Specifications	CPU Unit features		Ideal for large-scale, fast, and highly-accurate control with up to 256 axes.	Compact controller with up to 8 axes motion control.	Compact controller with up to 4 axes motion control, up to 4 axes single-axis control, and built-in I/O.	Ideal for large-scale, fast, and highly-accurate control with up to 64 axes.				
	Instruction execution times	LD instructions	0.37 ns or more	3.3 ns	3.3 ns	1.1 ns (1.7 ns or less)				
		Math instructions (for long real data)	3.2 ns or more	70 ns or more	70 ns or more	24 ns or more				
	Program capacity		80 MB	5 MB	1.5 MB	20 MB				
	Variable capacity		4 MB : Retained during power interruptions 256 MB : Not retained during power interruptions	1.5 MB: Retained during power interruptions 32 MB : Not retained during power interruptions	32 KB: Retained during power interruptions 2 MB : Not retained during power interruptions	2 MB: Retained during power interruptions 4 MB: Not retained during power interruptions				
	I/O capacity/maximum number of configuration Units (Expansion Racks)		–	– Up to 32 NX I/O Units connectable	Built-in I/O: 40 points max. Up to eight NX I/O Units connectable	2,560 points/40 Units (3 Expansion Racks)				
	Number of motion axes		128, 256	0, 2, 4, 8 *1	0, 2, 4 *1	16, 32, 64			16	16 *3
	EtherCAT slaves		512	64	16	192				
	Number of controlled robots		–	–	–	–	8 robots max. *2	–		
Functions	Database connection		● NX701-1□20	● NX102-□□20	–	–	● NJ501-4320	●	–	
	SECS/GEM communications functions		–	–	–	–			●	–
	Numerical Control (NC) functions		–	–	–	–				●
External memory			Memory Cards	Memory Cards	Memory Cards	Memory Cards				
Detailed specification (Datasheet)			P141	P130	P116	P140				

*1. Motion control axes and 4 single-axis position control axes.
*2. The number of robots that can be controlled depends on the number of axes used in the system.
*3. The number of controlled axes of the MC Control Function Module is included.

Figure 2.1: Except from specifications for Omron PLCs (Omron 2019).

Chapter 3

Methodology

3.1 System Requirements

As this project was conceived as a sponsored development project, the systems used to create a solution are those systems used by the sponsoring company (Mexx Engineering). For machine design, Mexx uses Solid Edge from Siemens (ST8 throughout most of the project). PLCs and Servo drives used by Mexx are from Omron, and Omron's fieldbus (Industrial communications network) of choice is EtherCAT. So, to be successful the system must be able to read in a Solid Edge ST8 model and convert is to a suitable format. It must run on an Omron PLC, or on a separate device that is able to communicate with an Omron PLC over EtherCAT. It must be able to identify an imminent collision and cause an Omron Servo drive to stop before the collision occurs. If parts of the machine change in future, the system must be able to accommodate the changes by replacing the previously exported model, with a new export of the model of the machine. No additional user configuration should be required.

Having determined that external processing would be required for the interference detection, it became possible to use an existing 3D physics library to conduct the analysis. This was not possible on the PLC, as the proprietary structured text programming language of Omron PLCs cannot make use of libraries developed for PCs. It was then decided that the external processor must be capable of using these libraries to reduce development time. The device chosen to perform the analysis was a Raspberry Pi 3B+ (RPi) shown in figure 3.1, which has a 1.4 GHz ARM processor and 1GB of RAM. The RPi runs a

Linux variant operating system (OS) specific to the single board computer, known as Raspberrian. This OS provides a Windows like interface, and many open source development packages. The RPi also provides 40 physical IO pins, some of which were used to interrupt the Servo drives directly.

Figure 3.1: Photo of Raspberry Pi 3 Model B+ used for development.

Communication between the PLC and the RPi was necessary both to pass axis positions and speeds to the RPi for machine tracking, but also so the RPi can inform the PLC when it has stopped an axis. For Omron PLCs, this communication would be via EthernetIP, or EtherCAT. As the EtherCAT cycle-time on an NJ5 is 250 us, and the fastest Repeat Packet Interval supported by the EthernetIP master is 1 ms, EtherCAT was chosen. AB&T of Italy manufactures a product called EasyCAT Hat which is an EtherCAT fieldbus adapter card (figure 3.2) designed to attach to the Raspberry Pi.

An alternative to using external hardware was considered during the early phase of the project. This would involve developing a PC application which takes the exported model and moves every axis through its full range of motion multiple times in a brute force attempt to identify every possible collision scenario. This program would be run once, and the data generated would be transferred to the PLC as a lookup table for collisions. The simplest data structure considered consists of an n dimensional array of bit-strings of at least n Bits, where n is the number of moving axes. The bounds of each array axis would

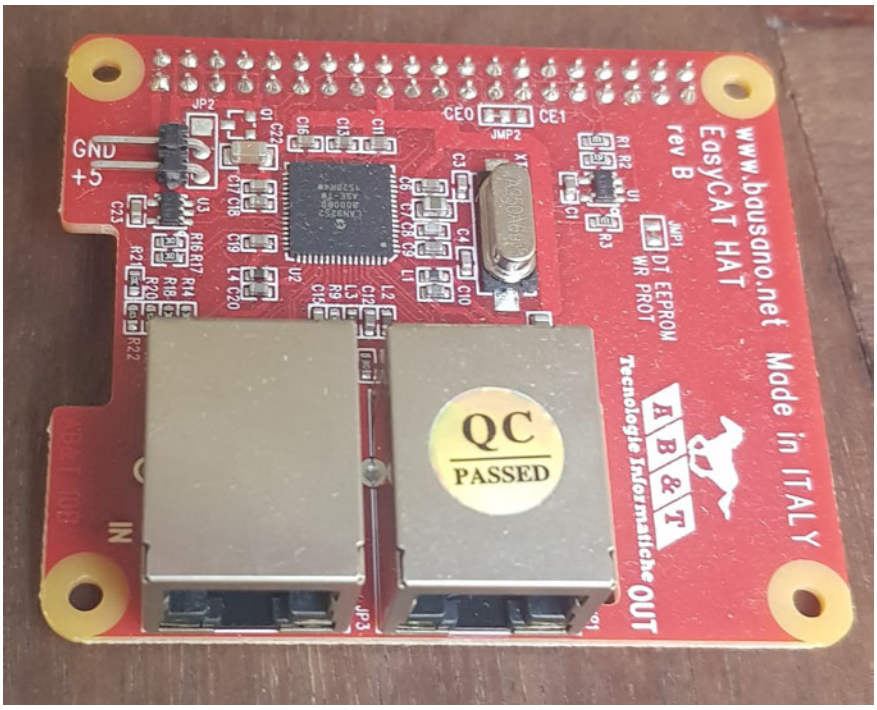


Figure 3.2: Photo of EasyCAT Hat from AB&T, for Raspberry Pi.

be the range of motion of the corresponding motion axis divided by the desired resolution. For example, a machine with 7 motion axes, with an average travel length of 500 mm, and a desired resolution of 1 mm, would require 500^7 x 1 Byte = approx. 7,105,427 TB of memory. Clearly this is not a viable solution due to memory use. A variant of the idea may yield a suitable solution. If axes that cannot collide with one another were excluded, then the memory requirement goes down dramatically. This would require a separate array for each axis, as one axis might collide with a different group of axes to another. If for example, each of the 7 axes could only collide with 4 others, then the memory requirement comes down to 198 TB. While still too large, it is possible to make other rules and range specifiers which can reduce memory requirement further. The brute force method would be very good in the PLC as this would only require a data lookup and so the response time would be excellent. To reduce the data size though, requires increasing processing complexity on the PLC side. With further work, a suitable compromise might be found where response time and accuracy are both optimized, and no external hardware is required, for the current project this path was abandoned as being unlikely to yield suitable results in a reasonable time.

The other advantage of going to external hardware is in the portability of the system. As it requires no programming in the PLC, only fieldbus configuration and the mapping of process data, it could be easily adapted to other PLC brands. The only change required

would be the use of a different fieldbus adapter attached to the RPi. Other fieldbus protocols such as EtherNetIP (Rockwell Automation), ProfiNet (Siemens) use a standard ethernet port which is included on the RPi, and would only require the protocol stack in order to communicate with these fieldbuses. The ability to make the system cross-platform was a reason for the decision to go with the external processor.

A small breakout board was built to gain access to the hardware IO pins which provide the Immediate Stop signals to the Servo drives. While the RPi devices used for development were run in a Windows environment for ease of use, the final version would run in a Command Line Interface (CLI) only mode to free up more processing power and memory. Also, for more complex machines with many axes, an industrial PC with faster processor, and more memory could be employed in-place of the Raspberry Pi. The only additional work required in the case the industrial PC case would be the requirement for a separate device to provide the digital outputs to the servo drives. Depending on the PC motherboard, this could either be a PCI expansion card, or a USB plugin device.

3.2 Model Export

To enable breaking up of the machine model into moving & fixed parts, it was necessary to add some variable fields to the parts and subassemblies within the model. These variables identify parts that move by specifying the axis number that will be used in the PLC, as well as the direction that the PLC axis will move the model in, and the position of the model part when the PLC axis is at zero. The model-exporting program was written in Visual Basic and made use of the ST8 SDK. It first makes a copy of the main assembly file and traverses the part tree looking for moving parts. When a moving part is identified, its filename is pushed onto a stack and the instance is deleted out of the assembly copy. The traverse is recursive down through the sub-assemblies until only fixed parts of the machine are left in the copy of the assembly, and all moving parts have been pushed onto the stack. The fixed part of the machine is then exported in STL format and some characteristics are written to a configuration file. The moving axes are popped off the stack one at a time and the process is repeated, where an assembly copy is taken and any moving parts in the tree are pushed onto the stack then deleted from the assembly copy. This second recursive function makes it possible to separate parts that are moved by other parts, so that a true representation of the system is the result. Each time a moving part model is

exported, its variables and its physical location within the parent model are written to the configuration file. At the end of processing the assembly copies are deleted, leaving the model exactly as it was before exporting began. The output is a configuration file which is used by the external processor to load the model parts and specify where they can move, and by which axis. And one STL file for each of the moving parts, plus one for the fixed parts of the machine. The record for each axis in the configuration file also identifies which other axis was the parent assembly from which this part was taken. This allows a movement hierarchy to be followed when part locations are being updated before interference detection. Appendix B contains a listing of the code used in the early phase of the project to export the model. Subsequent model exports were done manually after the company upgraded to Solid Edge 2019 which contains an SDK which is not backwards compatible. Rewriting of the model exporting program will be left for a later date.

3.3 3D Physics Library

A number of 3D physics libraries were considered for use, but only one considered allowed the definition of a complex shape through surface triangles and positions of objects to be controlled externally between time steps. SOLID version 2 created by Gino van den Bergen (van den Bergen 1999) provided this functionality and is open source. It also makes use of the QHULL library (Barber 2019) for a single pass broad phase analysis, in contrast to the Oriented Bounding Box, and Axially Aligned Bounding Box methods discussed in chapter **??**, this method essentially creates a wrap of the model to provide the minimum size convex body that encompasses all surface triangles of the part. This costs more in terms of processing time than OBB or AABB due to greater complexity, but reduces the chances of a non-collision test progressing to the detail phase. SOLID version 2 proved reliable in controlled tests to verify that it correctly identified interference of complex shapes and did not detect interference on interlacing complex shapes that did not touch.

3.4 External Processor Program

The program running on the Raspberry Pi first opens the configuration file which provides the instructions for importing the model parts. The model parts are imported as complex

shapes made up of surface triangles. Importing is done once, and then the model parts remain in memory. Then the program repeats the following sequence ad infinitum:

- Get current axis positions and speeds from PLC via fieldbus interface
- Update positions of complex shapes by loading transforms in SOLID 2 of the moving part and any parent parts that are recursively identified from information in the configuration file
- Run Interference test which performs callbacks if interference is detected
- Traverse motion hierarchy to identify all axes which may be responsible
- Set "Immediate Stop" bits of suspected axes
- Update status info to PLC via fieldbus interface

As 3D models of items such as ballscrews, and their nuts, would always be detected as interfering (models of male threads are larger than models of matching female threads), it was decided to exclude detection of interference between any part and its immediate parent part. This means that the system will not detect a collision when a linear axis runs into its own hard-stop. This type of collision though, is usually taken care of by use of limit switches which stop the servo if the axis is about to hit a hard-stop. The purpose of this project is to detect collisions that are too complex and dynamic to prevent by conventional means, so excluding basic single axis collision maintains the purpose, while mitigating the major obstacle of finding a way to allow penetration of certain parts but not others.

A optional command-line interface mode was also included in the program for conducting experiments when a real machine is not available to provide axis information. The command-line mode allows the user to specify the position of each moving part via the keyboard, then performs the interference detection tests, and reports the results, and processing time. Figure 3.3 shows an example a command-line session.

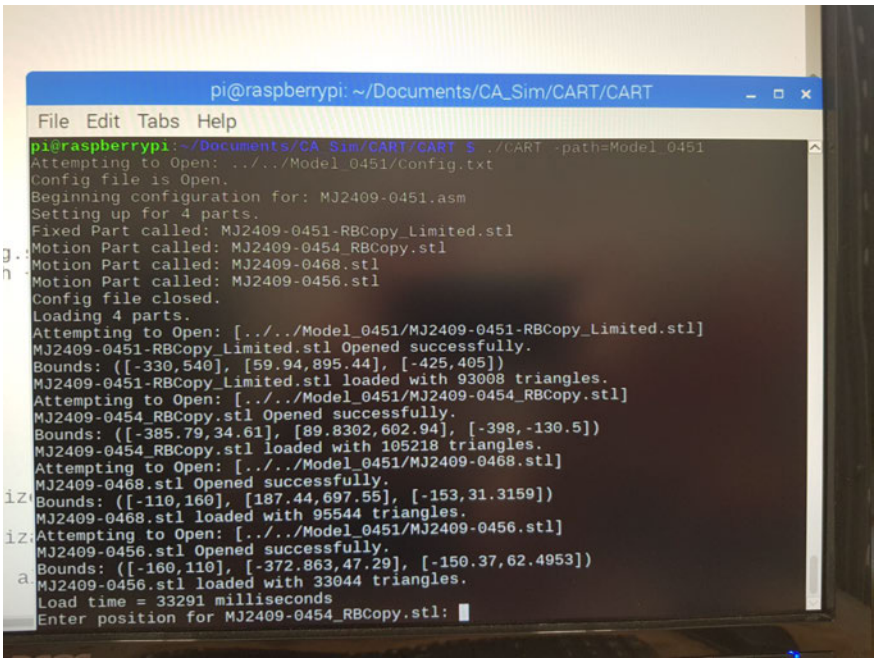


Figure 3.3: Screen shot of loading Collision Avoidance program in command-line mode

3.5 Test Cases

One of the fundamental measures of success for this project is the maximum permissible speed v_{m} (mm/s) that an axis can be allowed for a specified minimum clearance between axes s_{c} (mm). If the total response time of the system was t_{r} (ms), then the minimum clearance that the axis can be allowed is calculated by:

$$s_c = v_m\left(\frac{t_r}{1000} + \frac{v_m}{2d_e} - \frac{v_m}{2d_o}\right) \tag{3.1}$$

where d_o is operational deceleration, and d_e is emergency deceleration in mm/s^2.

For axes that must work in close proximity, the response time must be minimized. In some cases, even a zero-response time would not meet this requirement, and in these cases, this system could not be used in this form. For a system where the operational deceleration is equal to the maximum deceleration rate, the required clearance is calculated by:

$$s_c = \frac{v_m.t_r}{1000} \tag{3.2}$$

In order to test model loading time and testing time with varying complexities of machine, a simple analogy for a machine was created. This involved the creation of 3 parts, one

rectangle with rounded edges as a fixed machine part, then a sphere, and a toroid as moving parts, see figure 3.4. The variable complexity comes curtesy of changing the resolution of the export from ST8 to STL format. The model parts were exported with resolutions of 10 mm down to 0.001 mm. This effectively takes the model complexity from 536 triangles up to 3,588,512 triangles, while allowing the same series of axis positions to be applied for directly comparable results. The program in the RPi was then modified to allow the user to specify either axis position input from the command line, or fieldbus, and to choose the resolution of the model to be imported. The program was also extended to report the time to load each model, as well as how many triangles were imported. During testing it was also made to report which axes collide, and how long test processing took. The results table is shown and discussed in the Results section (chapter 4). Later the program and fieldbus interface specification were changed to allow the PLC to select the model resolution for testing. When the resolution selected by the PLC changes, the RPi turns on a Loading status bit in the fieldbus interface and unloads the existing model. It then loads the model for the new resolution and turns off the Loading bit. This feature was not necessary, as the resolution would not normally be changed during operation, but it did allow testing of the feedback part of the fieldbus interface, and could be used by the PLC program to avoid moving axes until the external processor has finished loading the model at startup.

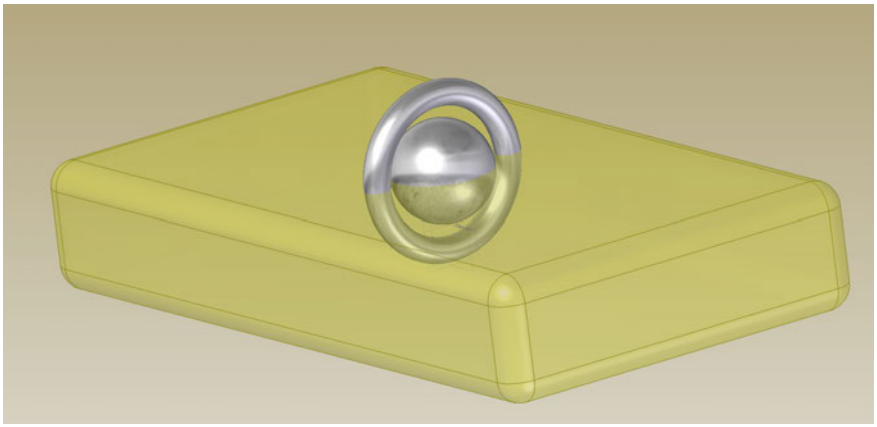


Figure 3.4: Simple model for testing functionally identical models with varying complexity

The next test case was to export the model of a simple real machine. Figure 3.5 shows the model that was exported for testing. This is a 3 axis machining head which was being prototyped at the time when this project was being conducted. The exported model consisted of 1 fixed model, and 3 moving models such that the Z axis moves on the fixed model, the X axis moves on the Z and the Y axis moves on the X. The complete model

consisted of 326,814 surface triangles and took 33 seconds to load on the RPi 3. Tests were run in command-line mode, where simulated axis positions are typed in to be tested and the results are displayed on a screen to be tabulated. Model exporting and position transformations were fine-tuned by moving the model parts in Solid Edge to the same positions as those entered on the command-line, and results were compared between the Solid Edge and the Raspberry Pi. This step not only ensured the accuracy of motion and geometry reproduction, but also served to validate the accuracy of the physics library's interference detection algorithm.

The RPi 3 was then connected to the physical machine in figure 3.6, where it communicates with the controlling PLC over the fieldbus, and the servos via digital outputs. Real-time operations were then conducted where the axis positions, and speeds are read from the PLC as the axes move, and collisions cause Immediate Stop signals to be sent to the relevant servos. To perform the real-time tests without the chance of damaging the machine, there were 2 versions of the Fixed part of the machine exported. One of the Fixed part models was just the real machine housing, and the other was the machine housing with a virtual obstacle around the spindle. By performing the same motion routine in the PLC with the different model configurations, it was possible to test the effectiveness of the motion interrupting function. Figure 3.7 shows the virtual obstacle (in red) which was placed in the path of the spindle. Table 4.3 lists the stopping distances from different directions and at different speeds during these tests.

In real-time operation, current Axis positions can be monitored in the PLC support software (Sysmac Studio). From the virtual obstacle version of the model, the actual axis positions where a collision will occur were measured in Solid Edge. A move command was then executed on the PLC which would drive the machining spindle into the virtual obstacle at speed. Once the Collision Avoidance system detects the collision, it commands the servos to stop, and the stopped position was then read from Sysmac Studio. By running PLC motion profiles which approach the virtual obstacle from different directions and speeds, the servo travel from the point of collision, here referred to as overshoot, could be measured. A set of parameters could then be determined to set maximum allowed speeds for each axis. The maximum speed will be a function of minimum required clearance between axes, the response time of the collision detection system and the stopping time of the servo. Experiments were conducted with the virtual obstacle attached to the front of the machining centre model. The motion tests consist of approaching the obstacle

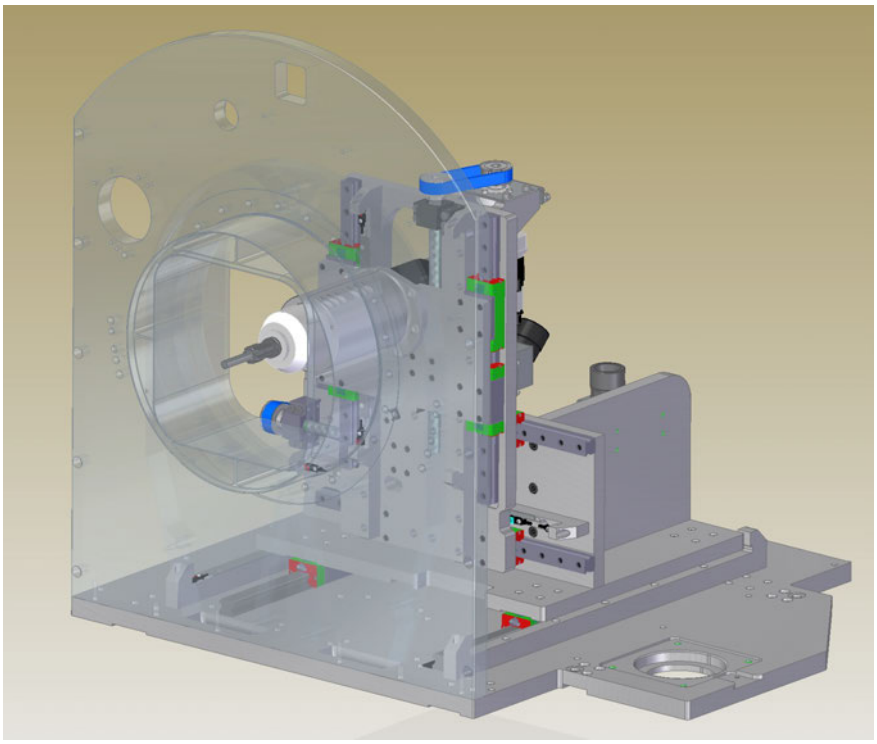


Figure 3.5: Simplified view of a real-world model exported for testing

from positive and negative in both X and Y directions. As the Y direction is vertical in this case, approaching in both positive and negative will show the influence of gravity on the response of the system. The X axis is horizontal and symmetrical, so it is not necessary to test from both directions, but in this case, it was done to verify consistency of results. These tests are reactive only, in that they require the axis to make contact with the obstacle before causing the servo to stop. In the real system, the collision must be predicted, and the servo stopped before the collision can take place.

3.6 Determine Response Time

Each scan cycle through the external processor, the positions of the model parts are updated, and the collision tests are performed. Once the response time of the system is known, the current speed can be accommodated by updating parts with a predicted position (P'), as in equation 3.3, rather than just the current position. This will mean that the tests are being performed for where the axes are expected to be at the end of the test, instead of where they were at the beginning of the test.

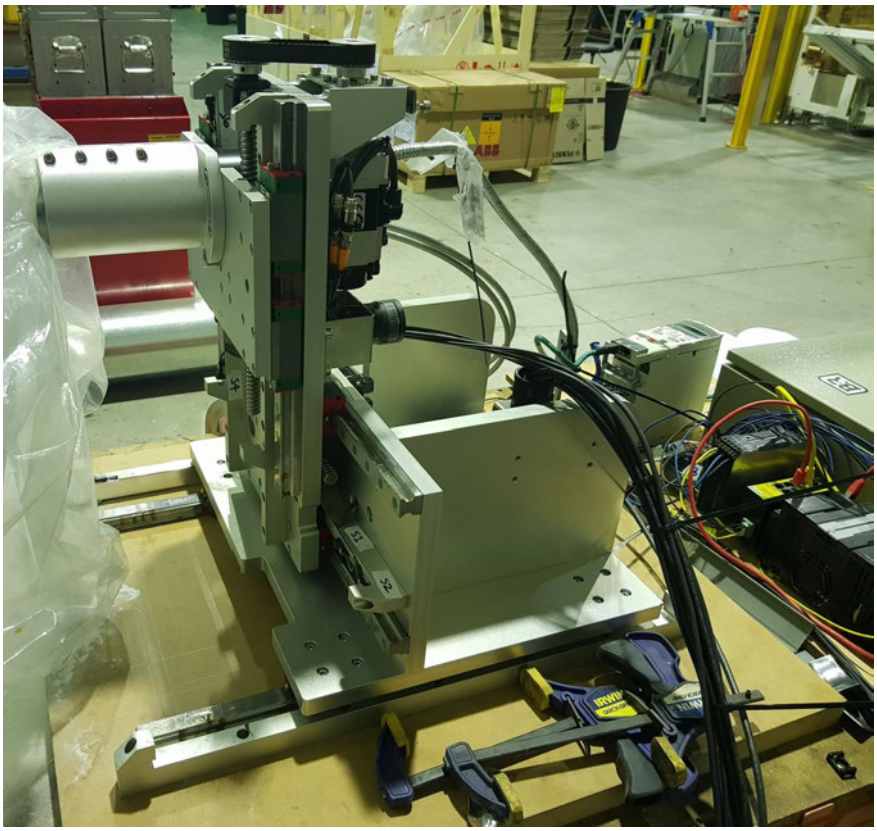


Figure 3.6: 3 Axis machining head prototype used for physical testing of Collision Avoidance system

$$P' = P + V.\frac{t_r}{1000} \tag{3.3}$$

where P is the current actual position and V is the actual velocity of the axis

For a commercial version of this system the experimentation being performed here cannot be required. An automated method for determining system limitations must be developed. A PC program would be written to communicate with the Collision Detector command line interface over a Secure Shell (SSH) connection. Over this interface the machine model can be loaded, then a phantom object mode could add virtual objects to clear areas of motion to test the responsiveness of the system. The machine programmer would specify minimum required clearance between axes and then run each axis at a number of different speeds into the phantom objects so the system could measure its response time. The program would then list the recommended maximum speeds for each axis before switching back to normal operation mode.

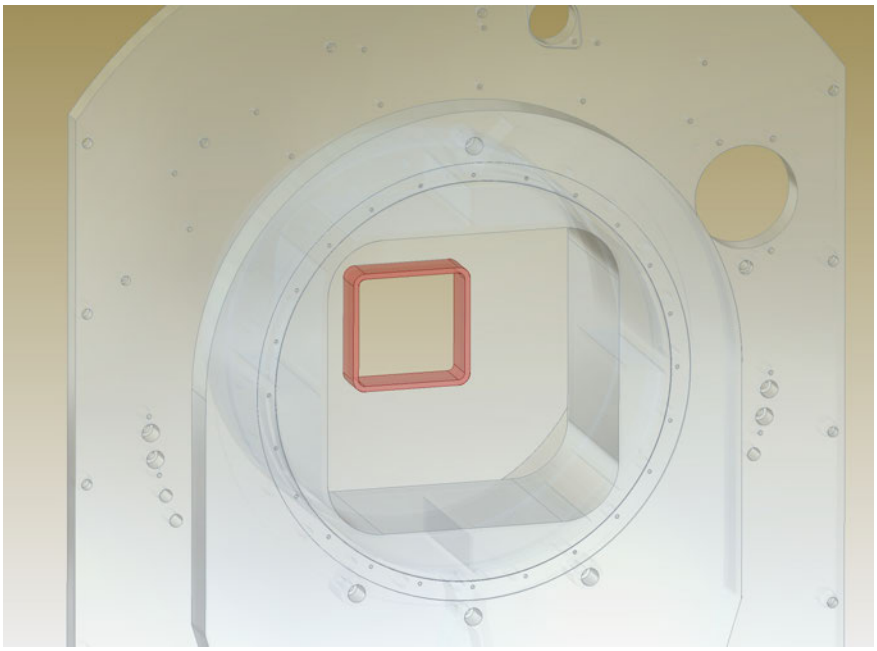


Figure 3.7: Virtual obstacle added to real-world model

3.7 Which Axis to Stop

In the real machine that was used for testing, collisions are usually encountered on the machining spindle, which is part of the Y axis, but is also moved by the X axis, which is in turn moved by the Z axis. In the configuration file for the model, each axis has an axis number, direction of travel and a parent axis. The parent is the part that this part is attached to. So, in the case of the machining centre tested, X is the parent of Y, Z is the parent of X, and the fixed part is the parent of Z. At the beginning of each cycle, when the positions are updated, each part is shifted in its direction of travel, to its own axis' current position. It is also moved by each of its ancestors, in their direction of travel. The Collision Detector program was only detecting collisions between parts, but not the direction from which the collision occurred. Not knowing the direction means that it is not possible to determine which servo axis is responsible for the collision. Not only the axis that collides but all of its moving ancestors will be stopped. A feature of SOLID v2 which has not yet been tested, is the ability to generate a normal vector to the plane on which the collision occurs. With this vector, it should be possible to selectively stop only axes which have a direction of travel which is not perpendicular to the collision normal vector.

Chapter 4

Results

4.1 Model Complexity Tests

Early tests centred around model size and processing time. Using surface triangles to describe an object works well, but the model data size grows very fast with curved surfaces and fine details in the model. This is because the surface triangles approximate the surface where the triangle must at all points be within a certain threshold distance (resolution) of the real model surface. For flat polygonal surfaces, the number of triangles required to describe it is at most one triangle for each edge, regardless of size. For a curved surface the size of the triangles is dictated by the radius of the curve, such that:

$$E_t = 2 \times \sqrt{(d+2)^2 - r^2} \tag{4.1}$$

where E_t is edge length of triangle, d is the allowable deviation from surface and r is the radius of the curve.

This formula can be used for surfaces with relatively large radius vs deviation. For cases with small radius, there is a maximum difference between the tangential angle at the curved surface, and the plane of the triangle. The default values when exporting from solid Edge are $d = 0.05$ mm, and $\theta = 30$ degrees. The number of triangles then increases with the occurrence of curves and with the number of faces in the model. Tests were conducted with a simple simulated model shown in figure 3.4, where the curved rectangle

Table 4.1: Load and processing times of RPi 3 for different model resolutions, with a range of axis positions

Raspberry Pi 3 Model B+ (1GB RAM)						
Resolution (mm)	10	1	0.1	0.01	0.005	0.001
Size on disk (kB)	147	842	7,917	77,102	153,552	977,712
# triangles	536	3,084	29,060	282,986	563,600	3,588,512
Load time (s)	0.103	0.335	2.73	27.4	56.1	368
Process time [0,0] (μs)	5,802	5,961	5,211	6,012	38,584	78,377
Process time [50,0] (μs)	2,192	3,340	1,740	2,860	37,655	77,661
Process time [150,0] (μs)	2,572	5,415	3,465	3,362	45,928	81.125
Process time [0,25] (μs)	3,780	5,426	3,059	4,426	37,082	75,352
Process time [0,100] (μs)	2,540	3,793	2,081	2,358	53,401	93,633
Process time [0,500] (μs)	603	1,642	513	674	11,430	17,465
Process time [850,0] (μs)	748	1,762	1,229	563	23,190	55,096
Process time [850,500] (μs)	83	828	349	81	87	86
Max Process time (ms)	5.80	5.96	5.21	6.01	53.40	93.63

was the fixed part, the sphere and the toroid were the first and second moving parts respectively. Table 4.1 contains the results of tests using varying resolutions to export the same models. This test the size of the models and loading times at the differing resolutions, as well as processing time for various combinations of axis positions.

Table 4.1 shows that as model complexity increase, load time increases linearly throughout the range. Process time stays flat until the model complexity gets above 283,000 triangles, and then it increases steeply. This was thought to be a result of the Raspberry Pi 3 running out of memory. During the development of this project the Raspberry Pi 4 came onto the market and was available with 2GB of RAM as compared to the RPi 3 which has 1 GB of RAM. The same set of tests were conducted with the RPi 4 and the results are shown in table 4.2. Figure 4.1 gives a graphical representation of the processing time difference between the 2 versions. In the graph the RPi 4 is consistently faster, but it still starts to increase at approximately the same data size. If it was simply a case of memory capacity, then the upturn should occur later in the graph. One possible reason for the similar result might be that the operating system restricts the total memory size available to a process regardless of the computer's memory capacity. Further investigation is required on this

Table 4.2: Load and processing times of RPi 4 for different model resolutions, with a range of axis positions

Raspberry Pi 4 (2GB RAM)						
Resolution (mm)	10	1	0.1	0.01	0.005	0.001
Size on disk (kB)	147	842	7,917	77,102	153,552	977,712
# triangles	536	3,084	29,060	282,986	563,600	3,588,512
Load time (s)	0.059	0.202	1.54	14.8	30.5	199
Process time [0,0] (μs)	2,955	3,221	2,774	3,063	20,502	41,578
Process time [50,0] (μs)	1,356	1,747	941	1,412	19,216	39743
Process time [150,0] (μs)	1,372	2,805	1,868	1,875	22,510	42,786
Process time [0,25] (μs)	2,064	2,815	1,657	2,200	19,120	39,112
Process time [0,100] (μs)	1,555	1,997	1,273	1,304	22,797	48,933
Process time [0,500] (μs)	337	746	304	403	5,768	9,125
Process time [850,0] (μs)	396	923	662	374	11,711	28,565
Process time [850,500] (μs)	46	409	407	50	50	50
Max Process time (ms)	2.96	3.22	2.77	3.06	22.8	48.9

issue.

The second notable outcome of these test is the confirmation that the SOLID 2 library is performing a broad-phase cull before processing detailed models. This can be seen by the differences in the processing time depending on the positions of the 2 moving parts in each test. In the image in figure 3.4, both moving parts are at their zero positions. The SOLID 2 library is first using a simplified version of the model parts, such as a bounding box, to determine if it is possible for 2 objects to collide. If the bounding boxes do not collide, then the detailed models cannot collide, and so no further testing is required for that pair of objects. If the bounding boxes do collide, then the detailed model must be tested. From the test results in both computers, and at all resolutions, the last test case is similar, and very short. This seems to verify that the only test being performed is on bounding boxes whose complexity is independent of the detailed model complexity. Another observation from the tabulated results is the limiting axis positions case for different resolutions. In each of the first 4 resolution trials, the first set of positions [0, 0] requires the longest processing time, but in the 2 most complex model tests, it is the [0,100] position case that requires the most time. This seems to demonstrate the principle that once a collision is

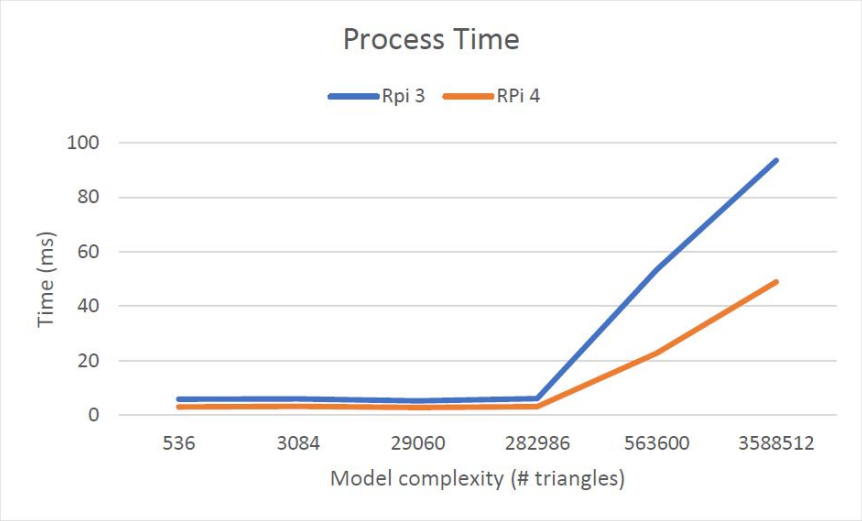


Figure 4.1: Comparison of processing times between models 3 & 4 of Raspberry Pi

confirmed, no more processing is required. Both moving objects definitely collide with the fixed part, so once confirmed, processing is done, but the sphere and toroid do not collide, even though their bounding boxes suggest that they might. Every pair of triangles between the 2 objects must be checked to find if there is a collision, but because there is not, the processing takes the maximum time. As the model complexity increases, the sphere and toroid have many more triangles and so the processing time increases with the product of the number of triangles of each part.

4.2 Real Model Tests

From the previous tests, it was believed that a complete model consisting of around 300,000 triangles would achieve a processing time of less than 10ms on the RPi 3. This was thought to be marginal for dynamic systems, but acceptable. The 3D representation of a real machine being built at the time, was exported for real-world testing. The exported model consisted of 3 moving axes, and one fixed machine frame, totalling 326,814 triangles. The expectation from the previous tests was that it would take approx. 30 seconds to load, and between 7 – 10 ms to process each time step. The reality though was surprising, while loading was as expected, at 33.04 seconds, process each time step took approx. 70 ms on the RPi 3. This processing time was similar to a model with 10 times the triangles from the previous test cases. Simulation of axis positions such that all bounding boxes would be clear of each other, still yielded comparable processing times to the previous

tests, so a non-testing related delay could be eliminated. One factor affecting testing of real assemblies is that when one axis is carried by another, they will usually be joined by some form of bearing, for support, and a drive mechanism, such as a ballscrew or belt and pulleys. These interfaces will most likely be detected as collisions during testing and will definitely register as potential collisions in the broad-phase analysis. The model in the real model tests consisted of a fixed part carrying a Z Axis, then the Z Axis carrying the X Axis, and the X Axis carrying the Y Axis. The Fixed part and Z Axis are always touching, Z and X Axes are always touching, and X and Y Axes are always touching. Furthermore, if the broad phase uses simple rectangular bounding boxes, then Axes X and Y would also always be candidates for collision with the fixed part of the machine. The SOLID 2 documentation states that it is possible to specify certain pairs of objects as candidates for collision and ignore the proximity of others. In the current machine there are a total of 6 possible pairs for collision analysis, and 5 of these will always proceed beyond the broad phase. If the pairs of axes that are attached to one another were excluded, the number of total pairs would be reduced to 3, and only 2 of these would always pass beyond the broad phase. Assuming processing time is similar for each pair, this should reduce total processing time from 70 ms to approx. 35 ms. Upon changing the Collision Detector program to only consider certain object pairs, the processing time was reduced by more than anticipated. The time to perform a test ranged from 5 ms to 18 ms, depending on axis positions.

4.3 Total System Response

The next tests required the addition of a virtual obstacle in the model of the real machine. In these tests the Collision Detector is placed in real-time mode where the PLC is sending current positions and velocities for each axis every 1.25 ms (1 ms PLC scan time & 250 μs EtherCAT comms cycle). The Collision Detector moves the loaded model parts to the current positions as reported, and then tests for interference. When a collision is detected, a group of digital outputs which are connected to Immediate Stop inputs on the servo drives, signal that the servos must stop. This puts the servos into an Emergency Stop state, and the actual stopped position of the servo is read from the PLC support software. For the tests listed in table 4.3, a series of runs were made at the virtual obstacle in each of 4 directions. For each direction 6 runs were made at speeds ranging from 10 mm/s to 100 mm/s.

Table 4.3: Overshoot results in reactive collision tests

Overshoot (mm)						
Direction	Speed (mm/s)					
	10	20	40	60	80	100
X+	0.25	0.47	0.80	1.39	1.97	2.16
X-	0.12	0.27	0.46	0.91	1.61	2.16
Y+ (up)	0.18	0.35	0.53	1.09	1.32	1.90
Y- (down)	0.34	0.41	0.92	1.70	2.5	2.74
Avg (mm)	0.22	0.38	0.68	1.27	1.85	2.24
Max (mm)	0.34	0.47	0.92	1.70	2.50	2.74

Overshoot in table 4.3 above is the difference in position at which the collision occurs, verses the position at which the servo comes to a standstill. The overshoot test results demonstrate a predictable increase in collision overshoot as travel speed increases. There are 2 independent components to the overshoot distance, the first is made up of the following factors:

- the PLC Scan time required to read current positions, and velocities from the Servos, and write it to the Collision Detector Process Data Objects,
- EtherCAT communications cycle,
- Collision Detector communication exchange with the fieldbus adapter,
- Collision Detector interference test time
- Collision Detector communication exchange with device hardware outputs,
- Switching time for level shift from 3.3v logic to 24v logic,
- And any other non-testing code being executed in the Collision Detector such as writing monitoring information to the console.

Due to the fact that tests are performed on discrete positions each time, it is possible that a collision occurs immediately after a test cycle has started and so it is missed in the current cycle, and will not be actioned until it is detected in the next cycle. For this reason, this first component can be anywhere from one to two scan times and this

time is all passed at the commanded axis velocity. The second component is the servo stopping time. For the machine being tested, the maximum operational deceleration rate is 10,000 mm/s^2. From 10 mm/s it will take 1 ms to stop and from 100 mm/s it will take 10 ms to stop. Both of these components of the overshoot should be considered in the compensation section. Figure 4.2 shows the overshoot curve for the reactive tests.

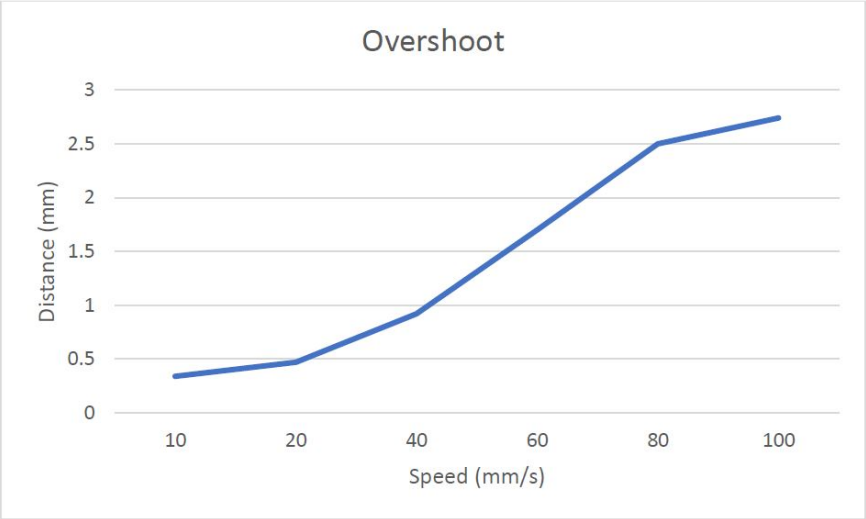


Figure 4.2: Overshoot curve for reactive collision detection tests

4.4 Avoiding Collisions

The next step in the implementing the system is to have it stop servos before they collide, rather than once they have collided. By moving the model parts to positions they are expected to occupy in the future, instead of where they are now, a future collision can be detected. Knowing how far ahead to look was the purpose of the analysis of the previous tests. By adding the distance that the axis would move in the time it takes to perform a test, to the current position, the calculation can be based on the positions of the axes as they are when the test is finished, and the stop signal can be sent immediately. With this method it is expected that a small amount of overshoot could still occur, as it would be equivalent to projecting the axes forward by the distance that they would travel in the testing time. By adding 2 x the distance travelled in the test time, should prevent most collisions, and adding an additional buffer for the servo stopping distance should guarantee no collisions occur. The problem with this predictive model is that it may result in false fails. These could occur when an axis is programmed to move quickly to a position, very close, but not touching, to another object. During the move,

the Collision Detector will be projecting the moving axis along its travel path, and if the speed and deceleration are high, the Collision Detector may see a collision as being imminent and stop the servo. The shorter the testing time, the less an axis needs to be projected ahead to avoid collisions, and so the more dynamic, the machine can be. The difference between the max operational deceleration rate, versus the emergency stop deceleration rate, and the minimum clearance between axes, are the 2 parameters which need to be balanced to avoid false collision detections. If false triggering occurs, either the operational deceleration rate must be reduced, or the minimum clearance must be increased, as illustrated in eq 3.1.

Another problem with this system is that the axis is being projected assuming the velocity is constant. If an axis which is approaching another is accelerating, the projection may not indicate the collision until it is too late. Likewise, if an axis is moving away from the path of another, and decelerating, it may not be out of the way when the other axis arrives. One option to deal with these problems is to perform the collision tests twice each scan, once with current actual positions, and once with projected positions. This doubles the execution time and could only be considered if significant improvements can be made in testing times by other means. Another option which could have a similar result is to elongate the surface triangles in the model loaded in memory. Applying a scaler transformation to all of the triangles making up a model part, where the direction of travel is stretched by the distance that the part would move in one or two test times at the current speed. If the transformation could be applied quickly enough, this may be faster than running the test twice, and if the transform only needs to occur on occasions where the velocity changes, then it would have even less impact on test time.

4.5 Velocity Compensation Tests

A number of trials were conducted implementing equation XX with various response time values. As predicted, setting response time at or below the exact testing scan time resulted in mostly missed collisions, but still the occasional hit. A value of 40ms yielded the results in table 4.4, and figure 4.3.

Undershoot in figure 4.3 is found by the same means as overshoot in the previous figure. Here however the negative value shows the clear distance between the axis and obstacle

Table 4.4: Undershoot results in predictive collision tests

Overshoot (mm)						
Direction	Speed (mm/s)					
	10	20	40	60	80	100
X+	-0.02	-0.33	-0.41	-0.98	-0.64	-0.75
X-	-0.26	-0.50	-0.63	-0.83	-1.40	-1.17
Y+ (up)	-0.23	-0.48	-0.84	-1.36	-1.6	-2.01
Y- (down)	-0.10	-0.20	-0.49	-0.38	-1.22	-1.37
Avg (mm)	-0.15	-0.38	-0.59	-0.89	-1.22	-1.33
Max (mm)	-0.26	-0.50	-0.84	-1.36	-1.60	-2.01

Figure 4.3: Undershoot curve for collision avoidance tests with $t_r = 40$ ms

after the axis comes to a standstill. While these figures confirmed that collisions can be avoided, the undershoot values may require further tuning for some machines. From analysis of the data in table 4.4, and knowledge of the mechanical characteristics of the machine being tested, it can be seen that when an axis has faster emergency deceleration rate, it produced greater undershoot values. For each run at speeds of 40 mm/s, or more, the +Y direction sees the greatest undershoot by a significant margin. This may be explained by the fact that gravity assists to decelerate the axis, and so the emergency stop performance is greater than the other directions, yielding a larger undershoot. More thorough tuning of velocity compensations for various axes may produce more consistent results.

The worst case in the results above is for the highest speed. With a 40 ms response time compensation, at 100 mm/s the axis is being projected 4mm into the future. If the

PLC program was sending the axis to a position 1mm from collision at 100mm/s, with a deceleration of 10,000mm/s, the motion controller would keep the servo at full speed until it was 1.5mm from the collision, and then decelerate to stop at 1mm. The Collision Detector would intervene when it detects an imminent collision at 4mm, and the axis comes to a stop at 2mm, errored in the Emergency Stop state. This example is extreme, and seems unlikely, but in larger, faster machines, axes can move many times faster than the one tested. A 40ms velocity compensation time applied to an axis traveling at 2.5m/s would be projecting the axis 100mm ahead of its current position. In most cases, very high acceleration rates would not be used with multiple axes in close proximity, but for a system like this to gain mainstream acceptance, it must account for these situations.

Chapter 5

Further Work

5.1 Tuning The Undershoot

From the undershoot curve in figure 4.3, it appears that the undershoot distance is linearly related to the axis speed. Further trials might be useful to create a more dynamic velocity response time, than a fixed value. If a function of velocity and max processing time can be found to minimize undershoot, while guaranteeing collision avoidance, then the system will become more universally acceptable.

5.2 Model Velocity Elongation

In the current program, each model part is simply moved to the position that it is expected to occupy at the end of the current test. This means that another axis passing behind this one may collide because it thinks this one has already moved out of the way. An alternative is to stretch the model parts in the direction of travel, by the distance they are expected to travel during the test. SOLID v2 has a Scale function which can scale all vertices in any cartesian direction. Trials will need to be performed to determine the effect these scaling functions will have on the position of the model, and the processing time required. Also work will need to be done on determining the effect that this has on axes when their parent axes are moving also.

5.3 Support Software

Setup, tuning and configuration software will need to be created such that they do not require intimate knowledge of the system for the operator to use. Automatic tuning functions could be performed in a PC connected to the system, with a graphical representation of the machine. The operator could place virtual obstacles in the 3D representation for response time tuning, where the program handles all of the calculation and setting parameters. Alternatively, the program could add virtual obstacles itself, and request the programmer to perform servo motions towards the virtual obstacle. Parameters which are currently hard coded would be shifted to a parameter file which is loaded at startup.

5.4 Physics Library Improvements

A certain proportion of the total response time is due to the interference detection testing. The methods used in the current setup may not be the optimal for this application. Some further investigation would determine the proportion of time due to the testing, and if it is considered high, then other methods might be trialled. Using a multi-step broad phase may be effective in reducing test time for machines with complex parts whose bounding boxes will overlap most of the time. Limiting the pairs to be tested on each scan to only those pairs with at least one member currently in motion could be another way to reduce the processing time. Plus eliminating pairs based on them never being able to collide would also help for very large complex machines. The current model exporting program does not identify axis limits, so the collision detection algorithm currently considers each axis to have an infinite range. While only broad phase tests would be performed on pairs that can not possibly collide, each of these tests may account for microseconds, and as the number of axes increases, this could become significant.

5.5 Rotational Axes

The current system only deals with linear motion axes, but rotational axes are also regularly found on machines. Particularly universal robots typically consist of 6 rotational axes in series with one another. Presently the variables contained in the model parts, the

exporting program and the collision avoidance program only consider linear motion, but rotational motion would need to be added for the system to be commercially successful. Changes required to the model exporting program include describing the axis of rotation with respect to the machine coordinate system as a quaternion. The current physics library supports model rotation by quaternion, so model translation can be achieved by forward kinematic means. One limitation is in the planed elongation of model parts in the direction of travel with respect to velocity. While for linear axes, this can be achieved by scaling the model in the direction of travel, it is not simple to do the same for a rotating axis.

5.6 Dynamic Product or Workpiece

Once the rotational axes are implemented, the system becomes more useful for applications such as robotic cells, like painting and welding, or pick and place of various products. If the product, or workpiece was also included in the model, and identified as a transient part that did not always exist, then a transient identifier field could be added to the fieldbus which specified the current product or workpiece in the cell. This would allow collision avoidance in all situations, even when a new welding gig, or fixture was added. Before the robot programmer teaches the new welding job (for instance) they would update the 3D model to include the new fixture and parts, with their transient identifier being the job number. During the programming and production phase the collision avoidance system will be watching everything else, while the programmer is focused on the welding torch.

Chapter 6

Financial Considerations

The cost of the project in terms of hardware was quite low, with purchase of Raspberry Pi, EasyCAT Hat and sundry electronic components totalling less than $250. For an experimental solution this is cost effective, but a commercial version would have different requirements. Professional electronics, and an industrial PC with custom enclosure would cost thousands of dollars. For a commercial venture, development time would be amortized over the expected number of sales, but a profit margin would also be expected. A conceivable price range might be $2,000 – $10,000 depending on the number of axes needing to be monitored.

Chapter 7

Conclusions

This project has successfully achieved the goal of providing "Collision Avoidance in Motion-Control Systems by Real-Time Predictive Interference Detection of 3D Assembly Models". It has shown that with minimal human input it is possible to have the machine protect itself, and that this protection can be maintained even in the face of changes to the machine, provided the 3D model from which the machine was built, has also been updated. As an experiment is it successful, but as a commercial proposition its future is uncertain. While no official data on the cost to a business from machine self-collision has been found, from my personal experience of commissioning and supporting 26 machines over a 6 year period, I estimate that repairs due to collisions that this system could have prevented, would total no more than $5000. This averages out to less than $200 per machine, which is far less than the commercial version of the system would be expected to cost.

The main beneficiary of a system like this is the programmer while creating or modifying the program that operates the machine. With all of this taken into consideration, it seems that another option exists. If the company building machines invests in one Collision Avoidance unit able to monitor the largest machine that they expect to make, then this one unit could be reused over and over on machine after machine, where it is connected to the machine for initial programming and testing, and then removed when the machine is handed over to the customer. The same unit then stays with the manufacturer to be used again when the next machine is being built, at which time the previous loaded model is replaced with the model of the next machine. An added benefit to this approach is

that if the customer sees the value in the system and wishes to buy it, then it could be left on a machine after handover, and the manufacturer then buys a new unit for future projects. If the customer does not choose to buy the system, but at some time in the future, they ask for changes to the machine, program, or commissioning of a new product; then the programmer reloads the machine model, reattaches the device to the machine and performs the changes. Once the changes are successfully tested, they can remove the device again. The likelihood of the machine manufacturer agreeing to purchase one unit that they can use to protect all machines that they make is far greater than that of selling a unit with each machine. This however reduces the expected sales numbers and so increases the amortized development cost per unit expected to be sold. Fortunately, the development cost in this case is labour being donated by the programmer, so development cost is not expected to prevent the project from advancing.

With the performance improvements identified in chapter 5, and transferring the system to a higher performance hardware platform, I expect that this system could be made effective for the majority of programable machines designed, and built by oem machine manufacturers.

References

Barber, C. (2019), 'Qhull manual', http://www.qhull.org/html/index.htm.

Huang, R., Tang, T., Lou, Y. & Xiao, M. (2014), 'A collision detection algorithm of Robot in off-line programming system', *2014 4th IEEE International Conference on Information Science and Technology* (4), 349–53.

ISO (1994), Industrial automation systems and integration – Product data representation and exchange – Part 21: Implementation methods: Clear text encoding of the exchange structure, Standard ISO-10303-21, International Organization for Standarization.

Kwak, H. & Park, G. (2009), ''Module-based efficient self-collision detection method for humanoid robots', *2009 IEEE Student Conference on Research and Development (SCOReD)* pp. 483–6.

ModuleWorks (2019), 'Simulation Software Components', https://www.moduleworks.com/software-components/simulation/#cas.

Omron (2019), 'OMRON - Machine Automation Controller - NJ/NX Series', http://www.omron.com.au/data_pdf/cat/nj_nx_p089-e1_7_1_csm1042739.pdf?id=3435.

Ouyang, F. & Zhang, T. (2012), 'Octree-based spherical hierarchical model for collision detection', *the 10th World Congress on Intelligent Control and Automation* (10), 3870–5.

Ping, Z. & Guang-long, D. (2011), 'A fast continuous collision detection algorithm based on K_DOPs', *2011 International Conference on Electronics, Communications and Control (ICECC)* pp. 617–21.

Shen, Y., Jia, Q., Chen, G., Wang, Y. & Sun, H. (2015), 'Study of rapid collision detection

algorithm for manipulator', *2015 IEEE 10th Conference on Industrial Electronics and Applications (ICIEA)* (10), 934–8.

Siemens (2010), 'Saving Solid Edge documents to other formats', `https://www.soliddna.com/SEHelp/ST5/EN/i_v/save2a.htm`.

Siemens (2019), 'Solid Edge 2019 SDK', `https://docs.plm.automation.siemens.com/docs/se/2019/api/webframe.html`.

van den Bergen, G. (1999), 'User's Guide to the SOLID Interference Detection Library', `http://solid.sourceforge.net/solid2_toc.html`.

Wikipedia (2019*a*), 'OPC Unified Architecture', `https://en.wikipedia.org/w/index.php?title=OPC_Unified_Architecture&oldid=916172216`. [Online; accessed October-2019].

Wikipedia (2019*b*), 'STL (file format)', `https://en.wikipedia.org/w/index.php?title=STL_(file_format)&oldid=916177065`. [Online; accessed October-2019].

Xing, Y., Liu, X. & Xu, S. (2010), 'Efficient collision detection based on aabb trees and sort algorithm', *IEEE ICCA 2010* pp. 328–32.

Zhang, X. & Liu, Y. (2015*a*), 'A Simple Filtering Algorithm for Continuous Collision Detection Using Taylor Models', *2015 14th International Conference on Computer-Aided Design and Computer Graphics (CAD/Graphics)* (14), 1–7.

Zhang, X. & Liu, Y. (2015*b*), 'An algebraic non-penetration filter for continuous collision detection using Sturm Theorem', *2015 IEEE International Conference on Mechatronics and Automation (ICMA)* pp. 761–6.

Zhiliang, L. & Desheng, Z. . (2011), 'The collision detection algorithm based on bounding volumes and space subdivision', *2011 International Conference on Image Analysis and Signal Processing* pp. 302–4.

Appendix A

Project Specification

ENG 4111/2 (or ENG8002) Research Project

Project Specification

For:	**Rob Brooker**
Topic:	Collision Avoidance in Motion Control Systems by Real-Time Predictive Interference Dete
Supervisors:	Dr Tobias Low
Sponsorship:	Mexx Engineering
Project Aim:	To create a method for machines to protect themselves from self-collision by making them aware of their own physical structure, and that of their moving parts.

Program:

1. Develop 3D model with at least 3 moving sub-assemblies that can collide with other moving parts, and fixed parts
2. Build PLC program which would control a machine made from the model (1)
3. Manually define collision rules by examining 3D model (1)
4. Add a background task to PLC program (2) to use rule from (3) to prevent collisions
5. Export 3D model (1) to SD card in STL format for dynamic collision avoidance on PLC
6. Iterative experimentation and further research on collision detection methods to find most cycle time efficient method
7. Write selected real-time collision avoidance technique into 2nd background task on PLC (2)
8. Run simulations of PLC program (2) with each of the background tasks (4,7) enabled to compare response times
9. Determine safety margin between parts considering response times, max speeds and max acceleration/deceleration

If time permits and real-time methods prove too slow:

10. Write PC program to perform a run once, exhaustive analysis of the 3D model (1) using selected collision detection method (6) to generate a rule set
11. Add a third background task containing the automatically generated rule set (10)
12. Run simulations again with all three background tasks (4,7,11)

Agreed:

Student Name:	Rob Brooker
Date:	13 March 2019
Supervisor Name:	Dr Tobias Low
Date:	9 April 2019

Appendix B

Model Export Program Listing

```
 1  Imports SolidEdgeFramework
 2  Imports System.Runtime.InteropServices
 3  Imports Excel = Microsoft.Office.Interop.Excel
 4  Imports Microsoft.Office
 5  Imports System.IO
 6
 7  Public Class SEAssy_Export
 8      Private mParent As String
 9      Private mName As String
10      Private mPath As String
11      Private mItemNum As Integer
12      Private mChildren As Collection
13      Private mMoveAxis As Integer
14      Private mMoveXScale As Double
15      Private mMoveXOffset As Double
16      Private mMoveYScale As Double
17      Private mMoveYOffset As Double
18      Private mMoveZScale As Double
19      Private mMoveZOffset As Double
20      Private mOriginX As Double
21      Private mOriginY As Double
22      Private mOriginZ As Double
23      Private mAngleX As Double
24      Private mAngleY As Double
25      Private mAngleZ As Double
26
27
28      Public Sub New(ParentFile As String, FileName As String, Folder As String,
          ItemNum As Integer, MoveAxis As Integer,
29                     Optional MoveXScale As Double = 0, Optional MoveXOffset As
                        Double = 0,
30                     Optional MoveYScale As Double = 0, Optional MoveYOffset As
                        Double = 0,
31                     Optional MoveZScale As Double = 0, Optional MoveZOffset As
                        Double = 0)
32          mParent = ParentFile
33          mName = FileName
34          mPath = Folder
35          mItemNum = ItemNum
36          mMoveAxis = MoveAxis
37          mMoveXScale = MoveXScale
38          mMoveXOffset = MoveXOffset
39          mMoveYScale = MoveYScale
40          mMoveYOffset = MoveYOffset
41          mMoveZScale = MoveZScale
42          mMoveZOffset = MoveZOffset
43          mChildren = New Collection()
44          'Traverse()
45      End Sub
```

```
46
47      Public Property Parent As String
48          Get
49              Return mParent
50          End Get
51          Set(value As String)
52              mParent = value
53          End Set
54      End Property
55
56      Public Property Name As String
57          Get
58              Return mName
59          End Get
60          Set(value As String)
61              mName = value
62          End Set
63      End Property
64
65      Public Property Path As String
66          Get
67              Return mPath
68          End Get
69          Set(value As String)
70              mPath = value
71          End Set
72      End Property
73
74      Public Property ItemNum As Integer
75          Get
76              Return mItemNum
77          End Get
78          Set(value As Integer)
79              mItemNum = value
80          End Set
81      End Property
82
83      Public ReadOnly Property FullName As String
84          Get
85              Return mPath + "\\" + mName
86          End Get
87      End Property
88
89      Public Property MoveAxis As Integer
90          Get
91              Return mMoveAxis
92          End Get
93          Set(value As Integer)
94              mMoveAxis = value
```

```
95          End Set
96      End Property
97
98      Public Property MoveXScale As Double
99          Get
100             Return mMoveXScale
101         End Get
102         Set(value As Double)
103             mMoveXScale = value
104         End Set
105     End Property
106
107     Public Property MoveXOffset As Double
108         Get
109             Return mMoveXOffset
110         End Get
111         Set(value As Double)
112             mMoveXOffset = value
113         End Set
114     End Property
115
116     Public Property MoveYScale As Double
117         Get
118             Return mMoveYScale
119         End Get
120         Set(value As Double)
121             mMoveYScale = value
122         End Set
123     End Property
124
125     Public Property MoveYOffset As Double
126         Get
127             Return mMoveYOffset
128         End Get
129         Set(value As Double)
130             mMoveYOffset = value
131         End Set
132     End Property
133
134     Public Property MoveZScale As Double
135         Get
136             Return mMoveZScale
137         End Get
138         Set(value As Double)
139             mMoveZScale = value
140         End Set
141     End Property
142
143     Public Property MoveZOffset As Double
```

```
144         Get
145             Return mMoveZOffset
146         End Get
147         Set(value As Double)
148             mMoveZOffset = value
149         End Set
150     End Property
151 
152     Public Property OriginX As Double
153         Get
154             Return mOriginX
155         End Get
156         Set(value As Double)
157             mOriginX = value
158         End Set
159     End Property
160 
161     Public Property OriginY As Double
162         Get
163             Return mOriginY
164         End Get
165         Set(value As Double)
166             mOriginY = value
167         End Set
168     End Property
169 
170     Public Property OriginZ As Double
171         Get
172             Return mOriginZ
173         End Get
174         Set(value As Double)
175             mOriginZ = value
176         End Set
177     End Property
178 
179     Public Property AngleX As Double
180         Get
181             Return mAngleX
182         End Get
183         Set(value As Double)
184             mAngleX = value
185         End Set
186     End Property
187 
188     Public Property AngleY As Double
189         Get
190             Return mAngleY
191         End Get
192         Set(value As Double)
```

```
193             mAngleY = value
194         End Set
195     End Property
196
197     Public Property AngleZ As Double
198         Get
199             Return mAngleZ
200         End Get
201         Set(value As Double)
202             mAngleZ = value
203         End Set
204     End Property
205
206     Public Sub Traverse()
207         Dim objApp As SolidEdgeFramework.Application = Nothing
208         Dim objDocs As SolidEdgeFramework.Documents = Nothing
209         Dim objDoc As SolidEdgeFramework.SolidEdgeDocument = Nothing
210         Dim objPart As SolidEdgePart.PartDocument = Nothing
211         Dim objAssy As SolidEdgeAssembly.AssemblyDocument = Nothing
212         Dim objAssySub As SolidEdgeAssembly.AssemblyDocument = Nothing
213         Dim objOccurances As SolidEdgeAssembly.Occurrences = Nothing
214         Dim objOccurance As SolidEdgeAssembly.Occurrence = Nothing
215         Dim objSubOccurance As SolidEdgeAssembly.SubOccurrence = Nothing
216         Dim objSelectSet As SolidEdgeFramework.SelectSet = Nothing
217         Dim objRef As SolidEdgeFramework.Reference = Nothing
218         Dim objVars As SolidEdgeFramework.Variables = Nothing
219         Dim objVar As SolidEdgeFramework.variable = Nothing
220         Dim tmpAssyExp As SEAssy_Export = Nothing
221         Dim strIndex As String = Nothing
222         Dim strName As String = Nothing
223         Dim strPath As String = Nothing
224         Dim strType As String = Nothing
225         Dim strExpPath As String = Nothing
226         Dim strExpName As String = Nothing
227         Dim mOriginX, mOriginY, mOriginZ, mAngleX, mAngleY, mAngleZ As Double
228         Dim mMatrix(16) As Double
229         Dim lMatrix(16) As Double
230         Dim tMatrix(16) As Double
231         Dim iItemNum As Integer
232
233         Dim i, j As Integer
234         mChildren = New Collection
235
236         Try
237             strExpName = Name.Substring(0, Name.Length - 4) + "~" +
                  ItemNum.ToString() + ".STL"
238             'Open tmpAssembly in current instance of Solid Edge
239             objApp = Marshal.GetActiveObject("SolidEdge.Application")
240             objDoc = objApp.Documents.Open(mPath + "\\" + mName)
```

```
241             objDoc.Activate()
242             'Check if open document is an Assembly
243             If (objDoc.Type = DocumentTypeConstants.igAssemblyDocument) Then
244                 objAssy = objDoc
245                 objOccurances = objAssy.Occurrences
246                 'Go through parts, or sub-assemblies looking for Moving parts
247                 i = 1
248                 While i <= objOccurances.Count
249                     tmpAssyExp = Nothing
250                     objOccurance = objOccurances.Item(i)
251                     'MsgBox("Occurance = " + objOccurance.Name)
252                     If objOccurance.Type = ObjectType.igPart Then
253                         objPart = objOccurance.PartDocument
254                         objVars = objPart.Variables
255                         strName = objPart.Name
256                         strPath = objPart.Path
257                         strType = "Part"
258                     ElseIf objOccurance.Type = ObjectType.igSubAssembly Then
259                         objAssySub = objOccurance.OccurrenceDocument
260                         objVars = objAssySub.Variables
261                         strName = objAssySub.Name
262                         strPath = objAssySub.Path
263                         strType = "SubAssembly"
264                     End If
265                     For j = 1 To objVars.Count
266                         objVar = objVars.Item(j)
267                         If objVar.Expose > 0 Then
268                             If objVar.ExposeName = "MoveAxis" Then
269                                 'MsgBox("Axis = " + objVar.Value.ToString())
270                                 If CInt(objVar.Value) >= 0 Then
271                                     iItemNum = CInt(objOccurance.Name.Substring
    (objOccurance.Name.IndexOf(":") + 1))
272                                     tmpAssyExp = New SEAssy_Export(strExpName,
    strName, strPath, iItemNum, CInt(objVar.Value))
273                                 End If
274                                 Exit For
275                             End If
276                         End If
277                     Next
278                     If tmpAssyExp IsNot Nothing Then
279                         objOccurance.GetTransform(mOriginX, mOriginY, mOriginZ,
     mAngleX, mAngleY, mAngleZ)
280                         tmpAssyExp.OriginX = Math.Round(mOriginX * 1000)
281                         tmpAssyExp.OriginY = Math.Round(mOriginY * 1000)
282                         tmpAssyExp.OriginZ = Math.Round(mOriginZ * 1000)
283                         tmpAssyExp.AngleX = Math.Round(180 * mAngleX / Math.PI)
284                         tmpAssyExp.AngleY = Math.Round(180 * mAngleY / Math.PI)
285                         tmpAssyExp.AngleZ = Math.Round(180 * mAngleZ / Math.PI)
286                         For j = 1 To objVars.Count
```

```
287                             objVar = objVars.Item(j)
288                             If objVar.Expose > 0 Then
289                                 If objVar.ExposeName = "MoveXScale" Then
290                                     tmpAssyExp.MoveXScale = CDbl(objVar.Value)
291                                 ElseIf objVar.ExposeName = "MoveXOffset" Then
292                                     tmpAssyExp.MoveXOffset = CDbl(objVar.Value)
293                                 ElseIf objVar.ExposeName = "MoveYScale" Then
294                                     tmpAssyExp.MoveYScale = CDbl(objVar.Value)
295                                 ElseIf objVar.ExposeName = "MoveYOffset" Then
296                                     tmpAssyExp.MoveYOffset = CDbl(objVar.Value)
297                                 ElseIf objVar.ExposeName = "MoveZScale" Then
298                                     tmpAssyExp.MoveZScale = CDbl(objVar.Value)
299                                 ElseIf objVar.ExposeName = "MoveZOffset" Then
300                                     tmpAssyExp.MoveZOffset = CDbl(objVar.Value)
301                                 End If
302                             End If
303                         Next
304                         objOccurance.Delete()
305                         i -= 1
306                         mChildren.Add(tmpAssyExp, tmpAssyExp.Name)
307                         'MsgBox(strType + ": " + tmpAssyExp.Name + " Deleted.")
308                     Else
309                         'MsgBox(strType + ": " + objOccurance.Name + " Remains as fixed Part.")
310                     End If
311                     i += 1
312                 End While
313             End If
314             'MsgBox(mChildren.Count.ToString() + " Occurances Deleted.")
315             strExpPath = Path + "\STL_Exports"
316             If Not Directory.Exists(strExpPath) Then
317                 Directory.CreateDirectory(strExpPath)
318             End If
319             objAssy.SaveAs(strExpPath + "\" + strExpName)
320             MsgBox(strExpName + " Saved")
321 
322         Catch ex As Exception
323             MsgBox(ex.Message)
324         Finally
325             'Dispose SelectSet object
326             If Not (objSelectSet Is Nothing) Then
327                 Marshal.ReleaseComObject(objSelectSet)
328                 objSelectSet = Nothing
329             End If
330             'Dispose SubOccurance object
331             If Not (objRef Is Nothing) Then
332                 Marshal.ReleaseComObject(objRef)
333                 objRef = Nothing
334             End If
```

```
335             'Dispose Occurance object
336             If Not (objOccurance Is Nothing) Then
337                 Marshal.ReleaseComObject(objOccurance)
338                 objOccurance = Nothing
339             End If
340             'Dispose Occurances collection
341             If Not (objOccurances Is Nothing) Then
342                 Marshal.ReleaseComObject(objOccurances)
343                 objOccurances = Nothing
344             End If
345             'Dispose Assembly object
346             If Not (objAssy Is Nothing) Then
347                 Marshal.ReleaseComObject(objAssy)
348                 objAssy = Nothing
349             End If
350             'Dispose Part object
351             If Not (objPart Is Nothing) Then
352                 Marshal.ReleaseComObject(objPart)
353                 objPart = Nothing
354             End If
355             'Dispose Document object
356             If Not (objDoc Is Nothing) Then
357                 Marshal.ReleaseComObject(objDoc)
358                 objDoc = Nothing
359             End If
360             'Dispose Documents collection object
361             If Not (objDocs Is Nothing) Then
362                 Marshal.ReleaseComObject(objDocs)
363                 objDocs = Nothing
364             End If
365             'Dispose Solid Edge Application object
366             If Not (objApp Is Nothing) Then
367                 Marshal.ReleaseComObject(objApp)
368                 objApp = Nothing
369             End If
370         End Try
371     End Sub
372
373 End Class
374
```

Appendix C

Collision Detection Program Listing

```cpp
// CART.cpp : This file contains the 'main' function. Program execution begins and ends there.
//

#include <iostream>
#include <fstream>
#include <stdio.h>
#include <string>
#include <vector>
#include <algorithm>
#include <functional>
#include <cctype>
#include <locale>
#include <sstream>
#include <wiringPi.h>
#include <chrono>
typedef std::chrono::high_resolution_clock Clock;

//Setup for SOLID physics library
#include "include/SOLID/solid.h"
#include "include/3D/Point.h"
#include "include/3D/Quaternion.h"
#include "ModelPart.h"

//Setup for EasyCAT fieldbus adaptor
#include <bcm2835.h>
#include <unistd.h>
#define CUSTOM
#include "EasyCAT.h"

using namespace std;

#define LOBYTE(x) ((unsigned char) ((x) & 0xff))
#define HIBYTE(x) ((unsigned char) ((x) >> 8 & 0xff))

typedef struct MyObject {
    int id;
} MyObject;

typedef struct
    {
        float       Axis0_Pos;
        float       Axis0_Vel;
        float       Axis1_Pos;
        float       Axis1_Vel;
        float       Axis2_Pos;
        float       Axis2_Vel;
        float       Axis3_Pos;
        float       Axis3_Vel;
        uint8_t     Axis0_Status;
        uint8_t     Axis2_Status;
        uint8_t     PLC_Status;
        uint8_t     Resolution;
```

```
53          uint8_t     Axis3_Status;
54          uint8_t     Axis1_Status;
55          uint8_t     AxisGroup;
56      }ECAT_Rcv;
57
58  typedef struct
59      {
60          uint32_t    DigitalOutputs;
61          uint8_t     Resolution;
62          uint8_t     Device_Status;
63          uint8_t     AxisGroup;
64          uint8_t     Axis0_Status;
65          uint8_t     Axis1_Status;
66          uint8_t     Axis2_Status;
67          uint8_t     Axis3_Status;
68      }ECAT_Snd;
69
70  //Define Device Status Bits
71  const uint8_t DS_LOADING = 0b0000'0001; //Bit0 Loading Model
72  const uint8_t DS_TESTING = 0b0000'0010; //Bit1 Testing for Collisions
73  const uint8_t DS_FAULT   = 0b0000'0100; //Bit2 Fault state
74  const uint8_t DS_READY   = 0b0000'1000; //Bit3 Operating normally
75  const uint8_t DS_Bit4    = 0b0001'0000; //Bit4 Reserved
76  const uint8_t DS_Bit5    = 0b0010'0000; //Bit5 Reserved
77  const uint8_t DS_Bit6    = 0b0100'0000; //Bit6 Reserved
78  const uint8_t DS_Bit7    = 0b1000'0000; //Bit7 Reserved
79
80  //Define PLC Status Bits
81  const uint8_t PS_RUN     = 0b0000'0001; //Bit0 Run tests continuously
82  const uint8_t PS_FAULT   = 0b0000'0010; //Bit1 Fault state
83  const uint8_t PS_READY   = 0b0000'0100; //Bit2 Operating normally
84  const uint8_t PS_Bit3    = 0b0000'1000; //Bit3 Reserved
85  const uint8_t PS_Bit4    = 0b0001'0000; //Bit4 Reserved
86  const uint8_t PS_Bit5    = 0b0010'0000; //Bit5 Reserved
87  const uint8_t PS_Bit6    = 0b0100'0000; //Bit6 Reserved
88  const uint8_t PS_Bit7    = 0b1000'0000; //Bit7 Reserved
89
90
91  /* ARGSUSED */
92  void collide1(void* client_data, DtObjectRef obj1, DtObjectRef obj2,
93      const DtCollData* coll_data) {
94  }
95
96
97
98  #ifdef STATISTICS
99  extern int num_box_tests;
100 #endif
101
102 //Prototypes
103 int LoadSTL(ModelPart mPart);
104
105 int LoadConfig();
```

```
106
107 void CreateParts();
108
109 void MoveParts();
110
111 void WriteOutputs();
112
113 void ClearOutputs();
114
115 void CleanupSOLID();
116
117 void ParentMove(int, DT_Scalar&, DT_Scalar&, DT_Scalar&);
118
119 int ReadIntAfter(string& strIn, int iFrom);
120
121 double ReadDblAfter(string& strIn, int iFrom);
122
123 void EtherCAT_Exchange();
124
125 // trim from start
126 static inline std::string& ltrim(std::string& s) {
127     s.erase(s.begin(), std::find_if(s.begin(), s.end(),
128         std::not1(std::ptr_fun<int, int>(std::isspace))));
129     return s;
130 }
131
132 // trim from end
133 static inline std::string& rtrim(std::string& s) {
134     s.erase(std::find_if(s.rbegin(), s.rend(),
135         std::not1(std::ptr_fun<int, int>(std::isspace))).base(), s.end());
136     return s;
137 }
138
139 // trim from both ends
140 static inline std::string& trim(std::string& s) {
141     return ltrim(rtrim(s));
142 }
143
144 //Global Variables
145 string path = "../../";
146 string arg;
147 vector<MyObject> vObjects;
148 vector<DtShapeRef> vShapes;
149 vector<ModelPart> vParts;
150 vector<DT_Scalar> vCurPos;
151 vector<DT_Scalar> vCurVel;
152 string resStrings[6] = {"10", "1", "01", "001", "0005", "0001"};
153 EasyCAT EASYCAT;                // EtherCAT slave istantiation
154 ECAT_Rcv rcvDat;        // Data received from EtherCAT slave
155 ECAT_Snd sndDat;        // Data for EtherCAT slave to send out
156 int iNumParts;          // Number of Model Parts in memory
157 int useCmd = 1;         // Allow position entry from command line instead of ↵
       EtherCAT
```

```
158 string strRes = "";     // Choose model resolution to be loaded
159 int oldRes;         // for checking if PLC has requested a resolution change
160
161 /* ARGSUSED */
162 void collide2(void* client_data, DtObjectRef obj1, DtObjectRef obj2,
163     const DtCollData* coll_data) {
164     FILE* stream = (FILE*)client_data;
165     fprintf(stream, "Object %d interferes with object %d\n",
166         (*(MyObject*)obj1).id, (*(MyObject*)obj2).id);
167     vParts[(*(MyObject*)obj1).id].Status |= AS_COLLISION;
168     vParts[(*(MyObject*)obj2).id].Status |= AS_COLLISION;
169 }
170
171 int main(int argc, char *argv[])
172 {
173     //define local variables
174     bool bContinue = true, bCollision;
175     string strCmd;
176     int col_count = 0;
177     Quaternion q;
178
179     //Handle command line arguments
180     for (int i = 0; i < argc; i++)
181     {
182         arg = argv[i];
183         if (arg.find("-cmd") == 0) useCmd = 1;
184         if (arg.find("-ECAT") == 0) useCmd = 0;
185         if (arg.find("-res=") == 0) strRes = arg.substr(5);
186     }
187
188     //Start Digital IO interface
189     wiringPiSetup();
190     for (int i = 0; i < 8; i++) pinMode(i, OUTPUT);
191
192     //Start EtherCAT slave if it is being used
193     if (useCmd == 0)
194     {
195         if (EASYCAT.Init() == true)                 // EasyCAT Hat
              initialization
196                 cout << "EtherCAT slave Initialized." << std::endl; //
                      succesfully completed
197             else                                    // initialization failed
198                 cout << "EtherCAT slave inizialization failed." << endl;//
                      the EasyCAT board was not recognized
199
200         sndDat.Device_Status &= 0;  //Clear all status bits being sent to
              the PLC
201     }
202
203     sndDat.Device_Status |= DS_LOADING; //Set LOADING Status Bit
204     LoadConfig();   //Load descriptions of ModelParts to be imported
205     CreateParts();  //Create ModelParts by importing STL files into SOLID
206     sndDat.Device_Status &= ~DS_LOADING;    //Reset LOADING Status Bit
```

```
207
208
209     while (bContinue)
210     {
211         //Update current positions of loaded objects
212         if (useCmd == 0)//Axis positions comming from EtherCAT fieldbus
213         {
214             EASYCAT.MainTask();      // execute the EasyCAT task
215             EtherCAT_Exchange();         // perform data exchange with
                  EtherCAT slave
216         }
217
218         MoveParts();    //Update positions of ModelParts to match real
              machine
219
220         //Test for collision
221         sndDat.Device_Status &= ~DS_READY;
222         sndDat.Device_Status |= DS_TESTING;
223         ClearOutputs();
224         auto t1 = Clock::now();
225         bCollision = dtTest();
226
227         auto t2 = Clock::now();
228         WriteOutputs();
229         std::cout << "Process time = " <<
              std::chrono::duration_cast<std::chrono::microseconds>(t2 -
              t1).count()
230         << " microseconds" << std::endl;
231         sndDat.Device_Status &= ~DS_TESTING;
232         sndDat.Device_Status |= DS_READY;
233
234         //Prompt Operator to Continue or Quit if using command line
235         if (useCmd == 1)
236         {
237             bContinue = false;
238             std::cout << "Type 'c' to continue, or 'q' to Quit: ";
239             std::cin >> strCmd;
240             if (strCmd != "q") bContinue = true;
241         }
242         else
243         {
244             if (oldRes != rcvDat.Resolution)
245             {
246                 sndDat.Device_Status &= ~DS_READY;
247                 sndDat.Device_Status |= DS_LOADING;
248                 CleanupSOLID();
249                 LoadConfig();
250                 CreateParts();
251                 oldRes = rcvDat.Resolution;
252                 sndDat.Device_Status &= ~DS_LOADING;
253             }
254         }
255     }
```

```
256
257
258     std::cout << "Number of collisions: " << col_count << endl;
259 #ifdef STATISTICS
260     cout << "Number of sep_axis calls: " << num_box_tests << endl;
261 #endif
262     //Cleanup on SOLID Library objects
263     CleanupSOLID();
264
265     return 0;
266 }
267
268
269 int LoadConfig()
270 // LoadConfig proceedure opens Config.txt and reads names and attributes
271 // of ModelParts to be tested for collision
272 // Requires global variables: vParts, iNumParts.
273 {
274     iNumParts = 0;
275
276     //Determine Model Resolution to use
277     if ((strRes.length() == 0) & (useCmd == 1))
278     {
279         strRes = "1";
280     }
281     else
282     {
283         if (useCmd == 0)
284         {
285             EASYCAT.MainTask(); // execute the EasyCAT task
286             EtherCAT_Exchange();// perform data exchange with EtherCAT slave
287             strRes = resStrings[rcvDat.Resolution];
288             sndDat.Resolution = rcvDat.Resolution;
289         }
290     }
291
292     //Load Config file
293     string inStr, strMachine, configFile;
294     string res;
295     ifstream infile;
296
297     //std::cout << "Enter Config file resolution (10, 1, 01, 001, 0001,
        etc): ";
298     //std::cin >> res;
299     configFile = path + "Config_" + strRes + ".txt";
300     infile.open(configFile);
301     cout << "Attempting to Open: " << configFile << "\n";
302     if (!infile.fail())
303     {
304         std::cout << "Config file is Open.\n";
305         while (getline(infile, inStr))
306         {
307             if (inStr.find("ONFIG: ") == 1)
```

```
308             {
309                 strMachine = inStr.substr(8);
310                 std::cout << "Beginning configuration for: " << strMachine
                      << "\n";
311                 getline(infile, inStr);
312                 if (inStr.find("PARTS: ") == 1)
313                 {
314                     inStr = inStr.substr(8);
315                     istringstream os(inStr);
316                     os >> iNumParts;
317                     std::cout << "Setting up for " << iNumParts << " parts.
                    \n";
318                     vParts.resize(iNumParts);
319                     getline(infile, inStr); getline(infile, inStr);
320                     if (inStr.find("IXED: ") == 1)
321                     {
322                         inStr = inStr.substr(12);
323                         vParts[0].Filename = inStr.substr(0, inStr.size
                    ()-1);
324                         vParts[0].Axis = -1;
325                         std::cout << "Fixed Part called: " << inStr << "\n";
326                         for (int i = 2; i <= iNumParts; i++)
327                         {
328                             getline(infile, inStr);
329                             inStr = inStr.substr(12);
330                             vParts[i - 1].Filename = inStr.substr(0,
                    inStr.size()-1);
331                             getline(infile, inStr);
332                             vParts[i - 1].Axis = ReadIntAfter(inStr, 12);
333                             getline(infile, inStr);
334                             vParts[i - 1].Parent = ReadIntAfter(inStr, 12);
335                             getline(infile, inStr);
336                             istringstream os(inStr.substr(12));
337                             os >> vParts[i - 1].q1 >> vParts[i - 1].q2 >>
                    vParts[i - 1].q3 >> vParts[i - 1].q4;
338                             getline(infile, inStr);
339                             istringstream os1(inStr.substr(12));
340                             os1 >> vParts[i - 1].XTrans >> vParts[i -
                    1].YTrans >> vParts[i - 1].ZTrans;
341                             getline(infile, inStr);
342                             vParts[i - 1].XOffset = ReadDblAfter(inStr, 12);
343                             getline(infile, inStr);
344                             vParts[i - 1].YOffset = ReadDblAfter(inStr, 12);
345                             getline(infile, inStr);
346                             vParts[i - 1].ZOffset = ReadDblAfter(inStr, 12);
347                             getline(infile, inStr);
348                             vParts[i - 1].XScale = ReadDblAfter(inStr, 12);
349                             getline(infile, inStr);
350                             vParts[i - 1].YScale = ReadDblAfter(inStr, 12);
351                             getline(infile, inStr);
352                             vParts[i - 1].ZScale = ReadDblAfter(inStr, 12);
353                             std::cout << "Motion Part called: " << vParts[i
                    - 1].Filename << "\n";
```

```
354                         }
355                     }
356                     else
357                     {
358                         std::cout << "Error: Did not find name of Fixed
                        Part.\n";
359                     }
360                 }
361                 else
362                 {
363                     std::cout << "Error: Did not find the number of Parts to
                    load.\n";
364                 }
365             }
366         }
367         infile.close();
368         std::cout << "Config file closed.\n";
369         return 1;
370     }
371     return 0;
372 }
373
374
375 void CreateParts()
376 {
377     int iCount = 0;
378
379     //Resize variables to hold the parts to be created
380     vObjects.resize(iNumParts);
381     vShapes.resize(iNumParts);
382     vCurPos.resize(iNumParts);
383     vCurVel.resize(iNumParts);
384     std::cout << "Loading " << iNumParts << " parts.\n";
385
386     //Get Load start-time
387     auto t1 = Clock::now();
388     for (int i = 0; i < iNumParts; i++)
389     {
390         //Import each STL file into a ComplexShape
391         vObjects[i].id = vParts[i].Axis;
392         vShapes[i] = dtNewComplexShape();
393         iCount = LoadSTL(vParts[i]);
394         dtEndComplexShape();
395         dtCreateObject(&vObjects[i], vShapes[i]);
396         if (iCount > 0)
397             std::cout << vParts[i].Filename << " loaded with " << iCount <<
                " triangles.\n";
398         else
399             std::cout << vParts[i].Filename << " failed to find any
                triangles.\n";
400     }
401
402     dtDisableCaching();
```

```
403     dtSetDefaultResponse(collide2, DT_SMART_RESPONSE, stdout);
404
405     //get Load end-time
406     auto t2 = Clock::now();
407     std::cout << "Load time = " <<
          std::chrono::duration_cast<std::chrono::milliseconds>(t2 - t1).count()
408       << " milliseconds" << std::endl;
409 }
410
411
412 int LoadSTL(ModelPart mPart)
413 // LoadSTL procedure opens an STL file specified in Config file, and
      generates
414 // a DT_ShapeRef to be used for Colision Detection
415 {
416     string inStr, xStr, yStr, zStr;
417     DT_Scalar inX, inY, inZ, outX, outY, outZ;
418     DT_Scalar q1, q2, q3, q4;
419     DT_Scalar minX = 0, maxX = 0, minY = 0, maxY = 0, minZ = 0, maxZ = 0;
420     DT_Scalar shiftX, shiftY, shiftZ;
421     int iCount = 0;
422     ifstream infile1;
423     string filename;
424     filename = path + mPart.Filename;
425
426     infile1.open(filename);
427     cout << "Attempting to Open: [" << filename << "]\n";
428     if (!infile1.fail())
429     {
430         cout << mPart.Filename << " Opened successfully.\n";
431         //Prepare Orientation
432         q1 = mPart.q1; q2 = mPart.q2; q3 = mPart.q3; q4 = mPart.q4;
433         //Prepare shift values
434         if(mPart.XScale == 0) shiftX = mPart.XTrans; else shiftX =
              mPart.XOffset;
435         if(mPart.YScale == 0) shiftY = mPart.YTrans; else shiftY =
              mPart.YOffset;
436         if(mPart.ZScale == 0) shiftZ = mPart.ZTrans; else shiftZ =
              mPart.ZOffset;
437         while (getline(infile1, inStr))
438         {
439             inStr = inStr.substr(0, inStr.size()-1);
440             //cout << inStr << " [" << inStr.length() << "]\n";
441             if (inStr == "    outer loop")
442             {
443                 //cout << "Start of Triangle.";
444                 dtBegin(DT_SIMPLEX);
445                 for (size_t i = 0; i < 3; i++)
446                 {
447                     //Reduce inStr to the 3 components of the Vertex
448                     getline(infile1, inStr);
449                     inStr = ltrim(inStr);
450                     inStr = inStr.substr(6);
```

```
451                     inStr = ltrim(inStr);
452                     //Separate the string into X, Y, Z values as strings
453                     istringstream os(inStr);
454                     os >> inX >> inY >> inZ;
455                     //Check if rotation is required
456                     if ((q1 == 1) && (q2 == 0) && (q3 == 0) && (q4 == 0))
457                     {
458                         outX = inX; outY = inY; outZ = inZ;
459                     }
460                     else
461                     {
462                         outX = (q1*q1*inX)   + (2*q3*q1*inZ) - (2*q4*q1*inY)
                        + (q2*q2*inX)   + (2*q3*q2*inY) + (2*q4*q2*inZ) -
                       (q4*q4*inX)   - (q3*q3*inX);
463                         outY = (2*q2*q3*inX) + (q3*q3*inY)   + (2*q4*q3*inZ)
                        + (2*q1*q4*inX) - (q4*q4*inY)   + (q1*q1*inY)   -
                       (2*q2*q1*inZ) - (q2*q2*inY);
464                         outZ = (2*q2*q4*inX) + (2*q3*q4*inY) + (q4*q4*inZ)
                        - (2*q1*q3*inX) - (q3*q3*inZ)   + (2*q1*q2*inY) -
                       (q2*q2*inZ)   + (q1*q1*inZ);
465                     }
466                     //Add the Vertex to the complex shape
467                     outX = shiftX + outX;
468                     outY = shiftY + outY;
469                     outZ = shiftZ + outZ;
470                     dtVertex(outX, outY, outZ);
471                     if((outX < minX) | (iCount == 0)) minX = outX;
472                     if((outX > maxX) | (iCount == 0)) maxX = outX;
473                     if((outY < minY) | (iCount == 0)) minY = outY;
474                     if((outY > maxY) | (iCount == 0)) maxY = outY;
475                     if((outZ < minZ) | (iCount == 0)) minZ = outZ;
476                     if((outZ > maxZ) | (iCount == 0)) maxZ = outZ;
477                 }
478                 dtEnd();
479                 //cout << " End of Triangle.\n";
480                 iCount++;
481             }
482         }
483         infile1.close();
484         cout << "Bounds: ([" << minX << "," << maxX << "], [" << minY << ","
               << maxY;
485         cout << "], [" << minZ << "," << maxZ << "])\n";
486     }
487     else cout << "Failed to Open: " << mPart.Filename << "\n";
488     return iCount;
489 }
490
491
492 void MoveParts()
493 {
494     DT_Scalar curAxisVal;
495     DT_Scalar moveX, moveY, moveZ;
496
```

```
497     for (int i = 0; i < iNumParts; i++)
498     {
499         if (vParts[i].Axis >= 0) //Check if current Part can be moved by an Axis
500         {
501             if (useCmd == 1) //Get axis position from command line if not using EtherCAT
502             {
503                 std::cout << "Enter position for " << vParts[i].Filename << ": ";
504                 std::cin >> curAxisVal;
505                 vCurPos[i] = curAxisVal;
506             }
507             dtSelectObject(&vObjects[i]);   //Select Part
508             dtLoadIdentity();       //Load Identity matrix for the current Part
509             //Apply Axis position to the Part according to the Config file
510             moveX = vParts[i].XOffset + vParts[i].XScale * vCurPos[i];
511             moveY = vParts[i].YOffset + vParts[i].YScale * vCurPos[i];
512             moveZ = vParts[i].ZOffset + vParts[i].ZScale * vCurPos[i];
513             if (vParts[i].Parent > -1) //Check if there is a moving parent Axis
514                 //Call recursive procedure to update position from all parents
515                 ParentMove(vParts[i].Parent, moveX, moveY, moveZ);
516             dtTranslate(moveX, moveY, moveZ);
517         }
518     }
519 }
520
521
522 void ParentMove(int parent, DT_Scalar& deltaX, DT_Scalar& deltaY, DT_Scalar& deltaZ)
523 {
524     int grandParent = -1;
525
526     for (int j = 0; j < iNumParts; j++)
527     {
528         if (vParts[j].Axis == parent)
529         {
530             deltaX += vParts[j].XScale * vCurPos[j];
531             deltaY += vParts[j].YScale * vCurPos[j];
532             deltaZ += vParts[j].ZScale * vCurPos[j];
533             grandParent = vParts[j].Parent;
534             break;
535         }
536     }
537     if (grandParent > -1) ParentMove(grandParent, deltaX, deltaY, deltaZ);
538 }
539
540
541 void EtherCAT_Exchange()
542 {
```

```
543     int numInGrp = 0;
544
545     // Read data from PLC via EtherCAT slave
546     rcvDat.Axis0_Pos    = EASYCAT.BufferOut.Cust.P2D_Axis0_Pos;
547     rcvDat.Axis0_Vel    = EASYCAT.BufferOut.Cust.P2D_Axis0_Vel;
548     rcvDat.Axis1_Pos    = EASYCAT.BufferOut.Cust.P2D_Axis1_Pos;
549     rcvDat.Axis1_Vel    = EASYCAT.BufferOut.Cust.P2D_Axis1_Vel;
550     rcvDat.Axis2_Pos    = EASYCAT.BufferOut.Cust.P2D_Axis2_Pos;
551     rcvDat.Axis2_Vel    = EASYCAT.BufferOut.Cust.P2D_Axis2_Vel;
552     rcvDat.Axis3_Pos    = EASYCAT.BufferOut.Cust.P2D_Axis3_Pos;
553     rcvDat.Axis3_Vel    = EASYCAT.BufferOut.Cust.P2D_Axis3_Vel;
554     rcvDat.Axis0_Status = EASYCAT.BufferOut.Cust.P2D_Axis0_Status;
555     rcvDat.Axis2_Status = EASYCAT.BufferOut.Cust.P2D_Axis2_Status;
556     rcvDat.PLC_Status   = EASYCAT.BufferOut.Cust.PLC_Status;
557     rcvDat.Resolution   = EASYCAT.BufferOut.Cust.Resolution;
558     rcvDat.Axis3_Status = EASYCAT.BufferOut.Cust.P2D_Axis3_Status;
559     rcvDat.Axis1_Status = EASYCAT.BufferOut.Cust.P2D_Axis1_Status;
560     rcvDat.AxisGroup    = EASYCAT.BufferOut.Cust.P2D_AxisGroup;
561
562     // Write data to PLC via EtherCAT slave
563     EASYCAT.BufferIn.Cust.DigitalOutputs   = sndDat.DigitalOutputs;
564     EASYCAT.BufferIn.Cust.Resolution       = sndDat.Resolution;
565     EASYCAT.BufferIn.Cust.Device_Status    = sndDat.Device_Status;
566     EASYCAT.BufferIn.Cust.D2P_AxisGroup    = sndDat.AxisGroup;
567     EASYCAT.BufferIn.Cust.D2P_Axis0_Status = sndDat.Axis0_Status;
568     EASYCAT.BufferIn.Cust.D2P_Axis1_Status = sndDat.Axis1_Status;
569     EASYCAT.BufferIn.Cust.D2P_Axis2_Status = sndDat.Axis2_Status;
570     EASYCAT.BufferIn.Cust.D2P_Axis3_Status = sndDat.Axis3_Status;
571
572     for (int i = 0; i < iNumParts; i++)
573     {
574         if ((vParts[i].Axis >= rcvDat.AxisGroup*4) && (vParts[i].Axis <
              (rcvDat.AxisGroup + 1)*4))
575         {
576             numInGrp = vParts[i].Axis - rcvDat.AxisGroup*4;
577             switch (numInGrp) {
578                 case 0: vCurPos[i] = rcvDat.Axis0_Pos;
579                     vCurVel[i] = rcvDat.Axis0_Vel;
580                     break;
581                 case 1: vCurPos[i] = rcvDat.Axis1_Pos;
582                     vCurVel[i] = rcvDat.Axis1_Vel;
583                     break;
584                 case 2: vCurPos[i] = rcvDat.Axis2_Pos;
585                     vCurVel[i] = rcvDat.Axis2_Vel;
586                     break;
587                 case 3: vCurPos[i] = rcvDat.Axis3_Pos;
588                     vCurVel[i] = rcvDat.Axis3_Vel;
589                     break;
590                 default:
591                     break;
592             }
593         }
594     }
```

```
595 }
596
597
598 void WriteOutputs()
599 {
600     for (int i = 0; i < iNumParts; i++)
601     {
602         if (vParts[i].Status & AS_COLLISION)
603             digitalWrite(vParts[i].Axis, HIGH);
604         else
605             digitalWrite(vParts[i].Axis, LOW);
606     }
607 }
608
609 void ClearOutputs()
610 {
611     for (int i = 0; i < iNumParts; i++)
612         vParts[i].Status &= ~AS_COLLISION;
613 }
614
615 void CleanupSOLID()
616 {
617     for (int i = 0; i < iNumParts; i++)
618     {
619         dtDeleteObject(&vObjects[i]);
620         dtDeleteShape(vShapes[i]);
621     }
622 }
623
624
625 int ReadIntAfter(string& strIn, int iFrom)
626 {
627     int iResult = 0;
628     istringstream os(strIn.substr(iFrom));
629     os >> iResult;
630     return iResult;
631 }
632
633 double ReadDblAfter(string& strIn, int iFrom)
634 {
635     double dblResult = 0;
636     istringstream os(strIn.substr(iFrom));
637     os >> dblResult;
638     return dblResult;
639 }
640
```