DESIGN AND IMPLEMENTATION OF A MODULAR CONTROLLER FOR ROBOTIC MACHINES

A Thesis Submitted to the College of

Graduate Studies and Research

In Partial Fulfillment of the Requirements

For the Degree of Doctor of Philosophy

In the Department of Mechanical Engineering

University of Saskatchewan

Saskatoon

By

RODNEY ATTA-KONADU

Keywords: architecture, communication, distributed control, embedded, Java, modular, real-time network, reconfigurable, robots, Zeroconf

© Copyright Rodney Atta-Konadu, September, 2006. All rights reserved.

PERMISSION TO USE

In presenting this thesis in partial fulfilment of the requirements for a Postgraduate degree from the University of Saskatchewan, I agree that the Libraries of this University may make it freely available for inspection. I further agree that permission for copying of this thesis in any manner, in whole or in part, for scholarly purposes may be granted by the professor or professors who supervised my thesis work or, in their absence, by the Head of the Department or the Dean of the College in which my thesis work was done. It is understood that any copying or publication or use of this thesis or parts thereof for financial gain shall not be allowed without my written permission. It is also understood that due recognition shall be given to me and to the University of Saskatchewan in any scholarly use which may be made of any material in my thesis.

Requests for permission to copy or to make other use of material in this thesis in whole or part should be addressed to:

Head of the Department of Mechanical Engineering

University of Saskatchewan

Saskatoon, Saskatchewan (S7N 5A2)

ABSTRACT

This research focused on the design and implementation of an Intelligent Modular Controller (IMC) architecture designed to be reconfigurable over a robust network. The design incorporates novel communication, hardware, and software architectures. This was motivated by current industrial needs for distributed control systems due to growing demand for less complexity, more processing power, flexibility, and greater fault tolerance. To this end, three main contributions were made.

Most distributed control architectures depend on multi-tier heterogeneous communication networks requiring linking devices and/or complex middleware. In this study, first, a communication architecture was proposed and implemented with a homogenous network employing the ubiquitous Ethernet for both real-time and non real-time communication. This was achieved by a producer-consumer coordination model for real-time data communication over a segmented network, and a client-server model for point-to-point transactions. The protocols deployed use a Time-Triggered (TT) approach to schedule real-time tasks on the network. Unlike other TT approaches, the scheduling mechanism does not need to be configured explicitly when controller nodes are added or removed. An implicit clock synchronization technique was also developed to complement the architecture. Second, a reconfigurable mechanism based on an auto-configuration protocol was developed. Modules on the network use this protocol to automatically detect themselves, establish communication, and negotiate for a desired configuration. Third, the research demonstrated hardware/software co-design as a contribution to the growing discipline of mechatronics. The IMC consists of a motion controller board designed and prototyped in-house, and a Java microcontroller. An IMC is mapped to each machine/robot axis, and an additional IMC can be configured to serve as a real-time coordinator. The entire architecture was implemented in Java, thus reinforcing uniformity, simplicity, modularity, and openness. Evaluation results showed the potential of the flexible controller to meet medium to high performance machining requirements.

ACKNOWLEDGMENTS

I will like to thank my supervisor Professor Chris Zhang for the generous support and guidance he commited to me. My co-supervisor Dr. Sherman Lang of NRC-IMIT provided key directions, motivations, and support toward this research. I appreciate the guidance and encouragement I received from my committee members Professors Schoenau, Hertz, Gander, and Dr. Peter Orban of NRC-IMIT.

I certainly acknowledge NSERC and the Department of Mechanical Engineering for supporting my PhD work. Most of my research time was spent at the NRC-IMIT where I received part-funding for this project. I thank Mr. Georges Salloum the Director General and Dr. Gian Vascotto (Director) for endorsing this collaborative project. Memories of the congenial IMTI microcosm will linger with me for years to come. I acknowledge the input of Dr. Orban whose pragmatism and prodding had tremendous impact on this work. I note the support of Marcel Verner on many technical issues. Dr. Zhuming Bi gave me great companionship and *bent* the rules a few times to let me in during silent hours. I am very thankful for the companionship and support of IMTI staff such as Susan Salo, Brian Wong, Stan Kowala, Dave Kingston, Steve Kruithof, Lori Cox, Dr. Helen Xie, Millan Yeung, Dr. Lihui Wang, Dr. Weiming, and Mike Meinert. Trent Steensma (UWO Electronics Department) gave me much advice in PCB production. I also benefited greatly from forum discussions with the Java and Systronix communities.

I am exceedingly thankful for the invaluable impact of precious friends, particularly; Nii and Celestina Allotey, Erasmus and Catherine Amoateng, Charles and Agnes Annan, Frank and Cynthia Arku, Ernest and Monica Ansah-Sam, Eddie and Francisca Bansah, Milan Bhasin, Basia Bodzanowski, Brian and Darleen Carter, Joyce Grant, Mobi Emodi, Ron Hampson, Franklin and Bridgette Krampa, Michael and Irene Nketia, Jay and Tammy Nyyssonen, Kwaku and Regina Odei, Felix Oppong, Esther Oppong, Dr. Osei, George and Gertrude Quainoo, Professor Tachie who introduced me to my supervisor, Dr. Jean Yuchuan, my pastors Greg Olson and Howard Katz, and my step-mum Esther. I am thankful to all my family for coping with my protracted *abeyance*, and my dear wife Edwoba for her relentless prayers and support. The Lord Jesus Christ has always been and will always be my life and my resurrection.

DEDICATED TO

My dear wife Edwoba
My adorable son Elijah
My precious parents Dr. Atta-Konadu and Beatrice Adjei
My uncle Jackson (rtd. aircraft engineer) who has inspired me from my pre-school days

TABLE OF CONTENTS

PERMISSION TO USE I

ABSTRACT II

ACKNOWLEDGMENTS III

DEDICATED TO IV

TABLE OF CONTENTS V

LIST OF TABLES XI

LIST OF FIGURES XII

LIST OF ABBREVIATIONS XV

1. INTRODUCTION 1

1.1 Overview 1
1.2 Motivation 1
1.3 Goals and Contributions 3
1.4 Organization of Thesis 4
2. ARCHITECTURE DESIGN CONCEPTS AND REVIEW 6

2.1 Robot Control Architecture, Introduction 6
2.2 Architectural Properties 6
2.2.1 Dealing with Complexity 6
2.2.2 Execution 7
2.2.3 Openness 8
2.2.4 Performance 8
2.2.5 Scalability 9
2.2.6 Simplicity 9
2.2.7 Modifiability 9
2.2.8 Portability 9
2.2.9 Reliability 9
2.2.10 Distributed Real-Time Systems 10
2.3 Software Architecture Styles 10
2.3.1 Pipe-and-Filter System (Data-Flow Model) 11
2.3.2 Layered Style 12
2.3.3 Time Triggered 12
2.3.4 Synchronous/Reactive 12
2.3.5 Process Networks 13
2.3.6 Publish and Subscribe 13
2.3.7 Client-Server (CS) 13
2.3.8 Process Control 14
2.3.9 Finite State Machines 14

2.3.10 Mobile Code 14
2.3.11 Data Abstraction and Object-Oriented Organization 15
2.3.12 Event-Based, Implicit Invocation 16
2.3.13 Repositories 16
2.3.14 Heterogeneous Architectures 17
2.3.15 Rules-of-Thumb for Selecting Styles 17
2.4 Taxonomy for Controller Architectures 18
2.5 Controller Hardware Architecture Review 18
2.5.1 WinRec 19
2.5.2 MUPAAC Architecture 20
2.5.3 Modular CNC System Architecture 22
2.5.4 GEECON Architecture 24
2.6 Controller Software Architecture Review 26
2.6.1 OMAC Architecture API 27
2.6.2 UBC Open Architecture Control System 29
2.6.3 NRC Tripod 32
2.6.4 CLARAty Architecture 34
2.7 Control Architectures 36
2.7.1 Classic Control 36
2.7.2 Reconfigurable Control Architectures 39
2.8 Concluding Remarks 42
3. SYSTEM ARCHITECTURE 44

3.1 Architecture Design Philosophy 44
3.2 Intelligent Modular Controller (IMC) 48
3.2.1 Interfaces 49
3.3 The System Coordinator(s) 49
3.4 Conclusion 49
4. EMBEDDED COMPUTING PLATFORM 51

4.1 Introduction 51
4.2 Java for Real-time System Design 51
4.3 Opportunities and Constraints for Java Embedded Devices 53
4.3.1 The Java Micro Edition (J2ME) 55
4.4 Microcontroller Hardware Selection 56
4.4.1 The aJile Processor 57
4.4.2 The JStick Platform 59
4.5 Motion Controller Hardware 60
4.5.1 Motion Controller Design Options 62
4.5.2 Motion Controller Board Design Criteria 63
4.5.3 Motion Controller Chip Selection 64
4.6 Conclusion 64
5. MOTION CONTROLLER BOARD DESIGN 65

5.1 Introduction 65
5.2 JStick's Peripheral Interface Signals 66
5.2.1 The Phase Locked Loop (PLL) 66

5.2.2 The External Bus Interface 67
5.2.3 HSIO Bus Address Space and Timings 67
5.2.4 JSimm Interface and Signals 68
5.3 The Interface Design 68
5.3.1 Clocking 74
5.4 LM628 DAC Output 75
5.4.1 Encoder Interface 78
5.4.2 Power Supply and Noise Emission 78
5.4.3 Board Schematics 79
5.4.4 The LM628 Hardware Driver 79
5.4.5 Initializing 82
5.4.6 Interrupt Service Routines and digital I/O operations 82
5.5 Conclusion 83
6. THE IMC COMMUNICATION ARCHITECTURE 84

6.1 Introduction 84
6.2 Requirements for Real-Time Communication 85
6.2.1 Environment State Capturing Strategy 87
6.2.2 Co-operation Models 88
6.2.3 Composability 89
6.3 Real Time Network Applications 90
6.3.1 Physical Layer 91
6.3.2 Data Link Layer 92
6.3.3 Network Layer 95
6.3.4 Transport Layer Protocols 95
6.3.5 Application Layer 95
6.4 Automatic Configuration 96
6.4.1 The Zeroconf Protocol 97
6.5 The IMC Communication Architecture 98
6.5.1 Communication and Computing Elements 99
6.5.2 Communication Flow and Control 101
6.5.3 Triggering and Scheduling 103
6.5.4 Automatic Configuration 107
6.6 Conclusion 108
7. CLOCK SYNCHRONIZATION 110

7.1 Introduction 110
7.2 Time 110
7.2.1 Properties of Physical Clocks 111
7.2.2 Global Clocks 112
7.2.3 Failure Mode 113
7.3 The Synchronization Problem 113
7.3.1 Internal Synchronization 114
7.3.2 External Synchronization 115
7.4 The IMC Clock Synchronization Architecture 117
7.4.1 Assumptions and Properties 117
7.4.2 Analysis 119

7.5 Conclusion 120
8. TRAJECTORY PLANNING 121
8.1 Introduction 121
8.2 Planer Motion Trajectory Planning 121
8.2.1 Interpolation by Search Technique 122
8.2.2 Linear Interpolation by Digital Differential Analyzer (DDA) 123
8.2.3 Circular Interpolation 125
8.3 Robot Motion Planning 127
8.4 Other Interpolation Methods 129
8.5 Implementation on the IMC 130
8.6 Conclusion 134
9. THE IMC SOFTWARE ARCHITECTURE 136
9.1 Software Development Phases 136
9.1.1 Analysis 136
9.1.2 The Design Phase 137
9.1.3 Implementation 137
9.2 The IMC Architecture Software Abstraction Development 137
9.2.1 Analysis 138
9.2.2 Design 139
9.3 The IMC Domain 139
9.3.1 Database Abstractions 140
9.3.2 Coordination Abstractions 144
9.3.3 Driver Abstractions 151
9.3.4 Network Abstractions 152
9.3.5 User Interface and Plug-And-Play Abstractions 153
9.4 The System Coordinator Domain 160
9.4.1 The Graphical User Interface (GUI) 161
9.4.2 Coordination Abstractions 166
9.4.3 The Database Abstraction 170
9.4.4 The Interpolator Component 172
9.4.5 Network Abstractions 175
9.5 The IMC Real-Time Coordinator 175
9.5.1 Real-Time Services 176
9.6 Conclusion 178
10. SYSTEM EVALUATION 179
10.1 Introduction 179
10.2 Sampling Time and Communication Latency 179
10.3 Block Processing Time and Real-Timeliness 180
10.4 Synchronicity 180
10.5 Positioning Accuracy 182
10.6 Architectural Flexibility 185
10.7 Conclusion 186
11. CONCLUSION AND DISCUSSIONS 187

11.1 Overview 187
11.2 Summary of Results 187
11.2.1 Control Architecture Review Summary 188
11.2.2 Recapitulation of the IMC Architecture 189
11.2.3 Prototype Development Summary 189
11.2.4 Summary of the IMC Communication Architecture 189
11.2.5 Summary of Computational and Software Models 190
11.2.6 Summary of the Controller Performance 191
11.3 Research Achievements 191
11.4 Discussions and Future Research Direction 192
REFERENCES 194

APPENDIX A. THE IMC HARDWARE..201

A1 IMC Motion Control Board Schematics.. 201
A2 IMC Controller BOM.. 207
A2 Motion Control Chips 209
APPENDIX B: IMC SOFTWARE INTERFACE..211

B1 com.IMC.database.. 212
Class Data 212
Class PushPullData 214
Class FIFO 216
Class FileServer_ConfigFiles 216
Class FileService 218
Class JmDNS_Coordinator_Data 219
B2 com.IMC.coordination 220
Class StateCoordinator 220
Class Counter 222
Class Device 222
Class EncoderReader 223
Class Interpolation_Server 223
Class Interpolation_Server_Starter 224
Class MainClass 225
Class Monitor 225
Class MultiCasted_States 226
Class StateBuffer 226
B3 com.IMC.servlets 228
Class PositionDump 228
Class ConfigureDevice 228
Class ControllerInfo 230
Class EditJmDNS 231
Class JmDNS_Coordinator 232
B4 com.IMC.drivers 233
Class Reference_Switch_Driver 233
Class GPIOPinA3 234
Class HSIO_Driver 234

Class JStickTimer_tc2 236
Class LimitSwitch_Left 237
Class LimitSwitch_Right 237
Class LM628 238
Class Board_Clock 244
Class LM628_Interrupt 244
Class MotorAmp 244
B5 com.IMC.network 245
Class TCPServer 245
Class DatagramServer 245
Class MulticastServer 245
APPENDIX C: COORDINATOR SOFTWARE INTERFACE 247
C1 com.coordinator.database.. 248
Class Traj_Configuration_Data 248
Class Data 249
Class DataTranspose 252
Class FileDataToArrayConverter 253
Class GCodeParser 253
Class GCodeSender 254
Class JmDNS_DATA 254
C2 com.coordinator.coordination 254
Class ControllerIO 254
Class Trajectory_Server 256
Class JmDNS_Coordinator 257
Class JmDNS_Event_Server 258
Class Monitor 258
Class SynchFlag 259
C3 com.coordinator.interpolation.. 260
Class Transmission_Flag 260
Class Interpolator 260
Class Transmission 261
Class Transmission_ACK 261
C4 com.coordinator.GUI 262
Class TrajTable 262
Class MainApplication 262
Class MainGUIFrame 262
Class PIDTable 265
Class TrajDataApplication 266
Class TrajDataFrame 266
C5 com.coordinator.network 268
Class UDP_Client 268
Class DatagramSender269
Class McastDirect 271
Class McastFlag 271
Class MulticastSender 271
Class TCP_Client 271

LIST OF TABLES

Table 4.1: Java Hardware Comparison 53
Table 5.1: Chip Configuration Register 67
Table 5.2: LM628 Signal Levels 68
Table 5.3: LM628 Timing Requirements 71
Table 5.4: Timer Settings 75
Table 6.1: Typical JmDNS Services on the IMC Architecture 108
Table 9.1: Software Packages 139
Table 9.2: The IMC Database Components 140
Table 9.3: Coordination Components 144
Table 9.4: Driver Component 151
Table 9.5: Network Component 152
Table 9.6: Servlets Component 155
Table 9.7: System Coordinator Software Packages 161
Table 9.8: GUI Component 161
Table 9.9: System-Coordinator Coordination Components 166
Table 9.10: System Coordinator Database Sub-Components 170
Table 9.11: System Coordinator Interpolator Sub-Components 173
Table 9.12: Network Package 175
Table 9.13: Real-Time Coordinator Software Packages 175
Table 9.14: Real-Time Service Components 176
Table 9.15: Real-time Coordinator Command Structure 176
Table 11.1: Summary of Research Objectives 188
Table 11.2: Performance Evaluation Results 192
Table A1: Bill of Materials (2004) 207
Table A2: IMC Microcontroller, Cables & Router (2003) 209
Table A3: Comparison of Motion Control Chips 210
Table B1: IMC Software 211
Table C1: System Coordinator Software 247

LIST OF FIGURES

Figure 2.1: MUPAAC Architecture 20
Figure 2.2: MUPAAC Software Architecture.... 21
Figure 2.3: Modular CNC System Architecture 23
Figure 2.4: GEECON Architecture Implementation 25
Figure 2.5: GEECON Architecture 26
Figure 2.6: OMAC Architecture.... 28
Figure 2.7: The UBC Open Controller Reference Architecture.... 30
Figure 2.8: Open Configuration System Software Architecture 31
Figure 2.9: Three-Tier Computing Hierarchy for the NRC-IMTI Tripod 33
Figure 2.10: Tripod Computing Architecture Variations 34
Figure 2.11: CLARAty Implementation on Various Rover Platforms 36
Figure 2.12: Classification of robot control problems 37
Figure 2.13: Model Reference Adaptive Control (MRAC).... 39
Figure 2.14: Control Configuration – General Strategy 40
Figure 2.15: Hierarchical Control Reconfiguration Structure 41
Figure 3.1: The IMC Hardware Architecture.... 45
Figure 3.2: The IMC Reference Architecture 46
Figure 4.1: The Java System Architecture 52
Figure 4.2: The aJile JEM2 Processor (aJ-100, 2001) 58
Figure 4.3: aJ-100 Architecture (aJ-100, 2001) 59
Figure 5.1: Peripheral Motion Controller Board Architecture 65
Figure 5.2: aJile PLL Circuit Diagram (aJ-100, 2001) 66
Figure 5.3: JStick LM628 Interface.... 69
Figure 5.4: Interface Architecture for Multiple LM628 70
Figure 5.5: HSIO Timing Diagram with Bus Speed of 7.37 MHz.... 72
Figure 5.6: HSIO Timing Diagram with Bus Speed of 12.9 MHz Showing Violations73
Figure 5.7: Final HSIO Timing Diagram with Bus Speed of 12.9 MHz 74
Figure 5.8: DAC – LM628 Interface Architecture 77
Figure 5.9: Receiver Line Filter for Single-Ended Totem Pole Encoder.... 78
Figure 5.10: Pseudo-code for High Speed I/O (HSIO).... 80

Figure 6.1: Timing in an Event Triggered System.. 87
Figure 6.2: Timing in a Time-Triggered System 88
Figure 6.3: The OSI Protocol Stack 91
Figure 6.4: Communication and Computing Elements: 100
Figure 6.5: Communication Flow.. 101
Figure 6.6: Produce-Consumer Co-operation between the RC and IMC Nodes........ 103
Figure 6.7: TT communication with TT processors 105
Figure 6.8: ET Communication with TT Processor... 106
Figure 6.9: Multicast DNS Query 107
Figure 7.1: IEEE1588 Precision Time Protocol Architecture 116
Figure 7.2: Clock Synchronization Capture on a Logic Analyzer 120
Figure 8.1: Circular Interpolation.. 126
Figure 8.2: Flowchart for Different Trajectory Configuration Modes 131
Figure 8.3: Flowchart for Coordinated-Motion 134
Figure 9.1: The IMC Architecture Software Components 138
Figure 9.2: IMC Configuration Servlet II 158
Figure 9.3: IMC Configuration Servlet II 158
Figure 9.4: JmDNS Service/Discovery Browser 159
Figure 9.5: JmDNS Service/Discovery Editor Webpage 160
Figure 9.6: Main GUI Browser... 163
Figure 9.7: Trajectory Editor I.. 164
Figure 9.8: Trajectory Data and Configuration Browsers 164
Figure 9.9: Trajectory Editor Frames 165
Figure 9.10: PID Filter Parameter Editor... 165
Figure 9.11: Interpolator Flow Diagram 174
Figure 10.1: Timing Variations–Uncompensated Delays 181
Figure 10.2: Timing Variations– Compensated Delays.. 181
Figure 10.3: Timing Variation– Synchronization by Interrupts.................................. 182
Figure 10.4: Linear Trajectory.. 182
Figure 10.5: Circular Trajectory – 5-mm radius 183
Figure 10.6: Radial Error – 1-mm radius 183

Figure 10.7: Radial Error – 5-mm radius 184
Figure 10.8: Radial Error – 25-mm radius 184
Figure 10.9: Combined Linear and Circular Paths 185
Figure 10.10: Tripod Slider Displacements 186
Figure A1: Motion Control Board (3.5mm x 3.2mm) 201
Figure A2: LM628 Pin-out to Header (H1), and Clock 202
Figure A3: Encoder Interface and I/O to JStick SIMM Interface 203
Figure A4: AD667 DAC Interface with Logic Devices 204
Figure A5: AD667 DAC Output Circuit 205
Figure A6: Filters and Power Supply 206
Figure A7: IMC Ensemble 207

LIST OF ABBREVIATIONS

$offset_i^{jk}$	Offset of microtick *i* between clocks *j* and *k* with the same granularity
δ	Network latency
δ_{max}	Maximum network latency
δ_{min}	Minimum network latency (delay)
Δu	Displacement step
Δx	Incremental displacement in axis *x*
A, B, I	Encoder output signals
aJile	Manufacturer of aJ100 and aJ80 microprocessors that directly execute Java bytecodes in hardware
Architecture	The structure of specific components (such as hardware software components) and the way they interact
ASIC	Application Specific Integrated Circuit
BLU	Basic Length Unit of a Machine indicates its precision
C	Chord error (circular interpolation)
Client-Server	Cooperation model based on Client node(s) that request the service(s) of server(s)
CLKO	Clock output
C_{max}	Maximum chord error
CMOS	Complementary Metal-Oxide Semiconductor; a major class of integrated circuits.
CNI	Communication Network Interface
COTS	Commercial-Off-the-Shelf
CSMA	Carrier Sensing Multiple Access
CSn	Chip Select n; n = integer
CTC	Continuous Path Control
DAC	Digital to Analog Converter
D-Type Flip-Flop	A pulsed digital circuit capable of serving as a one-bit memory. D (data) – type is one of the four basic types
EBI	aJ100 External Bus Interface

ET	Event-Triggered
Fieldbus	Control network based on a serial bus
fr^z	Reference clock frequency
f_x and f_y	x and y axes velocities.
g	Granularity: duration between two consecutive microticks of a clock
g^k	Granularity of a clock k
GPIO	General purpose I/O
hex	Hexadecimal
I/O	Input/Output
IMC	Intelligent Modular Controller
J or ε	Latency jitter; Difference between minimum and maximum latencies
Java bytecode	Machine-independent code generated by the Java compiler and executed by a Java interpreter.
JDK	Java Development Kit
JNI	Java Native Interface
JStick	aJile-based microcontroller by Systronix
JVM	Java Virtual Machine
k	Time interval (in the context of trajectory planning)
LSB	Least Significant Bit
microtick	Periodic event generated by a clock oscillating mechanism
$microtick_i^k$	microtick i of clock k is identified
MSB	Most Significant Bit
N	Number of interpolation steps (in the context of trajectory planning)
N_1	Number of interpolation steps in acceleration region
N_2	Number of interpolation steps in constant velocity region
N_3	Number of interpolation steps in deceleration region
n-DOF	n Degrees of Freedom

Node	An element on a network consisting of at least a host computer and a local communication network interface
NRC-IMTI	National Research Council Integrated Manufacturing and Technology Institute
OS	Operating System
PID	Proportional-Integral-Derivative filter for a controller
PKM	Parallel Kinematic Mechanism
PLL	Phase-Locked-Loop: a closed loop frequency control system based on the phase sensitive detection of phase difference between the input and output of the controlled oscillator
PnP	Plug-and-Play
prescalar	A continuous count-down timer that divides aJile's internal peripheral clock (or an external clock) by *PrescalerReloadRegisterValue – 1*
PrescalerReloadRegisterValue	value of the *prescaler* register
PS	Port Select for LM628
$P_s(x_s, y_s)$ and $P_e(x_e, y_e)$	Starting and end points of a line
PTP	Point-to-Point motion
Publish-Subscribe	Cooperation model based on publishers and subscribers of information
$R(t)$	Reliability
RC	IMC real-time coordinator
RD	Read control strobe
ReloadRegisterValue	Reload register value (aJile)
RST	Reset pin
SC	IMC system coordinator
T	Interpolation period (in the context of trajectory planning)
T_c	clock periods
TDMA	Time Division Multiple Access

T_i	Interpolation time interval (in the context of trajectory planning)
TT	Time-Triggered
TTL	Transistor-Transistor Logic: A class of digital circuits built from bipolar junction transistors (BJT), and resistors.
T_w	Timing duration in CLKO periods
VME bus	A 32-bit bus developed by Motorola
WR	Write control strobe
z	Reference clock
$z(e)$	Absolute timestamp of event e
λ	Failures/hour
ρ	Clock Drift
Π	Clock precision

1. INTRODUCTION

1.1 Overview

This thesis presents the design of a modular reconfigurable controller for machine tools. This is not a new concept, as the literature in the subsequent chapters reveals; however the design breaks new ground by incorporating embedded technology using commercial-off-the-shelf (COTS) components, flexible architectural design patterns based on object-oriented technology, plug-and-play, and web-design into a distributed control system. All these properties are harnessed by well-conceived and novel hardware, software, and communication architecture designs. The treatise takes a tour through the different design stages from concepts to production, and also gives a rich background of the related state-of-the-art. Experimental results and future research directions wrap up the main thesis body. The rest of this chapter gives the motivation for the research, goals and contributions, and the organizational structure of the thesis.

1.2 Motivation

The Mechatronic approach in modern design of control systems inevitably involves embedded technology. This approach is gaining a lot of attention due to the growing demand for distributed real-time systems. Indeed, the gradual paradigm shift from centralized systems is justified in many ways. The most compelling reasons are the needs for less complexity, more processing power, flexibility, and greater fault-tolerance. A typical distributed control system (DCS) consists of several processing nodes connected by a communication network. The network presents a duo of both convenience of connectivity, and inconvenience of dealing with real-time situations. For this reason, some control designers treat the DCS arena with caution, quite reluctant to breakaway from tried-and-tested microprocessor communication systems such as backplanes. However, designs based on such systems have limited flexibility in terms of scalability, unlike serial communication systems where a few serial lines could connect many elements. Currently, there are several such systems, collectively called fieldbuses, which are employed in many

industries to operate peripheral devices such as sensors and actuators. Fieldbus systems have roots in automotive systems where data rates are only a few kilobytes per second. Apparently, their bandwidths have not changed substantially since their debut some twenty years ago. Moreover the current market is now inundated with fieldbus products, and this poses a challenge to standardization and interoperability. Industries need interoperable devices and techniques to reduce integration costs of factory information systems (Dugenske et al., 2000). In the midst of evolving communication standards, Ethernet has stood out as the most consistent and robust, though its presence in low-level control is almost non-existent due to its non real-time properties. Against this backdrop, this research investigates the constraints and opportunities for using Ethernet and proposes and implements a design scheme to realize real-time control to meet stringent time demands. The communication architecture uses a producer-consumer coordination model for real-time data communication over a segmented network, and a client-server approach for point-to-point transactions. Moreover, the proposed scheme employs a time-triggered (TT) approach to schedule tasks for the network. Unlike other TT approaches – normally branded as inflexible, the proposed scheme does not need to be configured explicitly. Furthermore, an auto-configuration protocol that enables devices on the network to be cognizant of the operational environment has been successfully integrated in the architecture. This scheme allows devices on the network to automatically detect configuration changes, and react according to local event-service routines.

Another important aspect of the architecture is the use of an object-oriented architectural style. This style of programming leverages reuse, fast development, and high quality software semantics. Presently, C++ and Ada are the most commonly used object-oriented tools in embedded system designs. Both are robust real-time programming tools, but while C++ is prone to poor readability and maintainability, Ada is large and complex. Breaking from the norm, we use Java as the sole programming tool. Java adds flexibility by being platform-independent with rich support for networking. However, not many real-time applications have been written in this language due to its inherent tardiness. For this reason, we resorted to Java-based processors with native Java machines. This led us to design customized motion controller boards based on COTS components. Test results

demonstrate the capabilities of the design. In the section below, the main objectives and goals of this research are presented.

1.3 Goals and Contributions

The proposed research aims to advance reconfigurable controller architecture to a new limit by addressing the above issues. The research is a collaborative effort between the Mechanical Engineering Department of the University of Saskatchewan and the Integrated Manufacturing and Technology Institutes of the National Research Council (IMIT-NRC). The following list summarizes the research objectives for the proposed design:

Primary Objectives:

1. A generic framework for a modular reconfigurable control architecture. The framework addresses software and hardware requirements, and also the communication structure.
2. A small and simple design that fits into embedded low-cost platforms.
3. A working prototype, not just concepts and simulations of the architecture.

Secondary Objectives:

Literature review on the current work being done in the field of design and to identify those areas that require additional investigations.

1. A critical review of the state-of-the-art in control architecture, distributed communication paradigms, and reconfigurable networked systems.
2. A synchronization algorithm and protocols to enable Ethernet to be used for real-time control.
3. An operational software architecture based on modularity and reusability.
4. Demonstration of the strengths of the proposed design.

Contributions:

The main contributions of the thesis are as follows:

1. Embedded technology is a cost-effective approach to motion control design. C++ and Ada have dominance in this field, but Java has certain unique and superior strengths albeit some weakness in real-time design. In Chapters 4 and 5, a Java-based design is presented that utilizes COTS components. The concepts are simple (simplicity is the governing principle), and can easily be replicated. To the best of our knowledge, no

literature has revealed a hard real-time motion controller design thoroughly based on embedded Java technology.

2. Ethernet is the de facto network standard, but is seldom considered for time critical events due to its inherent 'sluggishness'. Therefore many distributed systems rely on fieldbuses or microprocessor communication hardware (e.g. VME). However, the real-time communications market is unregulated, and is therefore cluttered with many different systems, which normally require special middleware and hardware systems to make them interoperable. Moreover, not many of them have the robustness and flexibility of Ethernet. In view of this, an Ethernet-based real-time communication architecture with implicit clock synchronization has been implemented and demonstrated as part of the overall architecture. The technology also enables the controller sub-component design to incorporate embedded web-servers for remote monitoring and system configuration.
3. Zeroconf protocol was developed by Apple to enable networked devices to automatically reconfigure (plug and play) without the need for high level intervention. A subset of the Zeroconf protocol, JmDNS, is written in standard Java (J2SE). With regard to this protocol, two ideas are realized in this research: First, the protocol has been re-engineered and ported to a subset of J2SE called the Java micro-edition (J2ME) commonly used in embedded Java systems such as personal device assistants (PDA). Secondly, the protocol has been successfully demonstrated as an automatic configuration tool for controllers and other shop floor devices.

1.4 Organization of Thesis

The structure of the thesis is outlined in this section. In Chapter 2, a detailed review of control architectures is discussed and critiqued. Typical requirements for control and software architecture are presented to enable a comprehensive review. A snapshot of the proposed architecture codenamed the IMC (Intelligent Modular Axis Controller) architecture is presented in Chapter 3. The design is based on a layered reference model, and modularity, simplicity and flexibility are the governing principles. A detailed review of the constraints and opportunities for Java technology in real-time design is presented in Chapter 4. Inferences drawn from this enabled us to select and to understand the implications of using JStick, which is a Java-based COTS microcontroller. An overview

of the relevant characteristics of the microcontroller is outlined. The premise for the selection of a motion controller chip (LM628) is also included in this chapter. In Chapter 5, the hardware architecture is described in detail. The architecture consists of control modules (IMC) dedicated to each machine axis, and a host computer (system coordinator). Each IMC is made up of a JStick as the host microcontroller and a motion controller board purposefully designed for this research. Details of the board design are provided: This includes the procedure for integrating it with the JStick such as timing analysis. Chapter 6 details communication architecture concepts and principles. Following this, the system communication architecture is described. A Time-Triggered approach enhanced with a producer-consumer co-operation model, is employed to realize real-time communication on a switched-Ethernet network. Analysis of the computation and communication model is also presented. Clock synchronization in distributed systems is discussed in Chapter 7; an external clock synchronization model developed for the IMC architecture is described. Chapter 8 gives an overview of trajectory generation schemes and the methodologies adopted for this project. The software architecture framework is vividly described in Chapter 9. Experiments to verify the architecture are provided in Chapter 10, and finally conclusions and future research directions are outlined in Chapter 11. Schematic diagrams of the motion controller board, BOM (Bill of Materials) including prices, and the software interfaces are appended.

2. ARCHITECTURE DESIGN CONCEPTS AND REVIEW

2.1 Robot Control Architecture, Introduction

Robot control architectures embody several different notions and implications, particularly architectural styles and structures. Architectural structure shows how a system is decomposed into subsystems, and how subsystems interact. The computation and communication underpinnings of a given system invariably reflect a style. For example, one system might use a publish-subscribe message passing style of communication, while another may use a more synchronous client-server approach (Coste-Maniere and Simmons, 2000). Most often the holistic architecture is realized only at the working stage. This is unfortunate, since a well-conceived architecture can have many advantages in the specification, execution, and validation of robot systems. This chapter serves as a roadmap to developing a robot control architecture framework. Generally, a framework refers to the structure external to an architecture which organizes information about the architecture and its application (Kramer and Senehi, 1993).

2.2 Architectural Properties

The architectural style employed has a direct impact on the performance of the overall system. For example, the *pipe-and-filter* software style supports components' reusability and configurability of the application by applying generality to its component interfaces. However, components are constrained to a single interface type. Some salient properties of architectures are discussed in the following sections.

2.2.1 Dealing with Complexity

A daunting challenge is the need to manage the complexity of interactions between the system and its environment, and interactions between individual units of a system (Coste-Maniere and Simmons, 2000). One way to achieve simplicity is through modularity within a given structure. The global system complexity can be decomposed into smaller components with well-defined abstraction levels and interfaces between them.

Another means of curtailing complexity is to provide expressive languages and tools, e.g., ALPHA, and high-level languages such as TCA and TDL (Coste-Maniere and Simmons, 2000). Architecture description languages (ADL) are particularly useful for architecture-based development and formal modeling notations and analysis (Medvidovic and Taylor, 2000). The software architecture must provide a basis for complexity management by providing abstract models of the system under development, which is necessary in comprehension, cross-domain communication, verification, validation and maintenance (Chen, 2001). Coupling and cohesion issues must also be addressed by the architecture. Coupling refers to the way modules are connected, while cohesion indicates the degree of relatedness of sub-modules, or components within a module. An ideal system provides low coupling and high cohesion without violating performance parameters (Chen, 2001).

2.2.2 Execution

The architecture should also define the run-time execution of the software. This includes real-time responses, appropriate goal-directed behavior and reliable reactivity to environmental changes. The issue on real-time provokes this famous definition:

> A real-time computer system is a computer system in which the correctness of the system behavior depends not only on the logical results of the computations, but also on the physical instant at which these results are produced (Kopetz, 1997).

There are two variants of real-time systems: In a hard real-time system, timing violations are to be avoided at *all* times. In contrast, a soft real-time system can tolerate some degree of timing violations. Typical tasks for a real-time system include data collection, digital control and man-machine interactions. The architecture has to describe, quantify or specify the *temporal requirements* of the system. These include timing of events (e.g., deadlines), arrival patterns of events (e.g., sporadic or periodic), and the triggering policies required. Goal-directed behavior refers to the mechanism within the architecture to manage tasks and behaviors such as task decomposition and behavior arbitration, and for managing the interactions between tasks and behaviors such as resources management, multitasking and temporal sequencing. Task management facilitates concurrent execution in a single process or by a collection of distributed processes. For reliable reactive behaviors, the architecture can provide software support for monitoring the environment and invoking exception handlers, if necessary. The extent

of reactivity depends on the real-time capabilities of the system. The management of exceptional conditions also includes support for cleanly terminating tasks, and if necessary recovery strategies.

2.2.3 Openness

Openness is a buzzword in the control systems community. Open architecture control systems offer services according to standard resources, and/or standard rules that describe the syntax and semantics of those services (Tanenbaum and van Steen, 2003; Hong, et al, 2001). This approach forces all manufacturing vendors to conform to an agreed standard, thereby promoting integration and interoperability. It is becoming more acceptable for open-architecture control system to possess the common capabilities and functionalities offered by standard platforms: these include standard computing architecture, standard processors, standard operating systems, and standard and widely used programming languages. Moreover, openness promotes application of user-specified functions (Mehrabi et al., 2000; Chesney, 1998).

2.2.4 Performance

The performance of an architecture implementation is bound by first the application requirements, then by the chosen interaction style, followed by the realized architecture, and lastly by the implementation of individual component. As an example, if the application requires that data be located on system X and processed on system Y, then the software cannot avoid moving that data from X to Y. Also, an architecture cannot be any more efficient than its interaction style allows; e.g., the cost of multiple interactions to transport data from X to Y cannot be less than that of a single interaction from X to Y. Lastly, regardless of the quality of an architecture, interactions cannot take place faster than the capacity individual components can endure. The quality of a distributed control architecture hinges on its network performance, which is measured by data throughput, overhead, bandwidth and usable bandwidth (Fielding, 1999). From the end-user perspective, network performance metrics are latencies, jitter and ultimately the ability of the controller to track a prescribed motion path accurately and efficiently within the dynamic constraints of the system.

2.2.5 Scalability

Scalability is the ability of the architecture to support many components, or interactions among components within an active configuration. Scalability can be enhanced by simplifying components and decentralizing interactions between elements. Another complement approach is the proper control of monitoring or user interactions and configurations. The architecture style influences scalability by determining the location of application states, the extent of distribution, and the coupling between components (Fielding, 1999). Scalability is also affected by the frequency of interactions and timeliness requirements of data transfer.

2.2.6 Simplicity

Allocation of functionality to the individual components to reduce complexity enables easier understanding and implementation of components and the overall architecture.

2.2.7 Modifiability

Modifiability is the ease with which a change can be made to the architecture implementation. Modifiability is also a measure of evolvability, customizability, reconfigurability, and reusability (Fielding, 1999). Customizability refers to modifying a component at run-time, specifically so that the component can then perform an unusual service. A component is customizable if it can be extended by one client of that component's services without adversely affecting other clients of that component.

2.2.8 Portability

Portable software can run in different environments. Architecture styles that support portability include those that move code along with the data to be processed, such as the virtual machine and mobile agent styles, and those that constrain the data elements to a set of standardized formats.

2.2.9 Reliability

Reliability refers to the robustness of an architecture to failure at the system level in the presence of partial failures within components, connectors, or data. By definition

(Kopetz, 1997), if a system has a constant failure rate of λ failures/hour, then the reliability R at time t is given by

$$R(t) = e^{(-\lambda(t-t_0))};$$

where t-t_0 is given in hours. The inverse of the failure rate is referred to as the *Mean-Time-To-Failure* (*MTTF*).

2.2.10 Distributed Real-Time Systems

There are several definitions of distributed systems found in the literature. According to a general definition given by Tanenbaum and van Steen (2003), "A distributed system is a collection of independent computers that appears to its users as a single coherent system". Their definition has two implications: Firstly, the machine hardware is autonomous. Secondly, regarding software, the users think they are dealing with a single system. In terms of distributed system hardware, they divide all computers into two groups: those with shared memory (multiprocessors) and those without shared memory, i.e. multi-computers. Interconnection networks may be a bus such as a network or backplane, or switched as in the case of the public telephone system. Essentially, it is the software that largely determines the nature of a distributed system. The definition given by Kopetz (1997) is more specific: "If a real-time system is distributed, it consists of a set of (computer) nodes interconnected by a real-time communication network".

A distributed system leverages designing systems into fault-containment regions for easy diagnosing and smothering of faults, i.e., in a well-designed system, faults are dealt with locally and thus do not pervade the system. Moreover, scalability and modifiability is not as restrictive as in the case of a centralized system. However, the communication system can be a bottleneck if not properly designed.

2.3 Software Architecture Styles

Modern day robot control architectures encase intensive software engineering. This necessitates choosing software architecture styles. For example, one has to select a suitable style (or a combination) for network-based applications, which will induce laid out objectives. Researchers are still refining the definition of software architecture. Two views are as follows:

> A software architecture is an abstraction of the run-time behavior of a software system during some phase of its operation. A system may consist of many levels of abstraction and many phases of operation, each with its own software architecture. A software architecture consists of components, connectors, data, a configuration, and a set of architectural properties (Fielding, 1999).
>
> The architecture of a software system defines that system in terms of components and of interactions among those components. In addition to specifying the structure and topology of the system, the architecture shows the intended correspondence between the system requirements and elements of the constructed system. It can additionally address system-level properties such as capacity, throughput, consistency, and component compatibility (Shaw, 1995).

Two common words that show up in various definitions are 'components' and 'connectors'. Components are described by Garlan and Shaw (1993) as *computational components* and by Fielding (1999) as *abstract units of software that provide a transformation of data via their interface*. In Birla et al., (2001) it is described as *a reusable piece of software that serves as a building block within an application*. This approach has the potential to enhance productivity by reducing architectural complexity (Chadha and Welsh, 2000). It is generally accepted that a connector is an abstract mechanism that mediates communication, coordination or cooperation among components (Fielding, 1999; Shaw, 1995).

In this section we examine some widely used styles. Our purpose is to capture the variety of choices available and the implications thereof.

2.3.1 Pipe-and-Filter System (Data-Flow Model)

A pipe-and-filter (PF) system has two types of units: filters (component) and pipes (connector). Filters are responsible for incrementally transforming continuous streams of input data to streams of output data while pipes handle transportation of data streams between the filters, i.e., connections between producer and consumer components are data streams (Garlan and Shaw, 1993). Filters are usually implemented as software processes and pipes as system services. Filters are independent entities and hence are not allowed to share state with other filters. Invariably, a filter has no knowledge about its upstream and downstream filters. Filters are initiated when data is available and run to completion at the end of input data. In other words, computations are triggered by the availability of input data. Several variations of PF exist including those found in the UNIX family, LabVIEW and some database systems (Lee, 2002). The PF styles are suitable for applications where

the problem can be decomposed into a series of independent computations. This kind of configuration implies low coupling and high cohesion, which are necessary for simplicity, modifiability, reusability, and portability. However, it can be difficult to create interactive applications with this style. Secondly, there can be some performance-related problems due to the overheads of parsing and unparsing operations resulting from a fine-grained data stream (Chen, 2001).

2.3.2 Layered Style

A layered system is organized hierarchically, with each layer providing services to the layer above it and using services of the layer below it (Garlan and Shaw, 1993). A function in one layer can only interact with other functions in the same layer or adjacent layers through a protocol (connector). This decoupling enhances evolvability and reusability since each layer represents a group of modules or functions for one class of services. The most widely used application of this kind of architectural style are layered protocols such as the TCP/IP and OSI protocol stacks (Zimmerman, 1980), Windows NT and hardware interface libraries. Layered styles are suitable for systems that can be decomposed into application-specific and implementation-specific functions. The major disadvantage of layered systems is that they add overhead and latency to the processing of data and hence could degrade user-perceived performance (Fielding, 1999).

2.3.3 Time Triggered

In some systems timed events are driven by clocks, which are signals with events that are repeated indefinitely with a fixed period. A number of software frameworks and hardware architectures support this regular style of computation. The time-triggered architecture (TTA) is a hardware architecture that employs this regularity by statically scheduling computations and communications among distributed components.

2.3.4 Synchronous/Reactive

In the synchronous/reactive (SR) style, connections between components represent data values that are aligned with global clock ticks, as with time-triggered approaches. However, unlike the time-triggered approach, there is no assumption that signals have a value at each time tick. This model is ideal for concurrent models with irregular events such as concurrent and complex control logic. Also because of tight synchronization,

safety-critical real-time applications are a good match but this makes distributed systems difficult to model.

2.3.5 Process Networks

In this technique, components are processes or threads that communicate by asynchronous buffered message passing. The sender of the message does not need to wait for the receiver to be ready to receive the message. Process networks (PN) are excellent for signal processing. They are loosely coupled, and therefore relatively easy to parallelize or distribute. They can be implemented efficiently in both software and hardware. The main weakness of PN models is that they are awkward for specifying complicated control logic (Lee, 2002).

2.3.6 Publish and Subscribe

In publish-and-subscribe models, connections between components are through named event streams. A consumer component registers an interest in the stream. When a producer produces an event to the stream, the consumer is prompted that a new event is available. It then queries a server for the event's value.

2.3.7 Client-Server (CS)

The client-server style is the most frequently used architectural style for network-based applications (Fielding, 1999). A server component offering a set of services listens for requests: a client component in need of that service sends a request to the server via a connector. The server may either reject or serve the request and send a response back to the client. A client can be regarded as a triggering process while a server is a reactive process. Requests from clients trigger reactions from servers (Andrews, 1991). Separation of functionality is the main principle behind the client-server constraints. A proper separation of functionality simplifies the server component in order to improve scalability. This simplification usually involves moving all of the user interface functionality into the client component. The separation also allows the two types of components to be modified independently, provided that the interface does not change. There are several flavors of client-server systems, depending on the number of servers and clients in the overall system.

2.3.8 Process Control

Also called the control-loop style (Shaw, 1995), this style is based on the notion of closed loop control. Data flow topology is cyclic between control functions. The style has two types of units: controlled process including process modeling as well as process variables and sensors, and controller including control algorithms and set points. The interactions of these units involved intense data interactions, where the controller receives values of measured process variables at predefined time points and produces control signals to manipulate the controlled process.

2.3.9 Finite State Machines

It is often useful to combine models, especially concurrent ones, hierarchically with finite-state machines (FSM) to get modal models. FSM execution is strictly sequential. A component is called a *state* or *mode*, and exactly one state is active at a time. Connections between states denote transitions or transfer of control between states and execution is an ordered sequence of state transitions. FSM models are excellent for describing control logic and are easily mapped to either hardware or software implementations (Fielding, 1999). However, the number of states can get very large even in simple system. Hence, FMS is often combined with other styles.

2.3.10 Mobile Code

Mobile code styles use mobility to dynamically change the distance between the processing and source of data or destination of results (Fielding, 1999; Fuggetta et al., 1998). A site abstraction is introduced at the architectural level, as part of the active configuration, thereby taking into account the location of the different components. The concept of location makes it possible to model the cost of an interaction between components at the design level. An interaction between components in the same location is considered to have negligible cost compared to an interaction through a communication network. By changing its location, a component may improve the proximity and quality of its interaction, reducing interaction costs and thereby improving efficiency and performance. In all mobile code styles, a data element is dynamically transformed into a component. The virtual machine or interpreter is the main mobile code style (Garlan and Shaw, 1993). The virtual machine style provides code execution in a secured and reliable

way, preferably within a controlled environment. The benefits of VM are the separation between instruction and implementation on a particular platform (portability) and ease of extensibility. One of the most popular implementation is the Java Virtual Machine (JVM), which enables Java to be platform-independent. In the mobile agent style, an entire computational component is migrated to a remote site along with its state, the code it needs, and possibly some data required to perform the task. The main advantage of the mobile agent style is that there is greater flexibility in the selection of when to move the code. An application can be in the midst of processing information at one location when it decides to transfer to another location – presumably in order to narrow the distance between it and the next set of data it wishes to process (Fielding, 1999).

2.3.11 Data Abstraction and Object-Oriented Organization

In the data abstraction and object-oriented approach, data representation and its associated primitive operations are encapsulated in an abstract data type or object. The components of this style are the objects. The object-oriented approach has many favorable properties which have contributed to its widespread use. Because an object hides its representation from its clients, it is possible to change the implementation without affecting those clients (Shaw and. Garlan, 1995), as long as the interface is preserved. Furthermore, the bundling of a set of accessing routines with the data they manipulate enables programmers to decompose problems into collections of interacting agents. However object-oriented systems have some shortcomings. Unlike pipe-and-filter systems, an object must know the identity of another object it wants to interact with. Therefore whenever the identity of an object changes, it is necessary to propagate this update to all objects that explicitly invoke it. Brokered Distributed Objects is a popular intermediary style to facilitate communication between distributed objects (on a network) by reducing the impact of identity (Fielding, 1999). The brokered distributed object style creates name manager components whose purpose is to answer client object requests for general service names with the specific name of an object that will satisfy the request. Although improving reusability and evolvability, the extra level of interaction requires additional network interactions, thereby reducing efficiency. Brokered distributed object systems are currently dominated by the industrial standards such as CORBA within the

OMG (1997) and the international standards development of Open Distributed Processing (ODP) within ISO/IEC (1995).

2.3.12 Event-Based, Implicit Invocation

In systems in which component interfaces provide functions and procedures such as object-oriented systems, components interact by explicit calls on those routines. In contrast, in implicit invocation (also referred to as reactive integration, and selective broadcasting), a component can announce (or broadcast) one or more events. Other components in the system can register for the event by associating a procedure with it. When the event is published, the system itself invokes all the procedures that have been registered for the event. Inferentially, an event announcement *implicitly* causes the invocation of procedures in other modules (Shaw and Garland, 1996). One advantage of this style is that it supports strong reuse. Any component can be integrated into the system simply by registering it for the system events. Another advantage is that it eases system evolution since components may be replaced without affecting the interfaces of other components in the system. The main disadvantage of implicit invocation is that when a component announces an event, it cannot assume that other components will respond to it, or respond correctly. There are several examples of systems with this style such as integration tools, database management to ensure consistency constraints, and in user interfaces to separate presentation of data from applications they manage (Shaw and Garland, 1996). Variants of this style are also used in some network protocols for registering and automatically discovering services on the network.

2.3.13 Repositories

The repository style has two kinds of components; a central data structure that represents the current state, and a group of independent components that operates on the central data store. The choice of interaction between the repository and its external components vary significantly among systems. For example, the repository can be a database if the types of transactions in an input stream of transactions trigger selection of processes to execute. Alternatively, the repository can be a blackboard if the current state of the central data structure is the main trigger for selecting processes to execute. The blackboard style has been used for applications requiring complex interpretations of signal

processing, such as speech and pattern recognition. It is also common in AI systems and systems requiring shared information such as compiler architecture (Garlan and Shaw, 1993). This style presents an efficient way to share large amounts of data since the data is centrally managed. Sub-systems need not be concerned with how data is produced. However, it does not favor efficient distribution and flexibility. Moreover, the sub-systems must agree on a repository data model, and data evolution is difficult and expensive.

2.3.14 Heterogeneous Architectures

While it is important to understand individual architectural styles, most systems involve some combination of several styles (Shaw and Garland, 1996). One way to combine architectural styles is through hierarchy. A component of a system organized in one style may have an internal structure based on a completely different style. A second way to organize styles is to allow a single component to use a mixture of architectural connectors. For example, a component may interact through pipes with other components in the system and accept control information through another section of its interface (Shaw and Garland, 1996).

2.3.15 Rules-of-Thumb for Selecting Styles

The choice of an appropriate architectural style is normally driven by the preference of the end-user and/or implementer, the tools available and the application. However, a cautious analysis of the chosen style could lead to a shorter delivery time and robust product. There are no hard and fast rules governing the selection process, and a typical architecture could consist of several styles as mentioned earlier. Garland and Shaw (1993) have provided some basic rules-of-thumb for choosing styles: Consider pipe and filter style if the problem can be decomposed into sequential stages or if the problem involves transformations on continuous streams of data. If the system involves controlling continuous action, is embedded in a physical system and is subject to unpredictable external perturbation, consider a closed loop control style. If the system consists of tasks/processes and runs on a multiprocessor platform, consider independent component styles. Consider a layered style if the tasks in the system can be divided between application-specific ones and those generic to many applications but specific to the underlying computing platform, or if portability across different platforms is an issue.

2.4 Taxonomy for Controller Architectures

Several architectural styles exist to meet different needs and some attempts have been made to categorize them. Shaw (1995) classifies control architectures according to their software architectural styles and contrasts them along five design dimensions; changes in the processing algorithm, changes in data representation, enhancement to system function, performance, and reuse. Kramer and Senehi (1993) categorize architectures into those that emphasize control and those that emphasize data flow. In (Atta-Konadu, et al., 2004) and (Yook et al., 1998), control architectures for machine tools are classified according to their spatial distributions (i.e., fully centralized to fully decentralized). Ambrose's classification is based on hardware and software control architectures (Ambrose, 1992). In this case, criteria such as speed and modularity are used to evaluate the architectures in each category. There are also classifications that reflect intelligent interactions with the environment (Coste-Maniere and Simmons, 2000) as in the case of autonomous robotic systems. After reviewing literature items, it has become quite evident that designers and developers emphasize either one or more of these categories: hardware architecture, software architecture (system-level) and control architecture. The sections following present a review of some interesting control architectures.

2.5 Controller Hardware Architecture Review

Hardware architectures usually emphasize the computation platforms, their interfaces, and interconnections. In effect the hardware architecture should help readers to understand the underlying execution mechanics and dataflow through the different execution stages. This section reviews four hardware architectures. In all cases the following criteria inspired by (Ambrose, 1992; Kopetz, 1997; James and McClain, 1999) are used to evaluate them:

- Throughput (speed): The potential for high throughput is desirable.
- Communication between modules: The communication mechanism between entities should not present a bottleneck to system performance.
- Functional coherence: The modules or nodes of the system should implement self-contained functions with high internal coherence and low external interface complexity.

- Dependability: This is an indicator of the effect of a module or node failure on the entire system. The architecture should define a fault tolerance mechanism.
- Hardware Modularity: Hardware should be modular, reconfigurable and scalable in order to make the system an open architecture.
- Software Modularity: The controller software should also be modular, reconfigurable and scalable with minimal effort. In order to support evolving trends in control algorithms, the software should have standardized interfaces.
- Relevance to design objectives: Control architectures fall under four levels of specificity. In Domain-level design the architecture applies to a broad area of robot control. Second, in Task-level design, controllers are designed to suit a certain class of robots designed for specific tasks, for example welding. Third, controllers may be optimized to perform one robot algorithm such as inverse kinematics. Lastly, in Robot-level design, controllers are optimized to perform specific algorithms for a specific robot. In this case intrinsic knowledge of the robot geometry and dynamics is utilized to further simplify the algorithm and thus the controller.

2.5.1 WinRec

WinRec (Lee and Mavroidis, 2001) is a PC-based controller designed to run on MS-Windows NT. The project was driven by the need for a low-cost controller (less than US$1000) for academic experiments. Deterministic timing obtained from Windows MSDN library is used in both control and data acquisition. The hardware setup consists of a PC with Intel Pentium II 333 MHz CPU, 128 MRAM, two US Digital PC7166 PC to incremental encoder interface cards, and two Datel PC-412C analog I/O boards. The PC polls sensor data through the data acquisition boards or the encoder interface cards, performs feedback control calculations, and sends the signals to actuators through a digital-to-analog (DAC) converter. The maximum controller loop rate is 200 Hz. Experimental results with a 5 degree of freedom robot and different control laws (PID, LQR and H_2) were presented. This system was not really designed for high-end uses; hence for its purpose to demonstrate control laws, it is quite appropriate. Nonetheless, the single point of control is susceptible to complete system failure. Moreover, since the design (including the software structure) is not modular or open, system modifications could imply a major overhaul.

2.5.2 MUPAAC Architecture

The Multi Processor Architecture for Automatic Control (MUPAAC) architecture is described in Bellini et al. (2003). The project was a joint venture between the University of Florence and two European industries to provide solutions to costs and reconfigurable challenges with current production pipelines of manufacturing industries. The architecture is shown in Fig. 2.1 below.

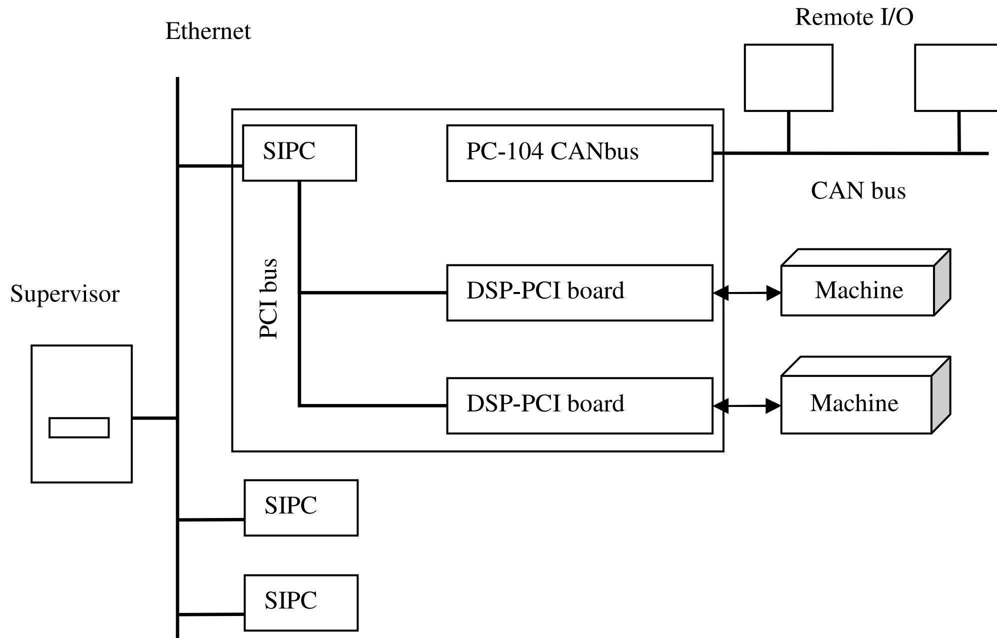


Figure 2.1: MUPAAC Architecture

The hardware architecture consists of three layers: The top layer is the MUPAAC Supervisor which is the general server of the control system of the production. It receives ISO commands from a CAD/CAM workstation and sends them by Ethernet (TCP/IP) to Special Industrial Peripheral Computers (SIPC). The Supervisor can also read alarms/errors detected by the SIPCs. The second layer is the SIPC cluster. These are microprocessor-based systems that execute ISO programs and single instructions coming from the Supervisor. The SIPC interacts with DSP-PCI boards for controlling axes and receiving alarms and synchronization. Finally, the SIPC interacts with remote I/O systems for activating and receiving I/O signals via a CANbus (Controller Area Network serial bus). Each SIPC board has a set of PCI boards for controlling a maximum of 4 remote

axes. The last layer is made up of the DSP-PCI boards. These are based on the Analog Device's AD2106x DSP and can control up to 4 machine axes. Communication between the SIPC and the DSP boards is via the PCI bus. The boards can be plugged on the bus directly without the need for configuration. The Supervisor and SIPC also control the remote I/O boards. These boards are equipped with Intel i8051 or SH7000 CPUs for interpreting messages received via the CANbus. On the CANbus, up to 56 boards can be attached in a plug and play fashion. Furthermore, each board can host 8 I/O modules; therefore each SIPC can accommodate a maximum of 512 I/O modules. The software components (Fig. 2.2) are distributed on the three layers described above.

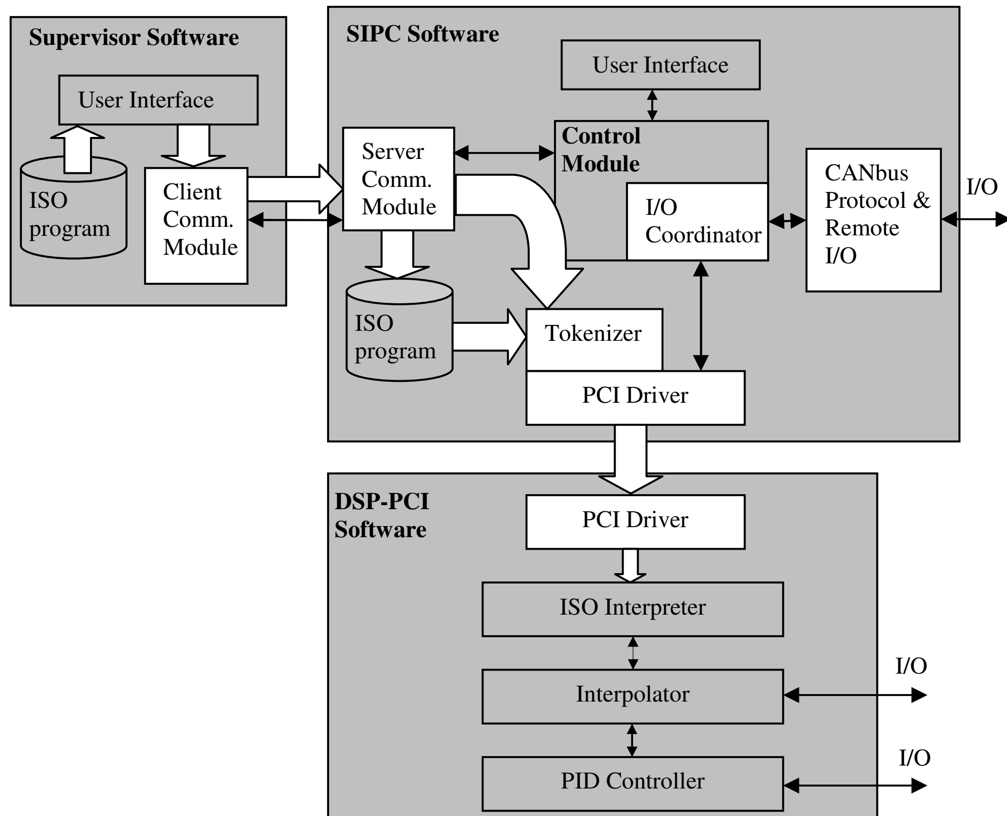


Figure 2.2: MUPAAC Software Architecture

The Supervisor software presents a simple user-interface and a client communication module. The SIPC software is hosted on a WindowsCT Operating System platform and consists of three modules; the *Server Communication Module*, the *Control*

Module and the *CANbus Protocol and Remote I/O Boards Module*. The *Server Communication Module* implements a TCP/IP server to receive ISO programs and commands. Programs are temporarily stored in a RAM file and then each instruction of the program is tokenized and sent to the specified DSP-PCI board by means of a SIPC-PCI protocol. Interaction with the DSP is interrupt driven and based on a dual port RAM. The *Control Module* coordinates the DSP-boards with the *Remote I/O Modules* on the CANbus and presents status information to the user interface. This activity is performed by an *I/O Coordinator Module* which is responsible for exchanging information between the DSP boards and the CANbus I/Os. The module updates the outputs periodically every 10 ms. The *Protocol and Remote I/O Boards Module* manages real-time communication on the CANbus which has a maximum bit-rate of 1 Mbit/s. The DSP-PCI software modules drive the performance of the control system. ISO commands are received from the PCI bus through a local PCI driver and executed accordingly. The DSP supports linear and circular interpolations, implements PID control, controls analog outputs of the board and manages some I/Os such as limit switches. Even though each DSP controller is capable of loop rates lower than 100 μsec, the Number of Entities Processed per Second (NEPS) on the DSP is influenced by communication latency with the SPIC, computing and interrupt servicing times, and the number of controlled axes. Nonetheless, NEPS of 1429 is achievable in the worse case, i.e., 4 axes in circular interpolation. Performance evaluation for two or more DSP boards connected to one SIPC was not presented but at a glance, it is evident that performance will degrade substantially.

The architecture does not explain how the SIPC synchronizes its clock with the slave DSP-PCI boards and remote I/O modules. Moreover, it has been presented as a domain-level type of design, but its internal structure is more tailored for CNC machines than robots. It is also more of proprietary system than a vendor-neutral architecture. Therefore, its robustness to obsolescence is quite questionable. Nonetheless, this is a very interesting architecture which follows many modern design paradigms.

2.5.3 Modular CNC System Architecture

The modular CNC System was designed by researchers at the University of British Columbia (Altintas, et al., 1996). The architecture is hierarchical with two independent backplane buses (see Fig. 2.3). A *Real-Time Master* computer (PC) is in charge of the

AT/ISA primary bus, and other processors (referred to as modules) dedicated to various monitoring or control tasks may be added to the bus if there are available slots on the backplane. All processors on the bus are memory-mapped with the *Real-Time Master* for access to shared memory on the *CNC Master Controller*. The memory area is refreshed periodically at 1 ms by the *Real-Time Master*, which collects the necessary position, cutting forces, etc provided by the processors. The core module in the primary bus is the *CNC Master Controller* computer, which is an off-the-shelf TMS320C30-based DSP board. The *CNC Master* executes and provides precision NC tool path trajectory and velocity values to each feed drive control unit on the machine. It also provides expansion through a memory-mapped secondary bus referred to as the *CNC bus*. An Intel 80C196KC based embedded microcontroller is dedicated to control each axis of the machine tool.

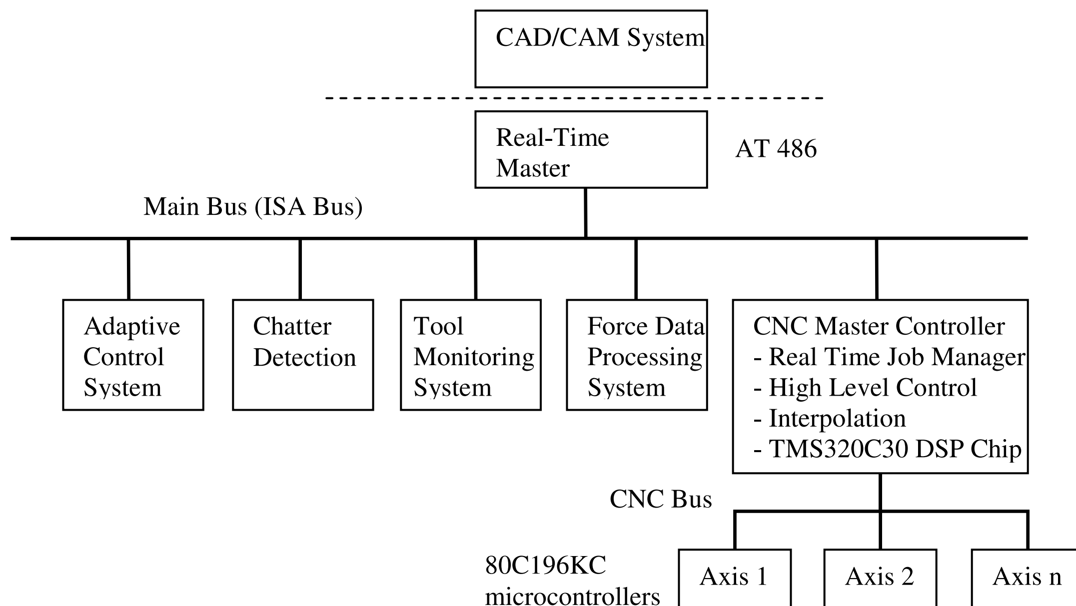


Figure 2.3: Modular CNC System Architecture

Axes position loops are synchronously closed at 0.2 ms by a common high-priority interrupt line. Each axis has a second interrupt line of lower priority connected to the *CNC Master's* high speed digital output line via the CNC bus. The interrupt is generated by a programmable timer on the *CNC Master*. When an interpolation interrupt is received, the

CNC master or linear interpolator in the axis controller calculates the next position and places it in a reference position register. When the controller receives the axis position loop closure interrupt, the error is calculated from the actual position, and sent through a position control filter to an internal PWM register. A circuit in the controller produces volt command signals for dc servo amplifiers via a 2-kHz low pass filter.

The design is a classic task-specific architecture but can be modified for other robotic tasks. It is a tightly-coupled design which trades off flexibility for hard real-timeliness.

2.5.4 GEECON Architecture

The vision for the Generic Embedded Control Node (GEECON) project was the development of an embedded controller which can interface to any actuated mechanical device (Sorensen, 2003). The system architecture, shown in Fig 2.4 is based on a six layer reference model.

The functional layers are distributed on a C*entral Control Node*, *GEECON* controllers (one per axis), and mechanical host modules. These are the shaded regions of Fig 2.4. The C*entral Control Node* consists of three layered software modules: The top module handles high level applications such as motion planning and user interfaces. The second module implements the part of the execution layer which require global knowledge of the robot such as kinematics, interpolations, etc. At the bottom is a module for interfacing and interacting with the distributed part of the system through a network. The *GEECON* controller is the main thrust of the research. It is designed to be embeddable in a mechanical axis such as a robot arm (Fig 2.5).

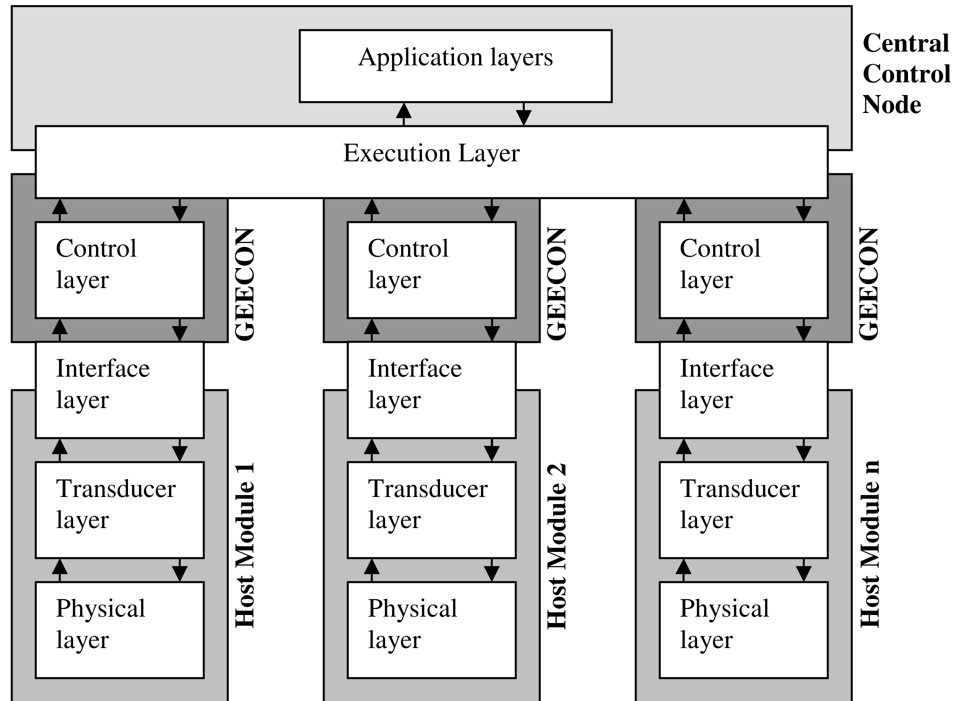


Figure 2.4: GEECON Architecture Implementation

The embedded controller is driven by a Texas Instruments based DSP board – Orys Gmbh, which has a serial bus and implements PID control and a sample rate converter. To make it as generic as possible, the *GEECON* mates with a reconfigurable I/O logic board developed for this research. The logic board features a Xilinx FPGA (Field Programmable Gate Array) and can be programmed to interface with a large range of industrial I/O needed for robot control. Information flow (e.g. nested control loops) within the entire architecture is designed to cover typical bandwidth requirements; the position control servo loop frequency is 4-MHz, while the high-level control loop (executed by the C*entral Control Node*) supplies set-points to the *GEECON* nodes at 200-Hz. These are reconstructed to high resolution paths for the position control by the *GEECON* nodes.

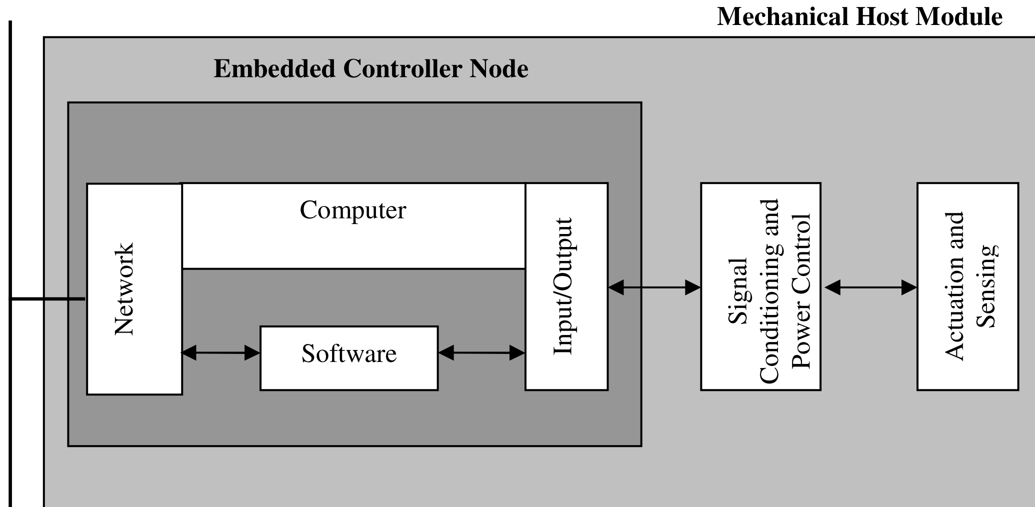


Figure 2.5: GEECON Architecture

The *GEECON* is a well conceived and fairly ambitious project which is still in its infancy stage. Further tests will have to be done to validate its objective as a domain-level solution. Designers may have to address clock synchronization between the central controller and the embedded controllers (GEECON), and re-engineer their software for greater flexibility.

2.6 Controller Software Architecture Review

Software architecture deals with abstractions, behaviors, and interfaces of the controller's software structure. Concepts discussed in the early sections of this chapter apply here. Three architectures will be reviewed in the backdrop of the following, most of which have already been discussed:

- Modularity: The architecture should curtail complexity and promote modifiability issues such as reconfiguration and scalability.
- Performance: Software abstractions and structure should not bottleneck performance such communication.
- Human Machine Interface: The user should have access to all levels of the architecture if performance will not be obstructed.

- Openness: The architecture should conform to an open technology such as standardization. Moreover, it is desirable that its implementation is not hardware-dependent.
- Fault-tolerance: The controller should have comprehensive monitoring systems and a fault-tolerant approach to harness problems.

2.6.1 OMAC Architecture API

The Open Modular Architecture Controller (OMAC) was initiated by a consortium of some prominent industrial and research groups (Birla, et al., 2001). The vision for the OMAC Application Programming Interface (API) is to enable control vendors to supply standard components that machine suppliers can easily configure and integrate to build machine control systems. The framework also seeks to leverage easy reconfiguration by end-users. The OMAC API is built on an object-oriented approach to plug-and-play modularization. Software entities are grouped into components, modules and tasks according to their level of granularity; each entity is based on a Finite State Machine (FSM) to facilitate collaborations with related activities. A module in this sense refers to a container for components. Figure 2.6 below shows how the OMAC API specification highlights the relationship between an application control system, modules and components.

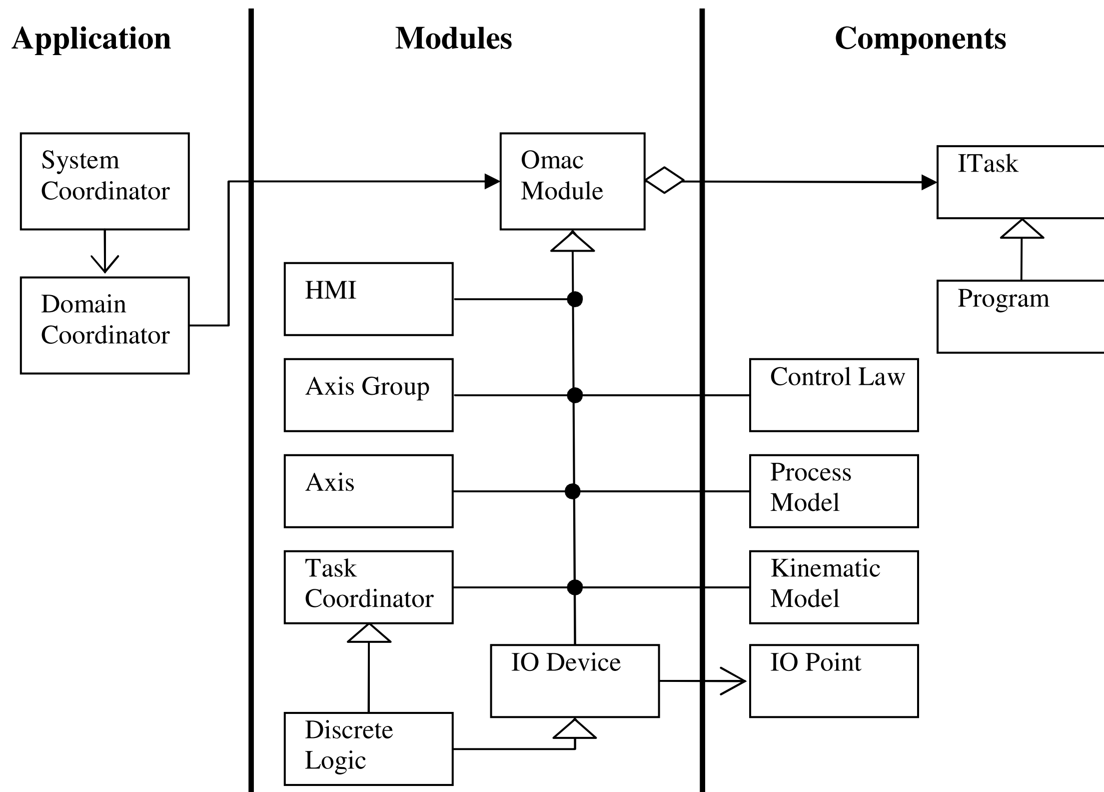


Figure 2.6: OMAC Architecture

The API defines two elementary FSMs: a lifecycle FSM and a task FSM. The lifecycle FSM is used to deploy, publish, connect, initialize and shutdown components during their lifetime. The Task FSM is a smaller state logic for executing programs and individual program steps. A C++ FSM class library handles state nesting and other advanced state machine logic. A prominent feature of the OMAC API is a framework environment for developing and integrating components. OMAC API initially adopted Microsoft Component Object Model (COM) but later evolved to a customized framework called IOmac. The framework defines the modus operandi of component interfaces to enable components to collaborate and advertise functionality and operational status. Thus a typical component encapsulates the following:

- A *Functionality interface* – defines behavior, state and parameter manipulation.

- An *Infrastructure interface* – provides support for advertising services, where it is and how it operates.
- A *Connection interface* – advertises module dependencies.
- *Attributes* define how a component can be customized.

The OMAC approaches reconfigurability principally through the flexibility it provides for exploiting component interfaces. Furthermore, interfaces provide a uniform API for dealing with most software (object) functionality. A connection API allows components to advertise what other components they require and assist in resolving component dependencies. To further enhance system reconfigurability, the OMAC API supports embedding information in a component. Thus a component has the ability to be used in the design phase in a drag-and-drop Integrated Developer Environment (IDE) of a visual programming tool. Embedded information also allows a component to be queried locally on the shop floor about local properties such as its history. Thirdly, embedded information supports component introspection that allows users to determine the capabilities of a component and how to customize it. Another interesting attribute of OMAC towards reconfigurability is through component plugs; i.e., the ability to plug-in or replace components within a module.

The OMAC API is one of the most cutting-edge reference architectures available, though a full implementation is yet to be realized. Obviously, this is due to the stringent details the framework demands.

2.6.2 UBC Open Architecture Control System

The UBC Open Architecture Control System (Oldknow and Yellowley, 2000) is quite an intriguing design. Both hardware and software architectures have been clearly defined with emphasis on openness and reconfigurability. Even though this is categorized under software architecture, we deem it appropriate to discuss the hardware architecture in order to understand the software layout. The hardware system is laid out on a double backplane similar to the Modular CNC System described earlier. Obviously this is a task-specific design. The reference architecture is illustrated in Fig 2.7.

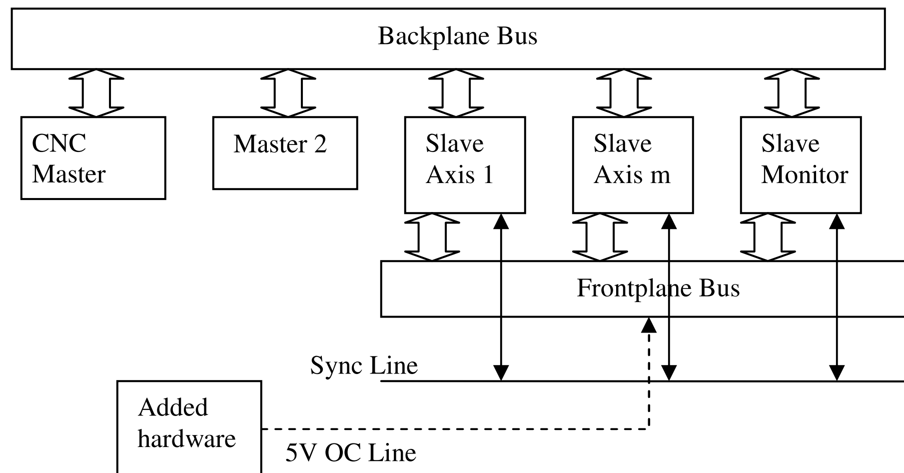


Figure 2.7: The UBC Open Controller Reference Architecture

The reference architecture is based on a distributed model involving several processing units and two communication channels. The primary master – *CNC Master*, is mainly responsible for user interface tasks and first-stage interpolation. *Master 2* is optional and may be used for CAD/CAM tasks. There is one slave controller per machine axis and these receive first-stage interpolation increments from the *CNC Master* over the backplane (such as STD) or a network (e.g. Ethernet). The first-stage increments are then broken down into second-stage increments required by the controller at each servo loop frequency using either a linear or a parabolic blending algorithm. As shown in Fig. 2.7, the servo controllers are connected to each other and other process monitoring boards through a second communication channel consisting of a Frontplane bus and a synchronization line. The channel enables the controllers to implement dynamic interpolation algorithm which facilitates the adaptive control of machining processes in response to multiple constraints (e.g. spindle torque, shank stress, etc) as well as integration of new hardware. The dynamic interpolator works by reading the value present on the Frontplane bus before closing the servo loop. The bus consists of open collector lines which can be written to by all connected processors. In the simplest case of a single stateline, if the controller reads a logic high value, the next second-stage increment is added to the position buffer; otherwise the servo loop is closed in the normal way. This approach enables any servo

controller or process monitor to adaptively control the system in response to violations to a process constraint. A more complex multi-bit stateline approach is also possible. The architecture also addresses flexibility in terms of hardware/software relationships. Static reconfigurability is achieved through hardware independence by using software abstraction; i.e., hardware device behavior is encapsulated in an object-oriented class definition and presented to the rest of the system as a well-defined neutral interface. Moreover, the architecture promotes dynamic reconfigurability, where communication and control methods can be tested, evaluated, modified and replaced without the need to re-start. To accomplish this, an *Open Configuration System Protocol* has been implemented in an object-oriented extension of the Forth programming language. The protocol is based on abstraction of system control hardware into a *Virtual Machine Tool* (VMT). The configuration system thus provides a VMT interface to high-level software as shown in Fig. 2.8.

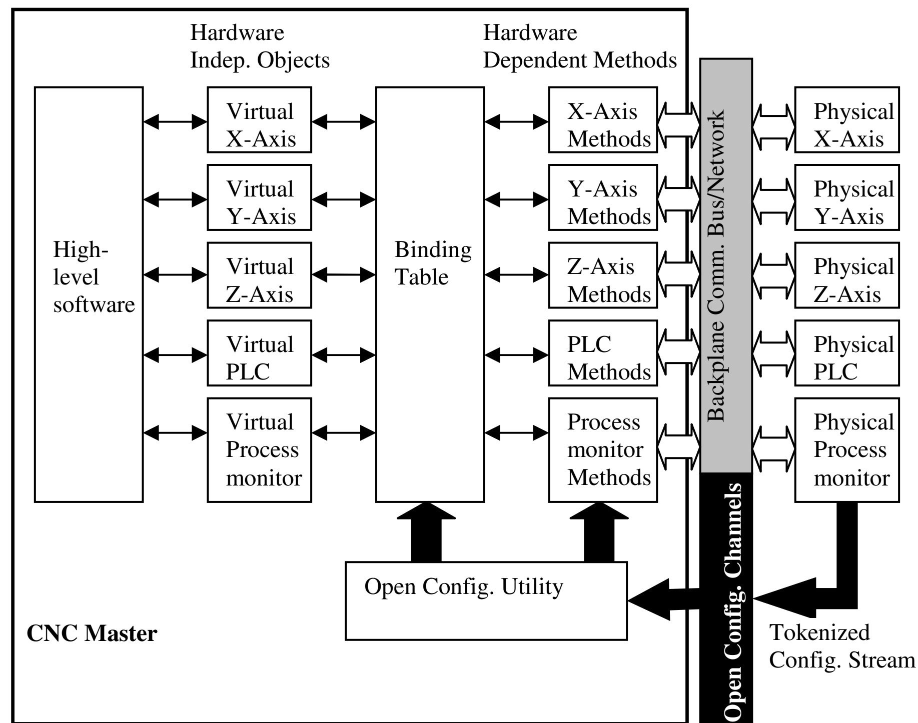


Figure 2.8: Open Configuration System Software Architecture

Calls made to the interface are managed by VMT objects (axes, PLC, process monitors) using a combination of hardware independent and hardware dependent methods. Hardware independent methods are defined within object class definitions while dependent methods consist of references into a reconfigurable software switch called the binding table. The latter translates these references into the corresponding device specific methods used to interface physical hardware. To this end, a technique within the Forth programming environment known as *Vectored Execution* is used. By storing device specific code in the firmware of the devices themselves (in a tokenized form), hardware devices can self-instantiate in the system in plug-and-play style. The start-up procedure is as follows; the open configuration system queries all available communication channels for compatible devices. When a device is discovered, a *Tokenized Configuration Stream* (TCS) is received from the firmware of the device. The tokenization scheme is an extended version of the Open Firmware standard by Sun Microsystems, and can be directly codified into Forth. Afterwards, the Forth code is interpreted by the system and the code registers the device, defines the method required to communicate with the device, and binds these to the appropriate VMT methods through the binding table. Implementation of the reference architecture together with the dynamic interpolator and TCS has been demonstrated with a novel Xilinx FPGA based single-axis servo controller and a simple DC motor. The device closes a velocity controlled servo loop at a frequency of 4 kHz.

This architecture bears the hallmarks of high performance reconfigurable architecture. It is not clear though how the *CNC Master* is synchronized with the DSP-PCI boards. The stateline also presents some complexities, and associated software modules are not encapsulated to give clear hardware-independence. The designers claim that the backplane may be replaced by a network. Obviously, in this case more work would have to be done to arrest network-induced problems such as jitter.

2.6.3 NRC Tripod

A three-tier flexible architecture framework to support computations of parallel kinematic mechanisms (PKM) is presented by Atta-Konadu et al (2005). The framework was originally designed for the NRC-IMTI Tripod project, but can be generalized for other mechanisms as well. The architecture successively computes different algorithmic stages

on reconfigurable computing platforms. In the basic configuration shown in Fig 2.9, the top level algorithm, *Frame-Constants*, is executed only once for a particular mechanical configuration, to derive global constants such as direction and position vectors for kinematic computations. Data from the top layer are passed to a second algorithm, *Iteration-Parameters*, where the orientation matrix and translation vector of the Tripod platform are computed for each tool-center point provided by a path-generator. At the bottom level, the algorithm *Joint-Interpolation* receives inputs from *Iteration-Parameters*, and computes joint position (in local joint coordinates) for individual joint controllers.

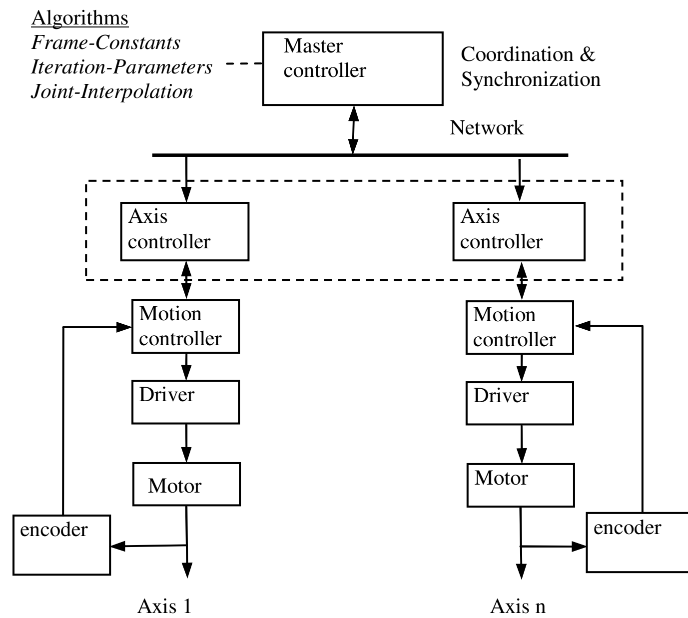


Figure 2.9: Three-Tier Computing Hierarchy for the NRC-IMTI Tripod

This concept enables the architecture to migrate the algorithms to different computing platforms on a network, depending on the decomposability of the algorithms. Figure 2.10 shows two other configurations that can be derived from the framework.

The distributed style that the architecture provides has the potential to greatly simplify and speed up computations. However, communication latencies over the network could severely degrade performance if delays are not properly handled. Moreover, the

clocks on the different nodes have to be continually synchronized for computation consistency.

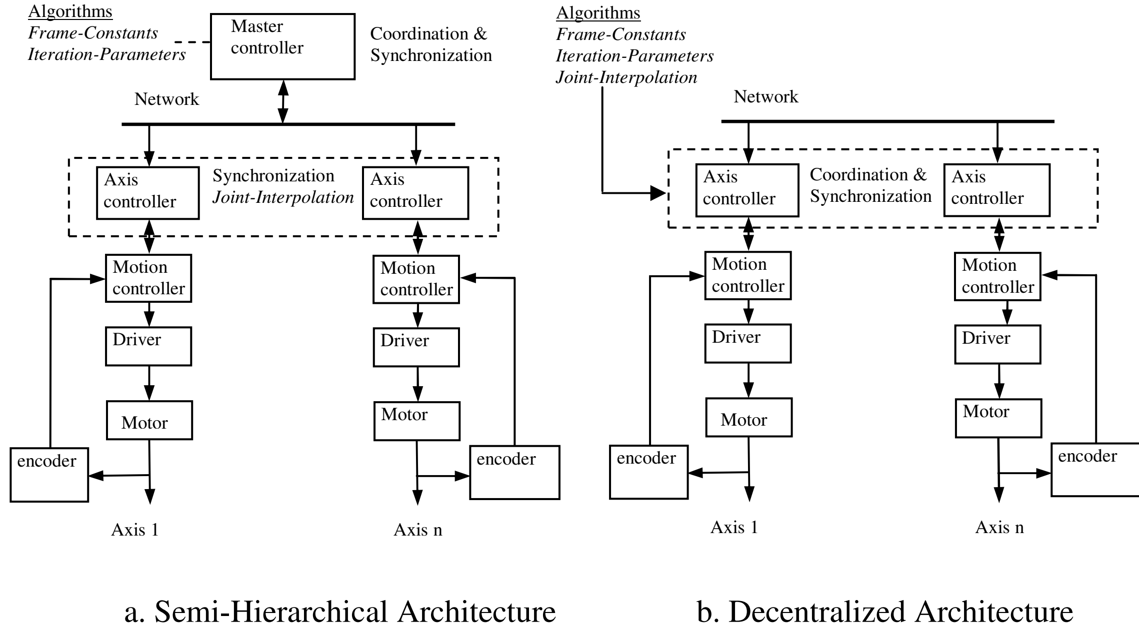


Figure 2.10: Tripod Computing Architecture Variations

2.6.4 CLARAty Architecture

The CLARAty (Coupled Layered Architecture for Robotic Autonomy) architecture is the invention of the Jet Propulsion Lab of the California Institute of Technology (Nesna, et al., 2003). The architecture approaches flexibility by defining reusable software components, and the target platforms are rovers for Mars exploration missions. Thus the issues of locomotion and manipulation with a robotic arm are dealt with. The architectural framework defines a 2-stage hierarchy; the Functional Layer at the bottom and the Decision Layer on top.

The Functional Layer includes a number of generic frameworks centered on different robotic applications. Packages included in this layer are: I/O, motion control and coordination, locomotion, manipulation, vision, navigation, mapping, terrain evaluation, path planning, science analysis, estimation, simulation, and system behavior. The system's low and mid-level autonomy are implemented by this layer. Most control logic including

vision-based navigation, sensor-based manipulation, and vision target tracking that use a predefined sequence of operations are implemented in the Functional Layer. The layer has four main features aimed at creating component reusability. First, it provides system level decomposition through object-oriented techniques with different levels of abstractions. For example, a general locomotor may be extended to specifics of wheels or legs, etc. Second, the layer separates algorithmic capabilities from system responsibilities. As an example, algorithms such as inverse kinematics are expressed in their general terms. On platforms (rovers) where an optimized algorithm is available, the general algorithm is overwritten. Thirdly, the Functional Layer partitions behavioral definitions and interactions from the implementation. This allows a motor, for instance, to separate the specialization to a particular hardware controller from the functional details of a controlled motor to a joint. Finally, the layer provides flexible runtime models; this enables a system with specific hardware execution (such as servo control) not to run that of the main processor.

The Decision Layer is the global engine that analyzes system resources and constraints. It encapsulates general planners, executives, schedulers, activity databases, and rover and planner specific heuristics. The Decision Layer communicates with the Functional Layer using a client-server model. Interactions with the Functional Layer include queries about system resources, and sending commands. The layer can also utilize encapsulated Functional Layer capabilities with high-level commands, or access low-level resources and combine them in ways not supported by the Functional Layer. The former is employed when planning capabilities are limited, or when under-constrained system operation is allowed. The latter is useful if detailed, globally optimized planning is possible, or if resource margins are limited.

The architecture has been adapted to different rovers with various motion control and communication architectures, and physical (locomotion) capabilities. Figure 2.11 shows the generic controlled motor and joint classes and their adaptations to Fido, R8 and R7 NASA rovers.

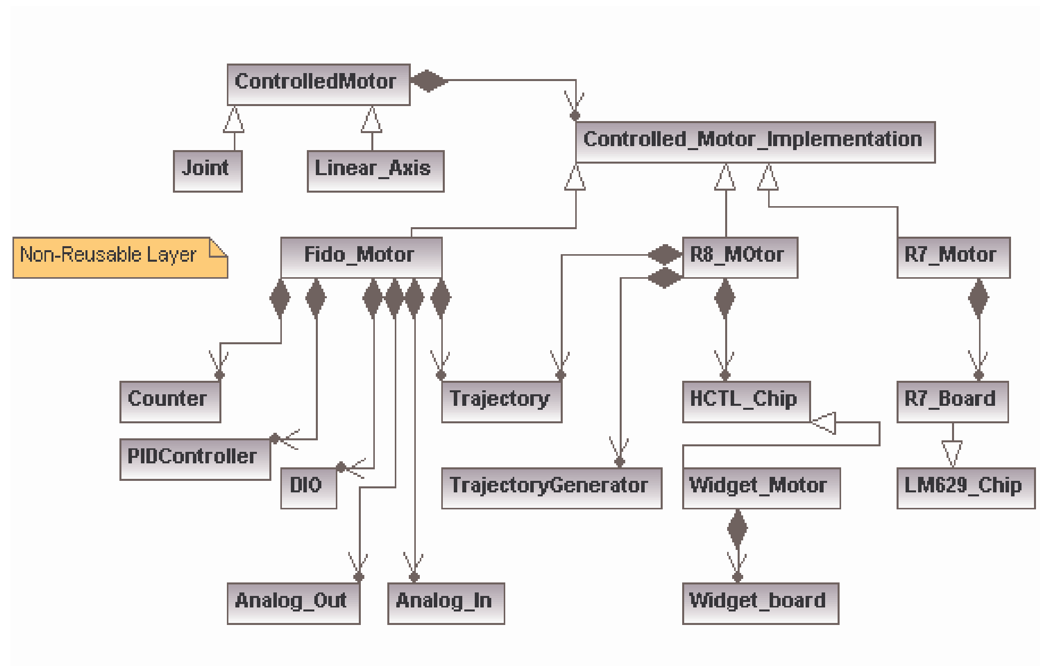


Figure 2.11: CLARAty Implementation on Various Rover Platforms

2.7 Control Architectures

A control architecture refers to the actual operational software used to run a machine (e.g. robot), and may also include intelligence to handle interactions with the environment or system, and optimization procedures necessary to enhance performance. This section reviews some well-known control strategies and some new paradigms in system-level control reconfiguration.

2.7.1 Classic Control

Robot control for contour following operations can be classified into gross motion and fine motion control (Somló, et al. 1997). In gross motion control, an end-effector or machine tool follows a prescribed path as closely and as quickly as possible; the task is to find a control law that governs its velocity and position. On the contrary, in fine motion control, the objective is to control position and force simultaneously by a technique called hybrid control. Gross motion control may be implemented in joint or Cartesian space depending on whether the desired path is specified in the joints or Cartesian space. Fine motion control may be passive or active compliance control. The former may be achieved through Remote Compliance Center devices to compensate for disturbances. In active

compliance control, there is force (torque) feedback to correct errors. The classification of robot control is given in the Fig 2.12 below.

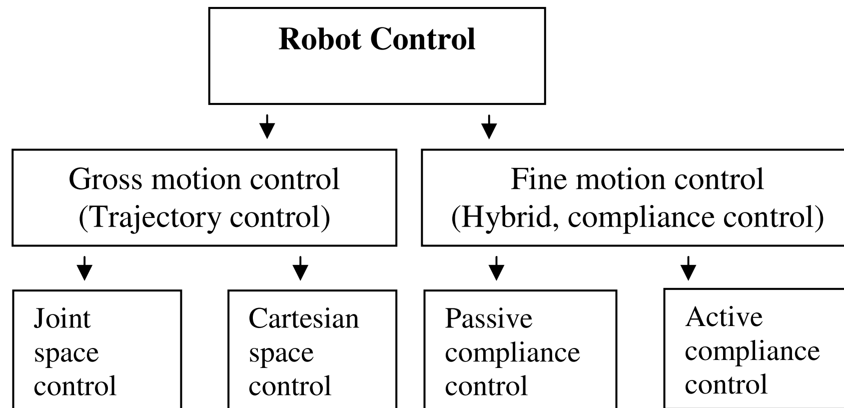


Figure 2.12: Classification of robot control problems

For a given configuration of the robot, one can create a mathematical model of the robot and with previous knowledge of the variation range of the parameters a conventional robust controller can be designed. The complete dynamic model of a robot consists of the dynamic models of the mechanical and actuator parts. In most cases of robot control, the dynamics of the actuator part are ignored as it is assumed that the torque produced by the motor is suitable to realize the goals at all times. The design task is then to find the appropriate input torque, which enables the robot to closely follow a desired trajectory. When using high torque or current controlled dc motors, this condition is normally met; hence the torque is simply proportional to current. By using fast feedback control a suitable torque can be realized. With more powerful drives, responses can be made very fast but implementation cost can be very high. In every case optimal solution should be applied to realize the appropriate power/torque to be applied. Therefore, in many practical cases ignoring the actuator dynamics may have significant effects on system performance. When voltage-controlled dc motors or electro-hydraulic actuators are employed, the torque is produced with delay depending on the time-constants. The time constants for small permanent-magnet dc motors are normally below 1 ms: when applied to robot systems, the time constants are small enough for actuator dynamics to be ignored in the

controller design (Somló, et al. 1997). However, for bigger motors it is necessary to consider the complete dynamic model incorporating mechanical and actuator dynamics. Most industrial robots are controlled by PID type controllers and work properly under conventional conditions (low velocity, constrained payloads etc). Usually, this is implemented as independent joint control if the interconnections of the joints are not closely tied. In such situations, joint interconnections are neglected in the model. Therefore two potentials problems could arise in robot control design (Somló, et al. 1997): There is the problem of uncertainties of parameters and the non-linear interconnections between the manipulator joints. To curtail the first problem, the controller must be robust; to compensate for the unwanted effects of the second problem, the interconnections must be taken into consideration. The interconnections between the joints appear in the total torque (or force) acting on the joints. The significant part of this torque can be measured by an appropriate sensor attached to the axis. An alternative approach is to compute the interconnections in real-time by using the kinetic equations of the robot with measured positions, velocities and accelerations of the joints. These real-time computations may be performed by an ASIC (Application Specific Integrated Circuit) or by a central computer.

Generally, classic control architectures may be decentralized or centralized (Somló, et al. 1997). In the first method, the control loops are independent of each other as in the case of PID control. In the second method, signals from other joints are used in the control of some of the joints. The latter offers high speed and quality processes and is realized on centralized control systems. A classic example of centralized control is the Computed Torque Control (CTC) technique. Regarding solution methods to control problems, there are two approaches: nonadaptive and adaptive methods. Examples of nonadaptive methods are CTC, Resolved Motion Rate Control (RMRC), Resolved Acceleration Control (RAC), Time Optimal Control, and Variable Structure Control (VSC). Adaptive control methods include models such as Model Reference Adaptive Control (MRAC), and Self-Tuning. The MRAC strategy is presented by Somló, et al. (1997). The motion of the robot is controlled in such a way that it closely follows the motion of a given model, which represents the desired performance of the system. The basic scheme of MRAC is shown in Fig. 2.13.

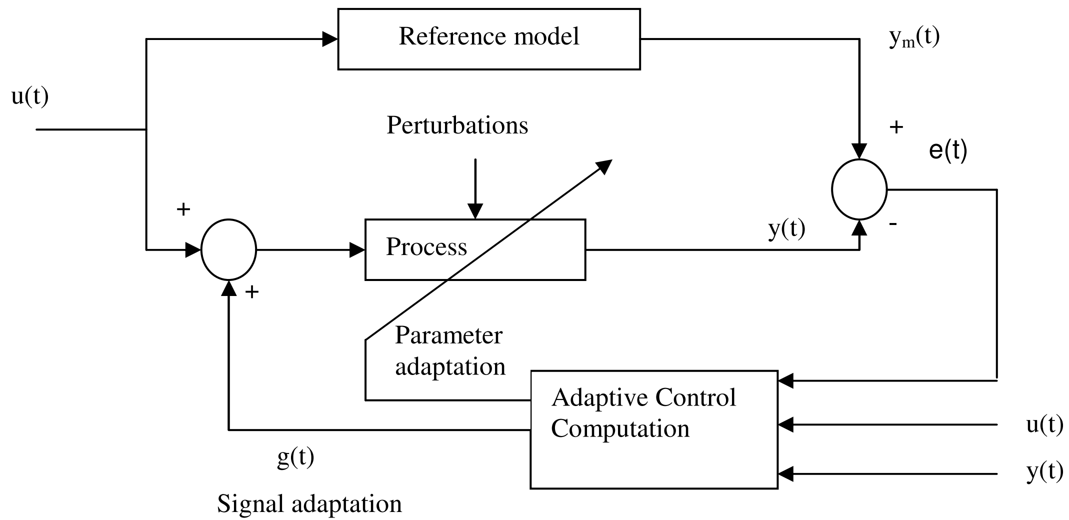


Figure 2.13: Model Reference Adaptive Control (MRAC)

The input signal vector **u** affects both the actuators of the joints and the input signal of the models of each joint. The output of the joints are compared and the differences of these are used to drive the system to track the model. The ideal case is when the adaptation error, which is the difference between the model output and the joint output, approaches zero. Two approaches used to realize this method are parameter adaptation and signal adaptation. Both of these strategies are illustrated in the Fig. 2.13 above. Based on the adaptation error, the adaptive control loop produces signals which in the case of parameter adaptation correct some parameters of the system, and in the case of signal adaptation send signals to modify the system signals.

2.7.2 Reconfigurable Control Architectures

Generally, there are two issues regarding reconfiguration: Distributed system reconfiguration and control reconfiguration. The former is related to a structural or system-level software reconfiguration while the latter deals with modifying control law in its structure to maintain a certain performance. Reconfigurable control for fault tolerance is an exceptionally challenging control design problem. Failure detection and identification, parameter estimation and controller redesign have to be carried out on-line and completed within tight time boundaries. Some methods that address reconfigurable

control include linear-quadratic (LQ) control methodology, adaptive control systems, knowledge-based systems, and Eigen-structure assignments. Considering adaptive control, there are several approaches but in general perspective control reconfiguration is attempted by using a continually adapting nonlinear model. At this point we make a distinction between fault-tolerance, robust control, and reconfiguration control strategies. In the first situation, when a fault appears in one peripheral element and the plant is still observable and controllable, the controller uses a fault accommodation strategy to achieve its original objectives by adapting control parameters to fault conditions. A robust controller (also called adaptive controller in some literature) aims at providing suitable system performance if the parameters and conditions vary within given domains. Parameters and conditions include uncertainties in the mathematical model of the plant, and strong non-linear interconnections (e.g., between the joints of a robot); for distributed systems, robust controllers accommodate network-induced jitters and sometimes computation delays.

In system reconfiguration, faulty peripheral elements are switched off, and the control structure and associated control objectives are modified to accommodate the absence of certain parts of the plant. A similar strategy may be developed for the presence of new elements. The initial model or reference model is based on initial information regarding a priori fault and fault-free scenarios (Benítez-Pérez, et al., 2005). To update current models, approaches such as neural networks and fuzzy logic may be used, where several approximations can be followed such as differential neural networks, and radial basis functions. Figure 2.14 shows the general strategy for fault diagnosis and control configuration of this method.

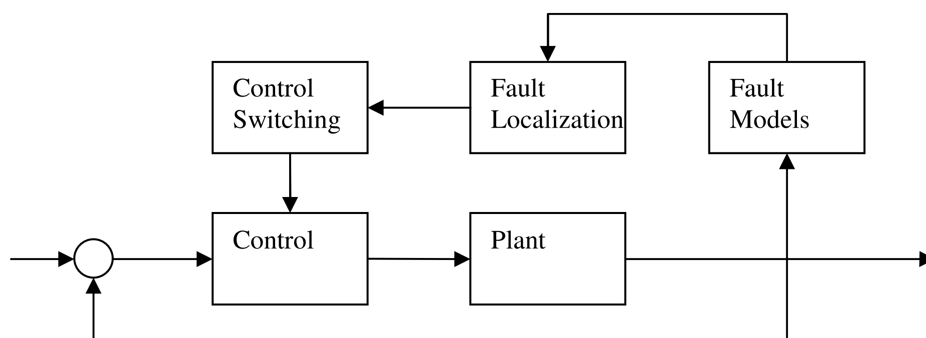


Figure 2.14: Control Configuration – General Strategy

Another reconfiguration strategy is implemented in a hierarchical manner, where a decision maker is used to switch from one control to another. The decision maker layer may be implemented by a knowledge-based algorithm such as neural networks. The 3-tier model presented in (Wills et al., 2001) for autonomous control is a typical example (Fig. 2.15). The complex structure combines high-level situation awareness and mode selection functionalities with mid-level coordination routines for leveraging mode transitioning and reconfigurable control as well as low-level control activities. A sensor management unit provides appropriate data to a high-level situation awareness module, a fault-detection-and-isolation module, and to all lower levels. When an external or internal situation (such as faults) is detected, the mode selection module generates in real-time sequences of new modes (such as waypoints) for the controller. The mode switching or reconfiguration module then schedules transitioning dynamics by using a nonlinear dynamic/fuzzy logic. The low-level provides set-points and command trajectories to low-level controllers.

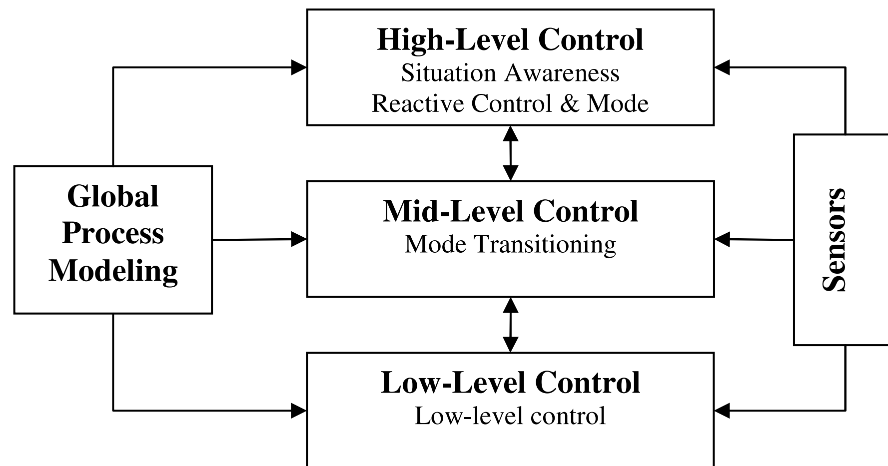


Figure 2.15: Hierarchical Control Reconfiguration Structure

Wang and Shin (2001) approach hierarchical reconfiguration from a different perspective, where reconfigurable control software is viewed as consisting of "*a set of inter-communicating components, each of which is a pre-implemented software module and used as a building block*". Components are modeled with a set of external interfaces, communication ports, a control logic driver based on a Nested Finite State Machine

(NFSM), and service protocols. The objective is to achieve execution-code-level reconfigurability for control software.

Another approach to reconfiguration deals with adopting different plant models by using reconfigurable execution models of the kinematics and dynamics of systems. This strategy is invariably associated with developing generic algorithms and/or execution strategies. Lin and Lee (1991) for example developed a strategy for parallel execution. They exploited possible inherent parallelisms in robotic algorithms and investigated their characteristics such as granularities, data dependencies, etc. Based on these, a medium-grained reconfigurable dual-network SIMD (Single Instruction-Stream-Multiple-Data-Stream) machine consisting of multiple processing units was developed.

Most of reconfigurable control algorithms are very recursive requiring intensive computation and are not able to recover the original performance level. For example, an LQ-based approach may not restore the original system performance due to ambiguities in the optimization procedure. The pseudo-inverse method (PIM) and linear model following (LMF) control are simpler to implement than the previous but the performance of the closed-loop system cannot be easily predicted. The approach presented by Dhayagude and Gao (1996) attempts to address some of these issues but even this is based on a priori assumed conditions.

2.8 Concluding Remarks

In this chapter, we examined several issues related to architectural design properties and software architecture styles. This led to a comprehensive review of control system architectures. Prior to this, control system architectures were classified into three groups according to what they emphasize. The categories were Hardware Architecture, Software Architecture (system-level), and Control architecture. Different architectures were reviewed and critiqued based on their throughput, communication efficiency, functional coherence, dependability, modularity, relevance to design objectives, openness, and fault-tolerance. The lessons learned which will be applied in our design are as follows:

- A distributed hardware design can greatly reduce complexities and enhance performance, scalability, modifiability, and reconfigurability if the underlying communication structure is reliable. A modular homogenous design is desirable for clean interface definitions.

- A *rigid* communication hardware such as a backplane guarantees high real-time throughput, but a network-based system introduces greater flexibility by providing loosely-coupled interconnections among elements or nodes. However, performance may degrade if network-induced jitter and bandwidth limitations are not properly deduced at the design phase. Hence, use tight-coupling for hard real-time and/or high rate communication such as servo loops and loose-coupling otherwise.
- An object-oriented software architectural style leverages component-based design for enhanced modularity, which leads to easy configuration, platform-independence, etc.
- Hardware abstractions should be used to limit or eliminate hardware dependencies.

The next chapter presents the conceptual framework of our architecture which we call IMC (Intelligent Modular Controller) architecture. It is based on a layered reference model with modularity, simplicity and flexibility as the design cornerstones.

3. SYSTEM ARCHITECTURE

3.1 Architecture Design Philosophy

The main objective of this research is an experimental design of a control architecture that is based on simple and cost-effective Commercial-Off-the-Shelf (COTS) components to create a generic or domain-level control system that is easy to configure and customize for a wide-variety of applications. It was realized from the previous chapter that certain key technologies enable such design solutions. In retrospect, a distributed system based on modularity and network communication has inherent or potential flexibility due to loose-coupling between entities. The main bottleneck though is the nature of its communication network properties such as bandwidth and transaction mechanisms. Also, such design-for-flexibility should necessarily be accompanied by well-conceived software architecture. Object-oriented styles lend powerful credence and support for flexible designs. This chapter presents the conceptual framework of the Intelligent Modular Controller (IMC) architecture. Below is a summary of the design concepts;

- Distributed architecture based on a reference (abstract) architecture that defines the various hierarchical decomposition.
- Loosely-coupled elements (software and hardware) for easy system development and flexibility. This may be achieved by object-oriented software architectural style and networked (Ethernet) communication elements.
- Controller elements should possess the ability to automatically subscribe for services they need and also publish their own services. This makes it possible to support automatic configuration of high level applications.
- Modular design.
- The execution flow may be configured to be biased in different ways; for example, a centralized or decentralized interpolator may be employed based on communication bandwidth. The proposed means to achieve this is as follows:

- An Ethernet network for high bandwidth communication and robustness.
- Synchronization in order for nodes to have a global sense of time.
- A synchronization mechanism for real-time entities.
- A protocol for devices to automatically discover themselves and publish their services to enable auto-configuration of architecture.

The system architecture strongly emphasizes modularity both in software and hardware. The hardware architecture is shown in Fig. 3.1. Instead of having a monolithic controller for each machine axis, control functionalities are distributed on a microcontroller host and a dedicated motion controller for each machine axis.

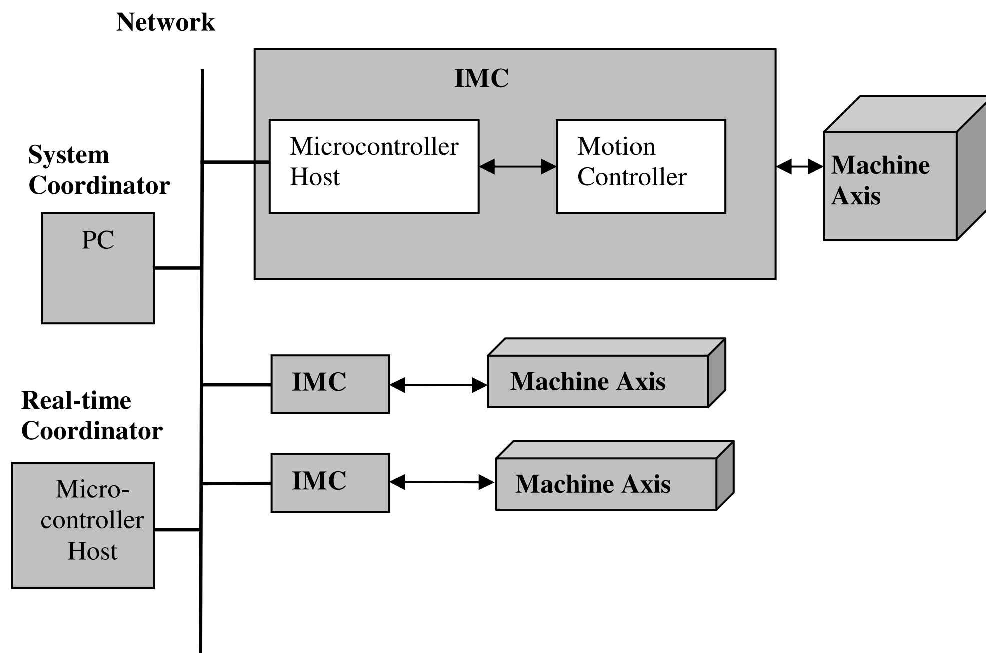


Figure 3.1: The IMC Hardware Architecture

Figure 3.2 shows the reference model architecture layers superimposed on the hardware entities (shaded regions). The reference model is inspired by work done by Sorensen in 2003. The model is made up of seven layers for emphasis. However, in the course of operation some layers may be redundant. The following is a discussion on the functionalities of the model.

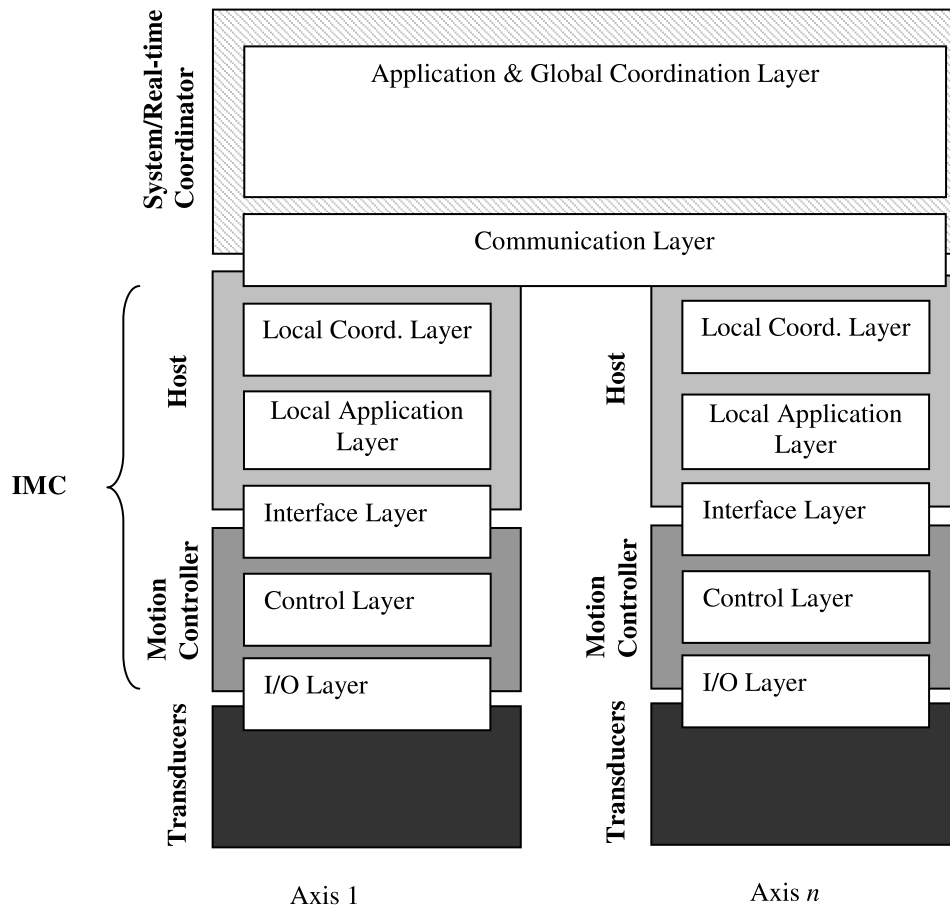


Figure 3.2: The IMC Reference Architecture

I/O Layer: This layer provides reading/writing of I/O ports connected to transducers, i.e., sensors and actuator drives. The layer contains different types of I/O units for digital and analog I/O, DAC (Digital to Analog Converter), ADC (Analog to Digital Converter), and switches. This layer also hosts the motor driver or servo amplifier, and power supply and conditioner needed to provide appropriate signals to/from the transducer layer. The I/O layer delivers energy to the actuator based on signals from its control layer.

Control Layer: This layer is responsible for translating motion set-points into input signals for the actuator that drives the mechanical element. It is imperative for this layer to implement a reliable synchronous sub-component for hard real-time performance. Servo-

control and sensor feedback conversion needed in the control loop are part of this layer. Fine trajectory generation for smooth motion may also be performed in this layer.

Interface Layer: The control layer communicates with the higher level command layer through a high-speed real-time network. This layer is responsible for ensuring the timely delivery of motion commands from the command layer to the control layer. It also delivers feedback and status data to the command layer. In essence, the interface layer serves as a hardware abstraction layer (HAL) in order to separate hardware-specific software components from generic ones. This is to allow easy hardware replacements and software reuse.

Local Application Layer: Some high-level applications can be hosted directly on the local microcontroller. These include kinematics and interpolation algorithms which can be decoupled for each axis. Local execution of such algorithms helps to reduce the volume and frequency of communication across the network. The application layer also includes an embedded web server, which is invoked by the higher layer to provide user interface services.

Local Coordination layer: This layer coordinates activities in the lower layers according to global (system-level) instructions received. For example, different motion control modes may be specified by the users such as coordinated motion or synchronized motion. If coordinated motion is requested, the layer activates a method to periodically synchronize the controller's timer with the global time. The synchronization methodology is discussed in Chapter 7. The layer also hosts a graphical user interface (GUI), and databases including a real-time database of temporal data. The user interface provides status information such as position logs, command functions and configuration editors. Lastly, the layer implements components to monitor the status of lower layers such as motor stall situations.

Communication Layer: This layer is the interface to the network on the distributed system. The layer is responsible for creating a virtual global environment so that the whole system appears as one entity to the IMC nodes, even though they are spatially separated. To enable this, the layer implements a configuration scheme to automatically discover services on the network, and register itself to other nodes on the network. Chapter 6 provides the communication architecture design details and analysis.

Application and Global Coordination Layer: All global tasks and functions are implemented in this layer. The layer provides high level interpolation, motion synchronization and coordination, and system configuration. The layer also supplies a graphical user interface for monitoring devices, creating trajectory data (e.g., NC code), and sending commands to the IMC nodes. High-level control algorithms may be implemented in this layer.

3.2 Intelligent Modular Controller (IMC)

Each axis controller (IMC) is designed such that there is minimal need for global data interchange. Therefore most hard real-time functions are confined within the IMC domain. Each IMC communicates with the central coordinator through a network while communication with the mechanical axis is through an embedded motion controller and a set of I/O signals. The IMC architecture allows a completely distributed or a hierarchical architectural structure in order to accommodate different demands imposed by higher application layers. A typical example is the control of a serial robot in contrast with a parallel kinematic machine (PKM): While it is practical to distribute the inverse kinematics of an n-DOF PKM to n computing elements (i.e., n IMCs), a centralized computing structure is required for the latter. Another feature of the IMCs is the ability for any one of them or an additional IMC to serve as a real-time coordinator. This is especially important in two different applications: The first is the situation where the global workstation or computer which hosts most of the higher layer application software runs on a non real-time operating system (OS). While many functions such as path-planning and NC program parsing may be abstracted from non real-time computing, real-time transactions are a necessity for coordinating coarse or finely interpolated data. The architecture allows an IMC to be selected for this purpose. In this case, the IMC is relieved of its motion control activities in order to conserve computing resources for real-time coordination. The architecture, therefore, does not predispose the user to any particular operating system which is a key advantage. The second interesting feature of the IMC architecture is that they are designed from the onset to be embedded mobile computing elements. This provides the ability for them to be integrated or embedded in the mechanical platform. The concept can therefore be used to control mobile equipment such as AGVs or mobile robots. There are a few architectural permutations that can address

such situations: One arrangement is to have one IMC inside each host mechanical axis; another variation is to map several motion controllers to one IMC microcontroller. The IMC hardware architecture design is discussed in Chapters 4 and 5.

3.2.1 Interfaces

Since the IMC architecture is meant to be a generic controller for a variety of host modules, the interface is *open* for extreme variations in the connection requirements of mechanical host modules. It is assumed that each host module will be equipped with a suitable motor drive and signal conditioner. The architecture provides the necessary I/O connections for interfacing signals to the motion controller and higher layers. The interface with other nodes on the network such as the global coordinator(s) is a network API, and the network hardware. The network provides a reliable and real-time communication medium for synchronous messages such as interpolation data streams as well as control messages such as start/stop which are asynchronous.

3.3 The System Coordinator(s)

The global coordinator(s) is/are responsible for all tasks which cannot be suitably decomposed to the IMC nodes; for example, certain kinematics, path planning, NC programs, interpolation, and coordination of axes motion. As aforementioned, if the global coordinator is incapable of coordinating real-time tasks, an IMC host is assigned to be a real-time coordinator. The global coordinator still plays the role of system monitoring at its backend, and user-induced front-end activities.

3.4 Conclusion

In this chapter, an overview of the architecture that supports the Intelligent Machine Controller (IMC) has been discussed. The architecture is distributed over three kinds of platforms; the IMC nodes, a system coordinator and a real-time coordinator. The IMC node consists of a microcontroller and an embedded motion controller board. All global services are implemented on the coordinators. Real-time high-level services such as interpolation are provided by the real-time coordinator if the system coordinator cannot provide such services. The real-time coordinator runs on a platform similar to the IMC microcontroller. The next chapter presents a review of Java technology and the

microcontroller hardware that is used for the IMC. The software structure for the entire system is presented in Chapter 9.

4. EMBEDDED COMPUTING PLATFORM

4.1 Introduction

This chapter elaborates how a suitable microcontroller was selected for the IMC architecture design. The discourse includes the rationale for Java in real-time system design, and enabling technologies for embedded Java systems. Essentially, an embedded system is an application specific computer system that is part of a larger system or mechanical component. It is designed to perform a limited range of functions with no, or minimal user intervention. These systems operate on significantly low power and consequently use slow processors and small memory sizes in order to minimize costs and energy consumption. Like typical real-time systems, real-time embedded systems also require a real-time operating system (OS) for process management and synchronization, memory management, interprocess communication, and I/O. As was mentioned in the previous chapter, the software signature of the architecture is a homogenous programming environment based on Java. The opportunities and implications of this preference are discussed in ensuing sections, and against this backdrop, a Java-based hardware is selected. The closing sections describe motion controller design options, and the controller ASIC (application specific integrated circuit) that was selected for the design of the IMC motion controller board.

4.2 Java for Real-time System Design

The most widely used embedded real-time systems are written in C++ and Ada83. Currently, these are also the most popular object-oriented tools in this field. Both are robust real-time programming software tools but not necessarily the ultimate tools. C++ is widely used because it is readily available and supports low-level programming. However, it suffers from low robustness to modification, and poor readability and maintainability. On the other hand, Ada83 is too large for many embedded systems, and too expensive. Java has many unique qualities that concur very well with modern-day paradigms in industrial automation such as software reuse, openness, single inheritance, software

modularity, and platform-independence (Vyatkin, et al., 2005). However, by design it is more suited for enterprise-level tools, where real-time requirements are not stringent. Nonetheless in recent years, Java developers and interest groups have been carving out specifications for Real-Time Java, in order to add *color* to the real-time world. Java is a unique blend of language definition, very robust, and offers a rich class library and a runtime environment. Programs are compiled to bytecodes that are executed by a Java virtual machine (JVM). Its robustness is derived from strong typing, runtime checks and avoidance of pointers. Intermediate bytecode representation simplifies porting of Java to different operating environments and is easy to implement requiring minimum system resource. As shown in Fig. 4.1, Java has four important components: the Java Programming Language, the Java Class Library containing binary representation or bytecode, the Java Native Interface to support functions written in C/C++ and the JVM.

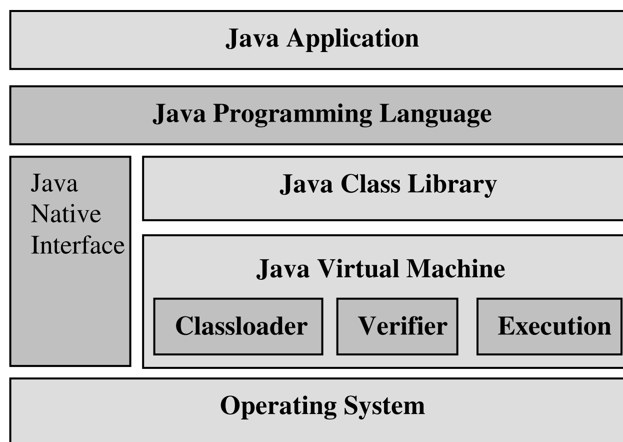


Figure 4.1: The Java System Architecture

The JVM loads, verifies and executes the bytecode of a Java program. Execution speed is hindered by interpreting bytecodes, and this has been one of the setbacks of Java in real-time applications until recently. One solution to this problem is a JVM with a just-in-time (JIT) compiler designed for desktop and server systems. However these require large memory footprints and have to be ported for different processor architectures. An excellent candidate for real-time embedded system designs is a Java processor (hardware)

that implements the JVM as a native machine. This avoids the slow execution model of an interpreting JVM and the memory requirements of a compiler, thus making it suitable for embedded systems. There are two approaches to Java bytecode execution by hardware. In the first approach, a Java coprocessor is placed in the instruction fetch path of a general purpose microprocessor and translates Java bytecodes to sequences of instructions for the host CPU or directly executes basic Java bytecodes. In the second approach, a Java chip replaces the general purpose CPU. All applications therefore have to be written in Java. Table 4.1 is a cross-section of some existing Java chips (Schoberl, 2005).

Table 4.1: Java Hardware Comparison

Product	**Type**	**Chip Technology**	**Speed (MHz)**	**Java Standard**
JIFFY	Translation	FPGA		
Jazelle	Co-processor	ASIC 0.18 μs	200	
JSTAR	Co-processor	ASIC 0.18 μs	104	J2ME CLDC
picoJava	Processor			Full
aJile	Processor	ASIC 0.25 μs	103	J2ME CLDC
Cjip	Processor	ASIC 0.35 μs	67	J2ME CLDC
Komodo	Processor	2600 LCs	20	subset

4.3 Opportunities and Constraints for Java Embedded Devices

Generally, program execution on an embedded platform is done either cyclically or concurrently with time constraints. Java utilizes the latter method, and its concurrency is derived from its *Thread* class. The *Thread* class is built on a shared memory communication model where all thread implementations use the same memory heap. Thread activation is governed by mutual exclusion (*synchronized* keyword) and the methods *wait*(), *notify*() and *notifyAll*(). The classes *java.util.TimerTask* and *java.util.Timer* can be used to schedule tasks for future execution in a background thread. Java defines a very loose behavior of threads and scheduling in order to avoid deadlocks and thread starvation. For instance, Java allows even low priority threads to preempt high priority threads. This is important for non real-time applications but a liability in real-time programming. Another limp in Java's response to real-timeliness is in its garbage collection policy. In Java, removal of unreferenced entities (objects) is done automatically, greatly simplifying programming and eliminating the infamous memory leak and dangling

pointer crises in programs written in C/C++. However, even real-time garbage collectors are usually avoided in hard real-time systems. Implementations of Java must include the full Java API library (JDK) constituting over 15MB, which is too large for many embedded systems. Lastly, since Java was designed to be a safe language with a safe execution environment, no classes are available for low-level access to hardware features. Hence the standard library was not defined and coded to support real-time applications.

The Real-Time Specification for Java (RTSJ) defines a new API with support from the JVM and the following guiding principles (International, 2000; International, 2001):

- No restriction on the Java runtime environment.
- Backward compatibility for non-real-time Java programs. This implies that the RTSJ is intended to run on top of J2SE (and not on J2ME).
- No syntactic extension to the Java language or new keywords.
- Predictable execution.
- Address current real-time system practice.
- Allow future implementations to add advanced features.

The RTSJ specification defines threads and scheduling for real-time behavior**.** The base scheduler is defined as a priority-based, preemptive scheduler with at least 28 real-time priorities. In addition, the 10 priority levels for the traditional Java threads need to be available. Threads with equal priority are queued in First-In-First-Out (FIFO) order. Additional schedulers such as Earliest Deadline First (EDF) can be dynamically loaded. The class scheduler and related classes provide optional support for feasibility analysis. Threads are either periodic or bound to asynchronous events. Since garbage collection is a bottleneck in real-time applications, the RTSJ defines new memory areas. *Scoped memory* is a memory region with bounded lifetime. On exit of the last thread from a scope, all *finalizers* of allocated objects are invoked and the memory area is released. *Physical memory* is used to regulate allocation in memories with different access time. *Raw memory* enables byte-level access to physical memory or memory-mapped I/O. *Immortal memory* is a memory regime shared between all threads without a garbage collector. *Heap memory* is the classic garbage collected memory regime. A bound can be set on maximum memory usage and the maximum allocation rate per thread. RTSJ restricts the implementation of the keyword *synchronized* to prevent priority inversion. The priority

inheritance protocol is the default and the priority ceiling emulation protocol can be used. Threads queuing to enter a synchronized block are priority ordered and FIFO ordered within each priority. Wait-free queues provide communication between instances of *java.lang.Thread* and *RealtimeThread*. Classes to represent relative and absolute times with nanosecond resolution are defined by RTSJ. Multiple clocks can represent different sources of time and resolution in order to allow for the reduction of queue management overheads for tasks with different tolerance for jitter. A new type, rationale time, can be used to define periods with a requested resolution over a longer period. Timer classes can produce time-triggered events (one-shot and periodic). External world events may be scheduled and dispatched by the scheduler. An *AsyncEvent* object represents an external event such as a hardware interrupt or an internal event. Event handlers are linked to these events and can be bound to a regular real-time thread. The RTSJ is a complex specification resulting in a big memory footprint beyond the capabilities of many embedded systems. Therefore many implementations use only a subset of RTSJ (Schöberl, 2005).

4.3.1 The Java Micro Edition (J2ME)

To provide a compact API for embedded systems, Sun has defined the Java 2 Platform Micro Edition (J2ME) which is a subset of standard Java API (J2SE). Instead of being a single unified entity, J2ME is a collection of specifications that define a set of platforms for specific products types. The subset of the full Java programming environment for a particular device is defined by one or more profiles which project the basic capabilities of a device configuration (Topley, 2002). The configuration and the profile(s) targeted for a particular device depend on both the nature of its hardware and the target market. J2ME defines three layers of software built upon the host operating system of the device:

Java Virtual Machine: J2ME reduces the function of the JVM to make implementation lighter on smaller processors. This layer is the usual JVM as in every Java implementation. It is assumed that the JVM will be implemented on top of a host operating system.

Configurations: The configuration defines the minimum set of JVM features and Java class libraries available on a particular class of devices. They specify such things as

the types and amount of memory available, the processor type and speed, and the network connections available to the device. J2ME currently defines two configurations, the Connected Limited Device Configuration (CLDC) and the Connected Device Configuration (CDC). The CLDC configuration is designed for *low-end* devices with a memory budget of 128 KB and a 16 or 32-bit processor. The main target devices are wireless devices. The CLDC is based on a small-footprint JVM called K Virtual Machine (KVM) and core class libraries. Many features such as floating point support and finalization have been removed from the Java API. The JVM handles errors by halting in an implementation-specific manner. The following features have been excluded from the JVM (version 1.1): Java Native Interface (JNI), Reflection, Finalization, weak references, user-defined class loaders, thread groups, daemon threads and asynchronous exceptions. The CLDC defines a subset of the Java class libraries *java.io, java.lang, java.lang.ref and java.util*. An additional library *javax microedition.io* defines a simpler interface for network connections. On the other hand the CDC configuration targets devices with at least 2 MB of memory and more capable processors, and can support a much more complete software environment. CDC places no restrictions on the JVM.

Profile: The profile complements a configuration by defining the minimum set of APIs for a particular class of devices. Both J2ME configurations have one or more associated profiles. *Mobile Information Device Profile* (*MIDP*) and *PDA Profile* (*PDAP*) add networking, user interface components and local storage to their CLDCs. The *Foundation Profile* is the building block for all profiles based on CDC. *The RMI profile* for example includes the J2SE Remote Method Invocation (RMI) libraries to the Foundation Profile. Two profiles are most appealing to embedded applications, namely, *EmbeddedJava* and *PersonalJava.* Target devices typically have 32-bit processors and 512 KB of ROM/RAM and 2 MB ROM/RAM respectively. These profiles allow the implementer to remove any package or class, or even a method within a class that is not required, in order to fit the final product into the memory available.

4.4 Microcontroller Hardware Selection

The criteria for the selection of an appropriate computing platform were as follows.

1. The hardware must be capable of supporting at least 60 MHz of computing.

2. The underlying operating system must be capable of multi-tasking synchronously and/or asynchronously.
3. There should be provision for a real-time serial bus and bi-directional Ethernet.
4. At least 2 Mbytes of storage space.
5. Computing platform based on Java.
6. There should be an appreciable number of input/output pins for digital or analog interfacing, serial and parallel interfacing.

Even though there is a wide variety of processing platforms available, not very many meet the aforementioned requirements of an embedded technology with native support for real-time, object-oriented programming and communication network. For this reason, we selected the aJile microprocessor (see Table 4.1). The chipset is only US$25 (2004 price). The next section gives a brief description of the processor with emphasis on the features of interest to us.

4.4.1 The aJile Processor

The aJile architecture uses JEM2 as a direct-execution Java processor that is available as both an IP core and a stand alone processor (aJ-100, 2001). The data path is made up of a 32-bit ALU, a 32-bit barrel shifter and the support for floating point operations (disassembly/assembly, overflow and NaN detection). The control memory is a 4K by 56 ROM to hold the microcode that implements the Java bytecode. An additional RAM can be used for custom microcode to implement new instructions. The aJile inventors report that this feature can increase the efficiency of frequently used algorithms by 5 – 50 times by decreasing execution overheads. This feature is also used to implement basic synchronization and thread scheduling routines in microcode to yield context-switching of 1 μs. A Multiple JVM Manager (MJM) supports two independent, memory protected JVMs, which can execute with a deterministic schedule and full memory protection (Fig. 4.2).

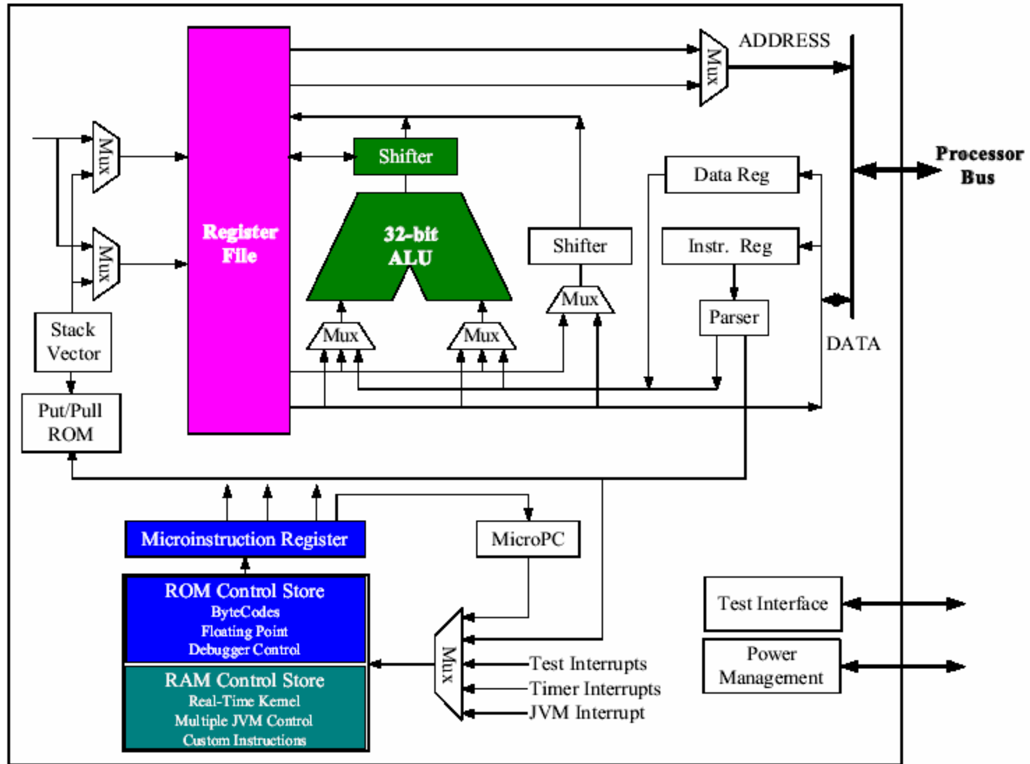


Figure 4.2: The aJile JEM2 Processor (aJ-100, 2001)

Currently, there are two silicon versions of JEM2 microcontrollers: the aJ-80 and the aJ-100. The aJ-100 shown in Fig. 4.3 (aJ-100, 2001), provides a generic 8-bit, 16-bit or 32-bit external bus interface and can be clocked up to 103 MHz, while the 66-MHz aJ-80 only provides an 8-bit interface. Both versions are made up of the JEM2 core, the MJM, 48-KB zero wait state RAM and peripheral components, such as timers, I/O's, a real- time serial communication bus (SIP) and UART. 16KB of the RAM is used for the writable control store and 32 KB for storage of the processor stack. Both microprocessors are bundled with J2ME-CLDC Java runtime system, optimizing application builder, and a very basic debugging tool. Complete implementations for real-time networked embedded Java applications are available (aJ-100, 2001 and Systronix, 2003).

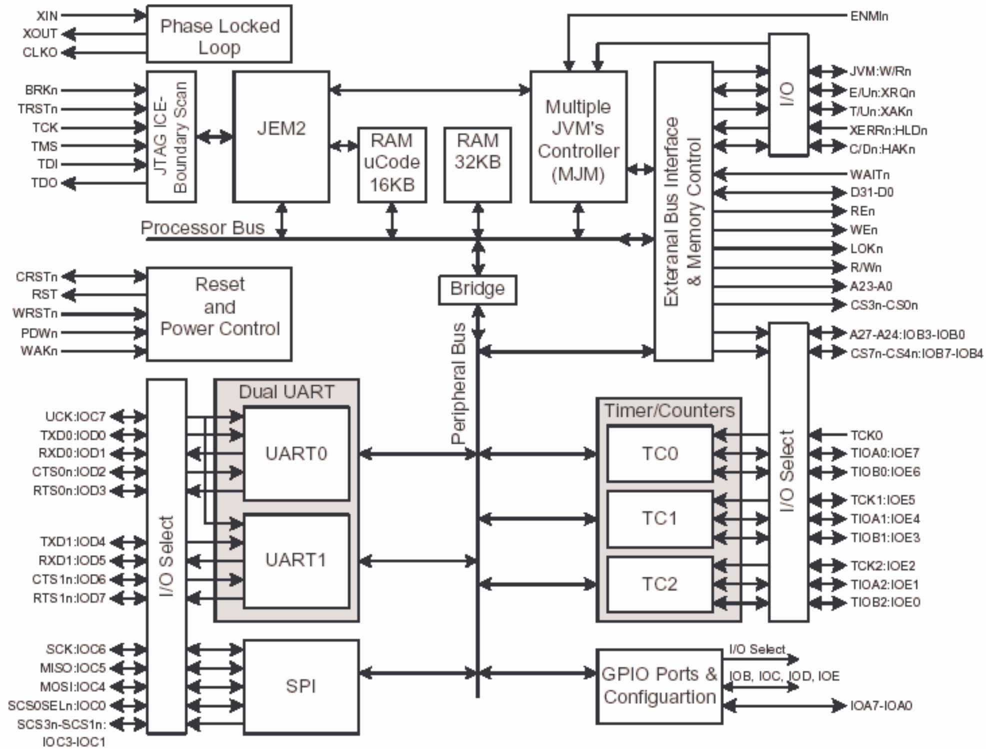


Figure 4.3: aJ-100 Architecture (aJ-100, 2001)

4.4.2 The JStick Platform

Systronix Inc. (2003) produces about five different boards with the aJile microprocessor. Their JStick microcontroller meets most of our preferences such as small form factor, I/O (input/output) interfaces, and network support. JStick is a Single Board Computer (SBC) with a SIMM30 (Single In-Line Memory Module) format. It features a host of facilities to provide most controller functionalities. The following is summary of the portfolio of this device:

- Processor: aJ100 (100 MHz).
- Memory: 2-Mbytes SRAM and 4-Mbytes flash.
- Power: switching power converters which provide 5 V and 3.3 V for peripheral devices.
- Power supply: unregulated DC of 9-14 volts or regulated DC power of 5V.
- Communication: serial I/O, 10BaseT Ethernet (including the RJ45 jack).
- Bus Interface: A high speed I/O expansion bus.

- Real-time network: SPI (Serial Peripheral Interface).
- Multiple timers, counters, PWM, interrupt inputs, etc.

4.5 Motion Controller Hardware

A motion controller is the most important element in a motion control system. Next to choosing a proper motor, the selection or design of a motion controller is the designer's most important decision. The fundamental function of a controller is to compare two signals: the command signal from the microprocessor and the position feedback signal from an encoder, resolver or tachometer. The position feedback signal is subtracted from the reference position to provide a following error which is converted by a digital-to-analog converter (DAC) to analog voltage for the servo amplifier. The controller's prime duty is to minimize the position error without causing system instability. With an appropriate motion controller in place, the designer can focus on stabilizing and programming the system.

There are two broad categories of motion controllers: application specific controllers and general purpose controllers. Application specific controllers are usually more expensive and have features, performance and packaging specific to certain types of applications (e.g. CNC). Since they are highly tailored for specific applications, they are not usually suited for general purpose tasks. On the contrary, general purpose controllers host a variety of communication interfaces and flexible programming to meet a variety of applications effectively. There are many configurations of general purpose controllers, the most common being board-level and stand-alone controllers.

Board-level controllers include a microprocessor containing the logic embedded in a communication bus such as a PC/ISA, VME or STD bus. Programming is accomplished through fast parallel processing with program storage on storage facilities residing on the controller. The execution sequence occurs when the program is downloaded to a non-volatile memory chip on the controller and executed. These controllers benefit from high-level language interfaces (DLL, C, etc) and available functionality found in the host. Its low cost, standard host architecture and rapid programming make board-level controllers the most popular control solution. The disadvantage of using board level controllers is that they require a separate host interface and are therefore not ideal for situations requiring portability.

Stand-alone controllers circumvent this by having their own host together with the controller in one package. Programs are typically downloaded from a PC through a serial communication link to an EEPROM or battery-backed RAM on the controller. Obviously, a disadvantage is that program size is limited to the on-board memory capacity of the controller host.

Motion controllers owe their speed and precision to high level ASCII-based or compiled programming languages plus powerful microprocessors, counters, etc. Although some controllers can control up to 32 axes of motion, most applications have much fewer axes. Motion controllers are usually capable of controlling a number of different components such as DC motors (brush or brushless), stepper motors, induction motors, pneumatic/hydraulic servos and proportional servo valves. In addition, they can also provide multitasking, PLC interfacing and analog and digital I/O control to control solenoids, relays, switches etc for complete machine control.

There are three basic types of motor control in motion applications: position, velocity and torque. Position (or current) control is found in servo and some stepper applications and employs feedback devices to close the position loop and make the system repeatable and dynamically responsive. The controller senses the speed increase or decrease via the feedback device and issues a new command voltage to maintain the original motor speed. Velocity control specifies the load velocity for a prescribed time interval and is not concerned with load position. The mode uses either the feedback from the motor's back EMF signal or a tachometer as a regulating signal, often as part of the inner velocity loop tied to an amplifier. Torque control's emphasis is on delivering constant torque regardless of load positions or velocity changes.

The most common motion profiles are point-to-point and coordinated motion. Point-to-point profiling involves constant and repeatable movements from one point to another such as pick and place, drilling and scanning. Coordinated motion involves tight synchronization of independent axes along a path such as is common in CNC, web lines, contouring, grinding, etc. This application requires that the load position follow the commanded position in a very predictable manner with high stiffness (loop gain) in order to reject external disturbances such as changes in load. On the contrary, point-to-point profiles typically are not as concerned with precise path motion as with settling times,

move times and velocity profiles. There are other motion profiles which are more complex. These include electronic gearing and cams, master/slave synchronization, linear and circular interpolation, contouring, cubic spines, parabolic profiling, spline interpolation, helical interpolation, S-curve acceleration and acceleration feed-forward. The controller uses a filter such as PID to stabilize the system and reduce the following error to an infinitesimal value. A more sophisticated filter achieves high dynamic response, high accuracy, minimum settling time and reduced instability, at the cost of computational resources.

4.5.1 Motion Controller Design Options

The decision to select an appropriate motion control scheme depended on a number of factors. The options considered are as follows;

- JStick as stand-alone motion controller
- Use auxiliary hardware in conjunction with JStick for motion control
- COTS motion controller board; Interface an off-the-shelf motion control board to JStick
- COTS motion control chip; design a motion controller board based on a COTS motion control chip.

The first option requires coding control filter algorithms such as PID, polling sensor (encoder) data and using JStick's hardware counters and software to decode encoder pulses for the filter, and sending PWM signals to an amplifier or motor driver. In terms of hardware, it is the simplest to implement but will present challenging performance issues. The embedded device will be overburdened to meet motion control requirements and at the same time provide communication and user interface facilities. We envisage that we will have to use low frequency sampling – 10Hz at most – for the PID filter algorithm. The setup will also result in very limited free memory storage for other coding that we might need to implement. A solution to the limited JStick computing budget is to use a servo interface board to either handle the quadrature decoding of encoder signals, digital to analog conversion of motion signals or both. In this case, JStick will be responsible for only reading/writing signals from/to the auxiliary board, motion control (commands and filter) and motion profiling. There are a few COTS that offer such options but their costs are too high for a cost-effective solution to our Java-based motion

controller. Secondly, the potential for JStick to interface with the peripheral device is the absolute determining factor; issues regarding interfacing include signal types, timing parameters and ease of integration. These factors prompted us to opt for an in-house designed motion control board as opposed to a proprietary product.

4.5.2 Motion Controller Board Design Criteria

The following features were considered for the motion controller board design:

- A programmable motion controller chip with a compensator (at least a PID filter), a sampling rate of at least 1 KHz for high performance (Bellini et al., 2003), motion profiling, quadrature decoding and a real-time means of populating an integrated buffer with motion set-points from the JStick host. The chip should also be capable of providing encoder readings to allow monitoring or adaptive control.
- A Digital to Analog (DAC) chip, which easily interfaces with the motion control chip and provides at least a 12-bit resolution and a voltage output range of +/- 10 V. It is worth noting here that the chip timing parameters should be compatible with that of the motion control chip.
- An encoder receiver chip (if the motion controller needs one) and circuitry for filtering out noise. Encoder signals are very susceptible to noise, especially single line encoders. This is compounded by long transmission lines and the use of relatively cheap encoder technology like totem-pole and open-collector types.
- Digital I/O with interrupts for enable-signal on motor driver (amplifier) and mechanical switches such as limit and home position switches.
- Logic chips such as flip-flops for buffering signals and signal inverters.
- Power converter chip for devices requiring power levels which are unavailable on the JStick.

The following criteria set further guidelines for selecting appropriate peripheral devices for the motion controller board.

- Compatibility with JStick’s HSIO interface: since JStick’s HSIO provides facilities for interfacing with peripheral devices, any such device should be capable of interfacing with the HSIO byte-wide data bus. Signals are TTL levels; therefore it will be advantageous to select TTL hardware in order to avoid voltage level shifter circuitry.

- Timing granularity: peripheral timing characteristics should be within JStick's timing range without clocking down JStick from its maximum speed of 103 MHz. A clock down will imply slower computational processing of supporting and other algorithms.
- Power levels: peripheral devices power should be capable of utilizing onboard power resources either directly or through a simple power converter which must also depend on JStick's power sources. In addition the peak power demand should be within JStick's power supply budget, which is about 1W. The vision is to have a compact system with one main power source for both the JStick and the motion control module.
- Ease of handling: components with extremely small form factors or pin spacing will be difficult to work with.

4.5.3 Motion Controller Chip Selection

There are a number of commercial motion controller chips on the market. A few of these were identified as potential candidates for the motion controller board design. The National Semiconductor's LM628 (LM628/LM629, 2003) was eventually selected based on the criteria outlined above, including its low price, simplicity, and proven capabilities. In Chapter 2, this particular chip is featured in the CLARAty architecture for space rovers (K7 rover). Cost and technology comparison of different motion controller chips and boards are provided in Appendix A.

4.6 Conclusion

This chapter provided some insight in embedded Java technology and the enabling technologies available. The JStick microcontroller, which is based on aJile's aJ-100 Java processor, was selected as the host computer for the IMC nodes. The selection was based on the several on-board facilities that JStick provides, including real-time scheduling, timers/counters and Ethernet – to mention a few. A motion controller chip, LM628, was also selected for an in-house design of a motion controller board. The next chapter details the hardware and timing features of JStick and how it interfaces with the LM628 chip on the IMC motion controller board. Timing analysis and the board design procedure is also discussed.

5. MOTION CONTROLLER BOARD DESIGN

5.1 Introduction

This chapter focuses on the hardware design of the motion controller board for the IMC node. The schematic of the board is shown in Fig. 5.1. The hardware architecture is a multi-tier system with independent controllers for each joint of the robot. Each joint controller is made up of a JStik (Systronix) microcontroller and a motion control board consisting of a LM628 PID motion controller chip (National Semiconductor), a 12-bit DAC, and various I/O interfaces. The LM628 has an 8-bit host interface and is made up of four major functional blocks; the Trajectory Profile Generator, Loop Compensating PID Filter, Summing Junction and Motor Position Decoder. The output interface is programmable for an 8-bit or 12-bit DAC. Its maximum sampling rate is 2.932 kHz.

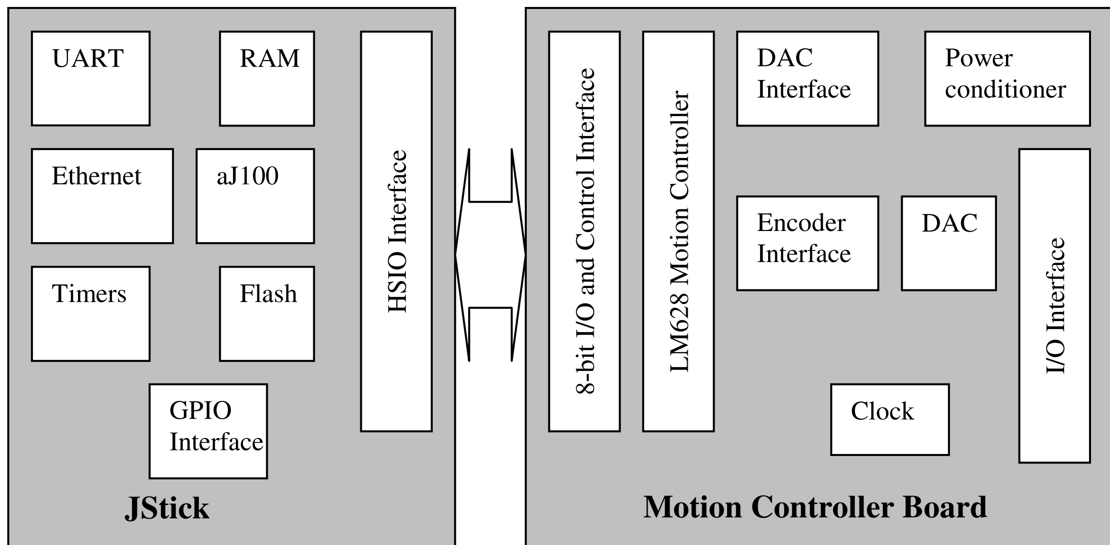


Figure 5.1: Peripheral Motion Controller Board Architecture

5.2 JStick's Peripheral Interface Signals

In order to interface this board with peripheral I/O devices, JStick uses an asynchronous, memory-mapped High Speed I/O Interface (HSIO). The HSIO provides byte-wide (8-bit) data, twelve address bits, read and write strobes, two chip selects, 3.3 VDC power and ground. More chip selects can be easily decoded from the 12-bit address. HSIO signals interface directly to 3.3V TTL and CMOS or 5V TTL devices (Systronix, 2001). In order to configure the HSIO for a peripheral device one needs to properly select timing parameters to match the timing specifications of the peripheral device. Fortunately, the bus timing can be varied to support most peripheral speeds. The bus timing is controlled by three features; a Phase Locked Loop (PLL), the aJile CPU external bus interface settings and the HSIO memory mapped address bits.

5.2.1 The Phase Locked Loop (PLL)

The aJ-100 chip utilizes a PLL circuitry shown in Fig. 5.2 to generate the high-speed internal clock from a low frequency external oscillator. JStick uses a 7.3728-MHz clock to generate internal clocks of up to 103 MHz for its aJ100 CPU. The outputs can be configured through the PLL Configuration Register. The clock output (CLKO) signal is derived from the CPU clock divided by 2, 4 or 8 (Systronix, 2003).

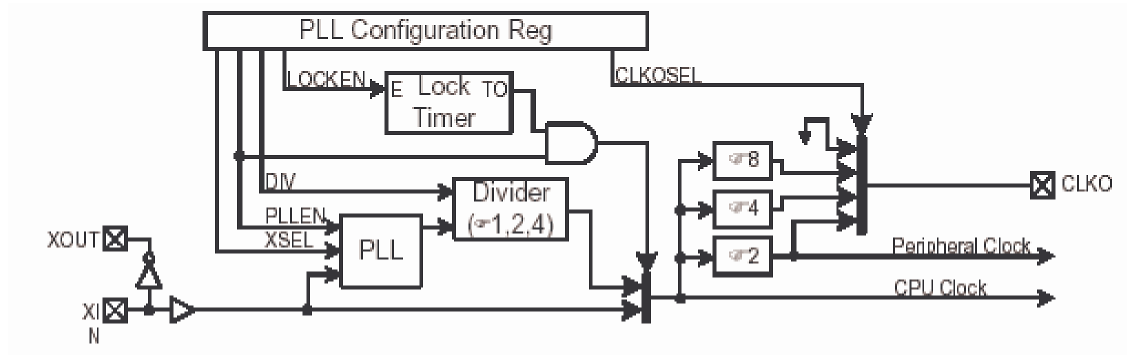


Figure 5.2: aJile PLL Circuit Diagram (aJ-100, 2001)

5.2.2 The External Bus Interface

The aJile's external bus interface (EBI) generates the signals to control access to external memory and peripherals devices. The EBI may be configured to support 32-bit, 16-bit, and 8-bit memory devices or a combination of these memory widths. The EBI provides eight chip selects (CS0n..CS7n) to control external memory devices and peripherals and each chip select has a configuration register to specify the setup times, hold times, wait states and memory widths (Systronix, 2003). The HSIO is controlled by CS5n. Its configuration register is shown in Table 5.1 below.

Table 5.1: Chip Configuration Register

Bit Positions	**31:12**	**11:10**	**9:8**	**7:6**	**5:2**	**1:0**
Field Name	unused	CS setup Ncs	Address setup (Nas)	Address hold (Nah)	Wait states (Nws)	Bus width

- CS setup configures the number of clock cycles the CS line should be valid before the assertion of the control strobe (REn/WEn). The range is 0 to 3 times the clock cycle (0 – 3T).
- The Address setup defines the number of clock cycles the address should be valid before the chip select signal. The range is 0 – 3T.
- The Address hold defines the number of clock cycles to hold write data and addresses following the release of the transfer control strobes. The valid range is 0 – 3T.
- The Wait states bits extend the duration of the bus transactions from 0 to 15. Each states increments the memory cycle by 1T.
- The Bus width defines three data bus configurations; bits = 0 defines an 8-bit memory transfer (D[7:0]), bits = 01 a 16 bit transfer (D[15:0]) and bits = 11 a 32 bit transfer.

5.2.3 HSIO Bus Address Space and Timings

The HSIO timing granularity is a multiple of the CLKO period (Systronix, 2001). At the maximum rated frequency of 103 MHz, JStik CLKO options are 51.6, 25.8, or 12.9 MHz. The HSIO bus uses 12-bits of address, 0-FFF and two 12-bit chip selects; each one

has its own 12-bit address space. More chip selects may be decoded within a given HSIO address space. The HSIO control logic uses address lines A[19:A16] exclusively to set the bus cycle wait states. The timing duration T_w, in CLKO periods T_c of the inserted wait states is defined as.

$$T_w = ((A[19:16] \times 2) + 1) \times T_c. \quad (5.1)$$

A[19:16] = 0 gives the fastest transaction. The state of address A20 (0 or 1) and the value of chip select configuration register CS5_CR [11:10] control the assertion of the HSIO read/write control strobes (X-RD/X_WR).

5.2.4 JSimm Interface and Signals

JStick provides several general-purpose I/O's (GPIO) on its 30-pin JSimm interface. aJ-100 includes five 8-bit discrete GPIO ports (A to E). Most of the GPIO pins are multiplexed with the I/O signals from other resources on the CPU such as timers, counters, etc. A few of these pins (port A) are capable of driving/sinking 24 mA while the others can drive/sink 8 mA. Each port is capable of generating an interrupt unlike the HSIO bus. The JSimm interface provides unregulated 15 V, regulated 5 V and ground. All signals on JStick are powered by 3.3-V logic, are TTL level compatible, and are 5 volt tolerant. As such they will interface directly to 3.3-V TTL and CMOS or 5-V TTL devices.

5.3 The Interface Design

The LM628 is a TTL device with signal levels shown in Table 5.2. Therefore there is no need for a buffer or level shifter to mediate signals to/from JStick.

Table 5.2: LM628 Signal Levels

Parameter	Limits	
	min	**max**
Logic 1 Input Voltage	2.0 V	
Logic 0 Input Voltage		0.8 V
Logic 1 Output Voltage	2.4 V	
Logic 0 Output Voltage		0.4 V

Figure 5.3 shows the simplicity of the interface connections between the JStick HSIO and a single LM628. An address decoder shown in Fig. 5.4 may be used to decode HSIO address bits for connections with multiple LM628 chips. In our current board design, we use a single LM628 chip.

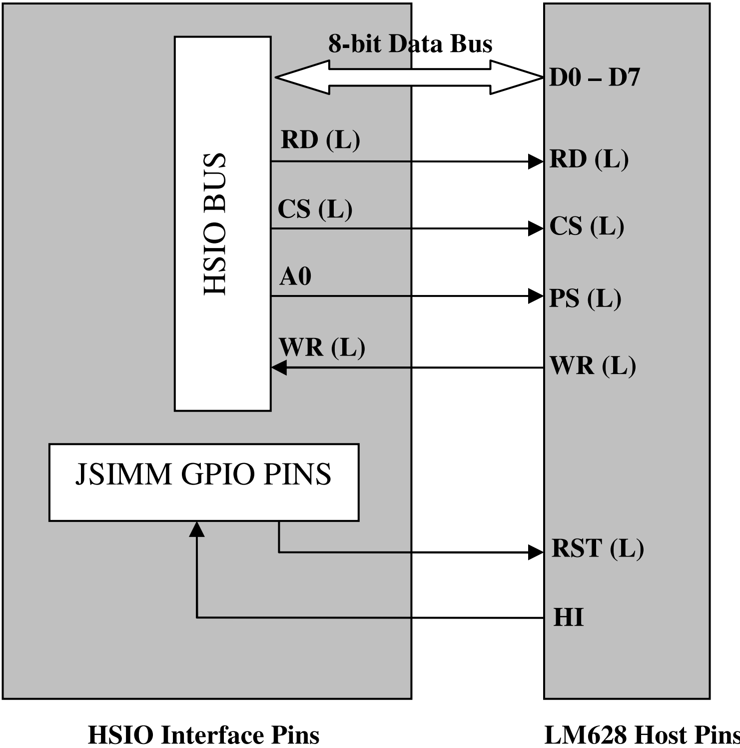


Figure 5.3: JStick LM628 Interface

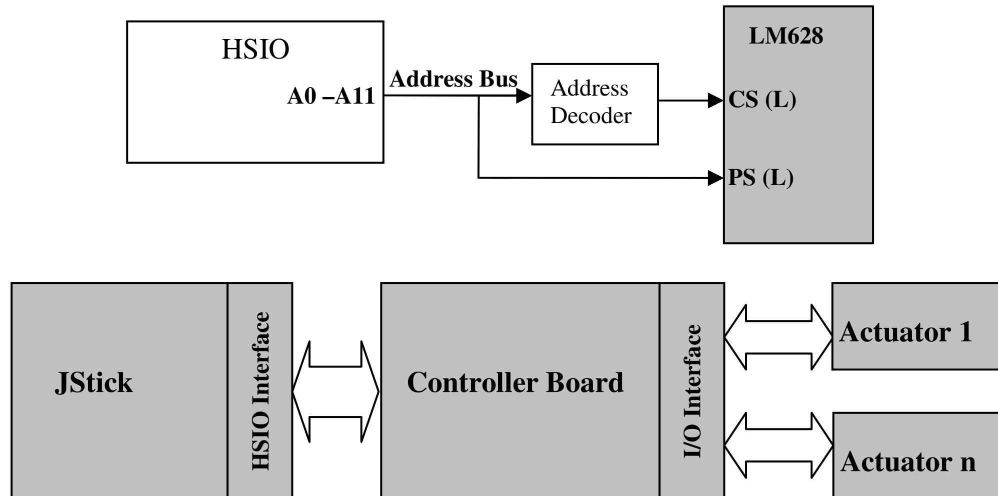


Figure 5.4: Interface Architecture for Multiple LM628

All control pins (strobes), Chip Select (CS), Port Select (PS), Read (RD) and Write (WR) assert low (logic 0). The Chip Select is for selecting the LM628 for writing and reading operations. The Read pin is for reading data from the LM628 and also its status while Write controls command and data write transactions. Port Select is used to select the LM628's command or data port depending on its logic level. The Host Interrupt (HI) signal alerts JStick that an interrupt condition has occurred. Lastly, the Reset (RST) pin resets the LM628 to default conditions. The next phase of the design is to establish communication between JStick running at 103 MHz and the much slower LM628 clocked at 6 MHz. Like in any processor interface design, the timing requirements of the LM628 must be stringently met to ensure the integrity of transactions such as data read/write, command byte write and status byte write. Consequently, we configure the HSIO parameters described in the previous section with the aid of a timing analysis. Table 5.3 below shows the timing requirements of the LM628 chip.

Table 5.3: LM628 Timing Requirements

Timing Interval	T#	Limits		Units
		Min	Max	
DATA WORD READ TIMING				
Chip-Select Setup/Hold Time	T7	0		ns
Port-Select Setup Time	T8	30		ns
Port-Select Hold Time	T9	30		ns
Read Data Access Time	T10		180	ns
Read Data Hold Time	T11	0		ns
RD High to Hi-Z Time	T12		180	ns
Busy Bit Delay	T13			ns
Read Recovery Time	T17	120		ns
DATA WORD WRITE TIMING				
Chip-Select Setup/Hold Time	T7	0		ns
Port-Select Setup Time	T8	30		ns
Port-Select Hold Time	T9	30		ns
Busy Bit Delay	T13			ns
WR(L) Pulse Width	T14	100		ns
Write Data Access Time	T15	50		ns
Write Data Hold Time	T16	120		ns
Write Recovery Time	T18	120		ns

The timing intervals may be constraints or delays; a constraint establishes a relationship between the two signal edges that must be maintained within the Min/Max values; setup, hold times and pulse widths are constraints. A delay represents a cause-and-effect relationship between two signal edges, such as the propagation delay due to *Data Access Time*. Using the *Timing Designer* software (Timing Designer, 2003), timing diagrams are created by adding these delays and timing constraints to JStick's HSIO timing diagram. When the timing parameters of JStick are updated, *TimingDesigner* instantly performs a true worst-case timing analysis and automatically flags any violations of the timing diagram specification. Using aJile's PLL settings, the lowest clock (CLKO) speed we can derive for the HSIO at maximum processor speed (103 MHz) is 12.9 MHz, i.e., CPU Clock divided by 8. Intuitively, this is too fast for LM628 which is clocked at a maximum of 6 MHz. The obvious is to clock down the aJ100 CPU; using a PLL multiplier of 8, we get a CPU clock speed of 58.98 MHz and a corresponding CLKO (bus speed) value of 7.37 MHz. Figure 5.5 shows the resulting timing diagram of the HSIO bus transactions with the LM628. Commands are written to LM628 by bringing WR and PS low. When PS is high, WR brought low writes data into LM628 and similarly, RD is brought low to read data from LM628. Address pin A0 of the HSIO drives the PS pin and we recall that CS has to be brought low to select LM628 for all transactions.

Subsequently, the maximum read and write transaction times in Fig. 5.5 are approximately 800 and 700 nanoseconds respectively – with no timing violations.

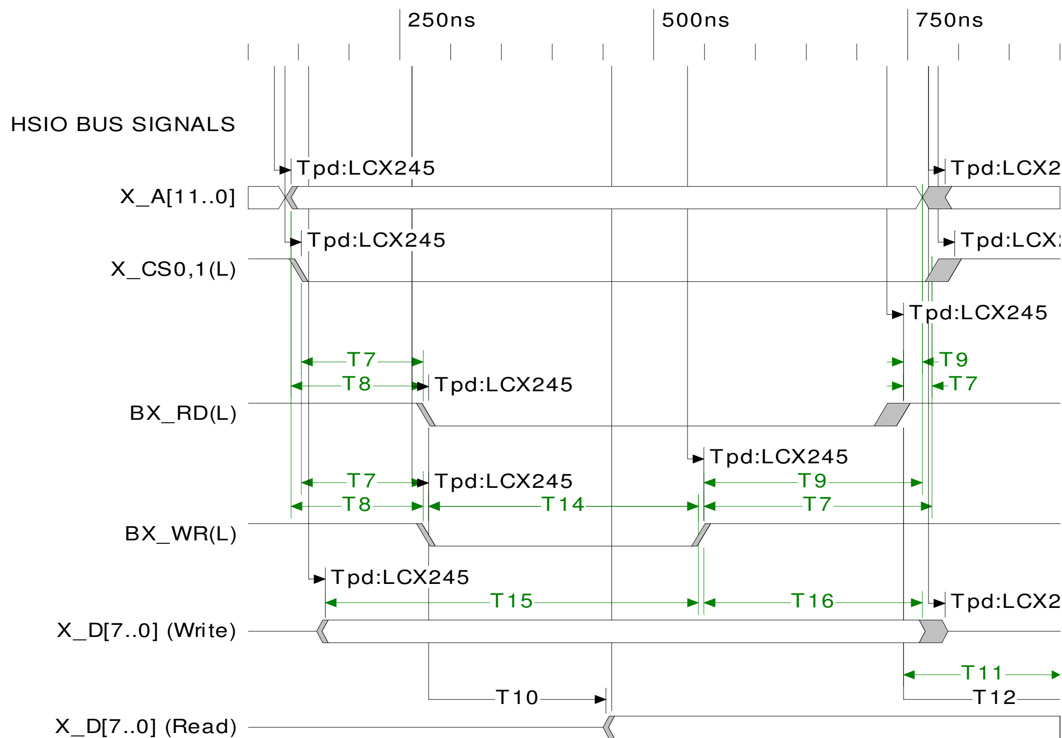


Figure 5.5: HSIO Timing Diagram with Bus Speed of 7.37 MHz

Next, we investigated the possibility of maintaining the maximum aJile CPU speed (CLKO = 12.9 MHz) since there are many other processes sharing the time space. Inserting a 232-ns wait state in the HSIO by setting A[19:16] to 1 (see 5.1) results in the timing diagram shown in Fig. 5.6. There is yet a timing violation of -9.24 ns on the Write Data Hold Time (T16), which implies that data output from JStick to LM628 will lag timing constraints.

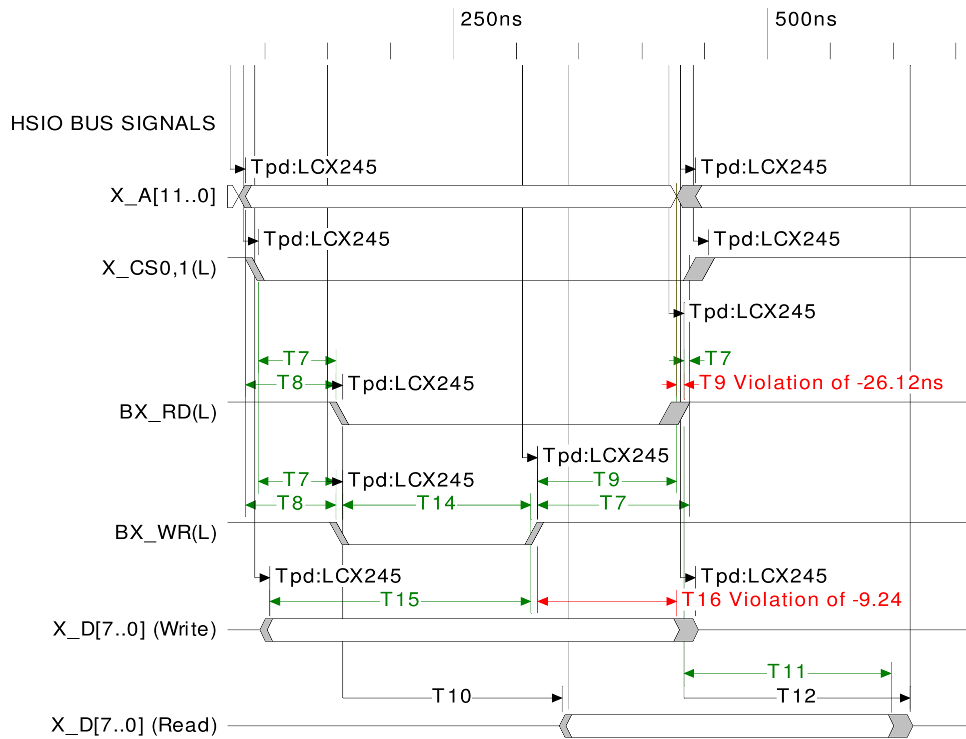


Figure 5.6: HSIO Timing Diagram with Bus Speed of 12.9 MHz Showing Violations

To resolve the above problem, the bus interface (EBI) control strobes was compelled to use the maximum address hold time (recall from section 5.2.2 that bit-fields 7:6 of the EBI chip configuration register influence the number of clock cycles to hold write data and addresses following the release of the EBI's transfer control strobes). The resulting timing diagram (Fig. 5.7) shows a lag of 16 ns on the read control strobe, however since there is no violation in the corresponding data propagation we regard this as benign. The maximum read and write transaction times are approximately 500 and 400 ns respectively.

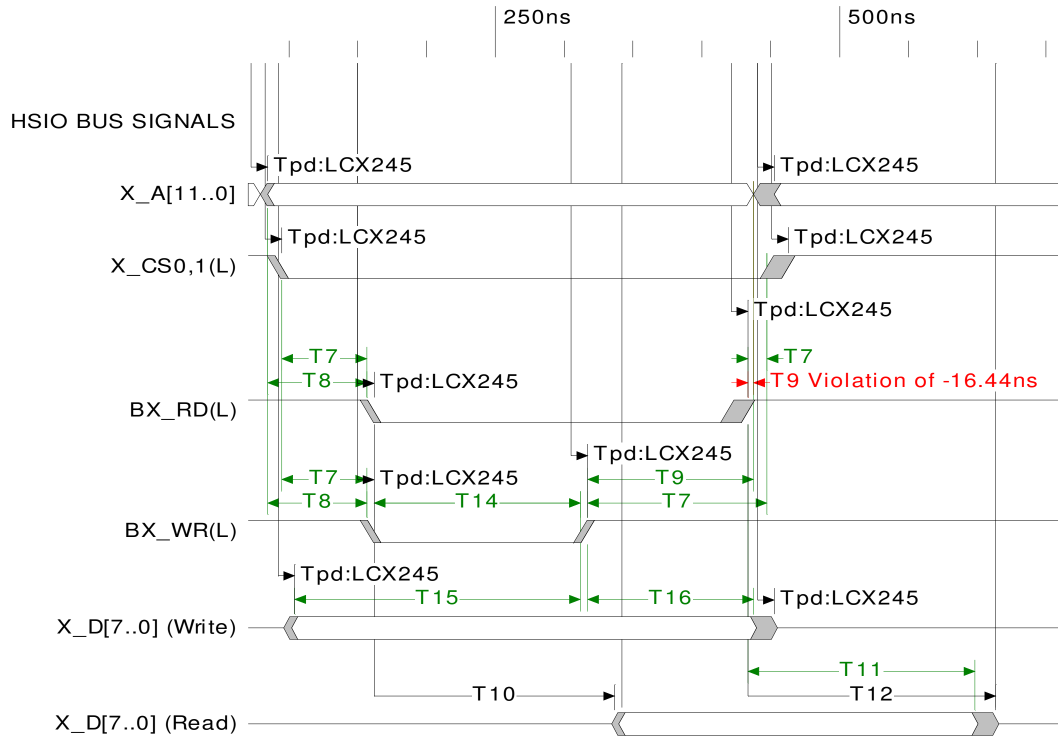


Figure 5.7: Final HSIO Timing Diagram with Bus Speed of 12.9 MHz

5.3.1 Clocking

The hardware architecture provides two means of clocking the LM628 chip; static clocking and dynamic clocking. There is an onboard 6 MHz clock which provides very accurate clocking for the chip. The clock can be enabled or disabled in software but the output frequency is fixed. Alternatively, the aJile chip provides a Timer/Counter circuitry comprising of a 16-bit *prescalar* and three 16-bit timer/counter blocks. The *prescalar* is a continuous count-down timer that divides aJile's internal peripheral clock (or an external clock) by

$$PrescalerReloadRegisterValue + 1; \quad 0 \leq PrescalerReloadRegisterValue \leq 65535;$$

Where *PrescalerReloadRegisterValue* is the value of the *prescaler* register.

The internal peripheral clock shown in Fig. 5.2 divides the CPU clock by two. There is also a reload register value, *ReloadRegisterValue*, associated with each timer/counter. Its value is transferred into the corresponding timer when a reload event occurs. A reload event can be specified in software or when the timer reaches a 0×0000 value. In effect the timer's low pulse width is driven by the *PrescalerReloadRegisterValue*

while the high pulse width is determined by the *ReloadRegisterValue*. The flexibility of adjusting the clock rate in software or by trigger events is extremely advantageous in designing distributed systems where clock rates have to be synchronized occasionally. Additionally, the sample rate of the LM628 can be varied very conveniently for different requirements. Table 5.4 shows the results of different settings to derive the LM628's clock range of 1 – 6 MHz using the class following. Pulses and frequencies were measured with a logic analyzer. The register values selected result in near symmetric square waves.

Table 5.4: Timer Settings

Prescaler ReloadRegisterValue	***Reload RegisterValue***	**High Pulse Width (ns)**	**Low Pulse Width (ns)**	**Frequency (MHz)**
3	1	70	80	6.4
4	1	96	96	5
5	1	112	120	4.3
7	1	150	160	3.4
11	1	240	230	2.15
24	1	490	480	1.032

The method for setting the timer is listed below.

```
1.  public JStickTimer2(double freqMHz) {
2.  //calculate prescalerReloadRegisterValue and reloadRegisterValue from freq given
3.  prescalerReloadValues(freqMHz);
4.  //configure the prescaler to use the internal preipheral clock = 50MHz
5.  TimerCounter.setPrescalerClockSource(TimerCounter.INTERNAL_PERIPHERAL_CLOCK);
6.    //set the PrescalerReloadRegisterValue and enable it
7.  TimerCounter.setPrescalerReloadRegisterValue(prescalerReloadRegisterValue);
8.  TimerCounter.setPrescalerEnabled(true);
9.  //configure TIMER2 output
10. TIMER2.setMode_IO_Line_B(TimerCounter.TIMER_2_OUTPUT);
11. TIMER2.setExternalSampleMode(
12. TimerCounter.CONTROL_SIGNAL_2_TRAILING_EDGE_ENABLED);
13. TIMER2.setExternalTimerEnableMode(
14. TimerCounter.TIMER_ENABLED_ONLY_VIA_MTEN_AND_TRIGGER);
15. //set the setReloadRegisterValue
16. TIMER2.setReloadRegisterValue(reloadRegisterValue);
17. TIMER2.setMasterTimerEnabled(true);
18. }
```

5.4 LM628 DAC Output

The LM628 precision motion controller outputs data to a DAC (Digital to Analog Converter) on its ports DAC0-DAC7. Its output port can be configured for either a latched

8-bit parallel output or a multiplexed 12-bit output. While the 8-bit output can be directly connected to non-input-latching DAC (Digital to Analog Converter), the 12-bit output has to be demultiplexed using an external 6-bit latch. The IMC motion controller board uses the 12-bit output mode for better resolution. In this mode two multiplexed 6-bit words are outputted by pins DAC2-DAC7, the less-significant 6-bit word first followed by the more significant word. DAC0 and DAC1 are converted into control signals; DAC1 is driven low to latch the less-significant word while the positive-going edge of DAC0 is used to strobe the output data. In effect the 8-bit output port is converted to a 6-bit port with two control strobes. Currently, there is no commercially available 12-bit DAC chip that allows direct interfacing of 6-bit buses. Versatile ones interface with microprocessor bus widths which are multiples of four. The problem is resolved by a combination of an input-latching 12-bit DAC (AD667) from (Analog Devices, 2003) and logic devices. The bus interface logic of the AD667 consists of four independently addressable registers in a pair of ranks. The first rank comprises of three four-bit registers which can be loaded directly from a 4-, 8-, 12-, or 16-bit microprocessor bus. Once a 12-bit data word has been assembled in the first rank, it can be loaded into the 12-bit register of the second rank. The latches are controlled by the addresses, A0–A3, and the CS input. All control inputs are active low. The four address lines each enable one of the four latches, as indicated in Fig. 5.8. All latches in the AD667 are level-triggered, implying that data present during the time when the control signals are valid will enter the latch. When any one of the control signals returns high, the data is latched. Since LM628 outputs 6 bits in each cycle, we used an external D-Type flip-flop and inverter combination (Fig. 5.8) for the latching process.

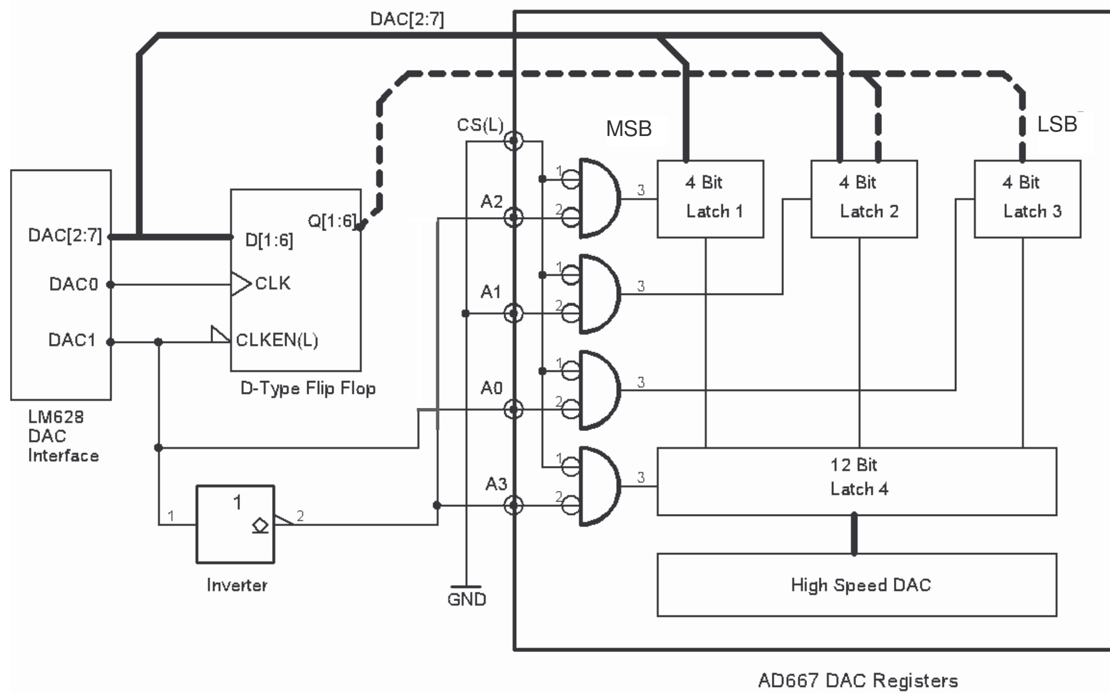


Figure 5.8: DAC – LM628 Interface Architecture

The following is the procedure for a 12-bit data transfer from the LM628;

- The AD667's chip select (CS) and Latch 2 are permanently enabled by hardwiring CS and A1 to digital ground.
- LM628 drives DAC1 low to transfer the less-significant 6-bit word to the Flip Flop's Q outputs. Subsequently, the lower 4 bits of this word are latched to Latch 3 while the next 2 bits are latched to the lower half of Latch 2. Simultaneously, the inverter inverts the DAC1 signal to disable Latches 1 and 4 through control addresses A3 and A2.
- LM628 drives DAC1 high to transfer the more-significant 6-bit word. The inverted DAC1 signal drives A3 and A2 low, thus enabling Latches 1 and 4. The lower 2 bits are latched to the upper half of Latch 2, which now has 4 bits, while the next 4 bits are latched to Latch 1. Since Latch 4 is enabled, all 12 bits in the first rank are loaded to it.

Note that without the intermediary flip-flop, bits in Latch 2 will be ill-formed during the above process. Our architecture configures the AD667 to function in a ± 10 V bipolar mode which requires a power supply of ± 15 V.

5.4.1 Encoder Interface

The LM628 provides three channels of quadrature encoder input for A, B and I encoder signals. Each time the quadrature decoder detects an edge either on the A or B encoder phase, it increments or decrements a counter, depending on the sense of motion. The relative phasing of the A and B phases determine the direction of motion. The resulting decoding means that an N-pulse encoder produces N × 4 counts per revolution. The I signal is gated to the A and B phases so that it is active over a region between two consecutive phases. The region which is called index position may be used to set a home or reference position. The board provides additional support for two kinds of encoders, differential and single-ended totem pole input encoders. Encoders with differential outputs drive two lines per signal (A and A*, etc). While one is driven high, the other is driven low, giving greater signal amplitude, and greater immunity to noise. The interface on the board uses a 26LS32 differential line receiver to convert the signal pairs to single phases for the LM628. Single-ended totem pole input encoders are more sensitive to noise and cable length, therefore additional noise immunity may be necessary. In this configuration, the board provides a voltage divider resistor circuit on the inverted inputs of the 26LS32 line driver, as illustrated in Fig. 5.9.

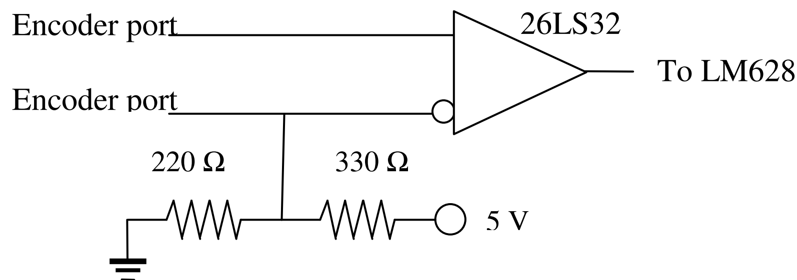


Figure 5.9: Receiver Line Filter for Single-Ended Totem Pole Encoder

5.4.2 Power Supply and Noise Emission

JStick provides three power sources with a total budget of 1 A; regulated +3.3 V and +5 V, and unregulated +17 V. Except for the AD667 DAC, all devices on the board are powered by the 5 V source. Since the DAC requires ± 15 V, a miniature regulated DC-DC switched-mode converter (C&D Technologies, 1997) is used to convert the 17 V to

the voltage levels required by the DAC. A switching converter is generally very compact and efficient but can be very noisy if not well isolated. The converter selected has a fixed characteristic operating frequency, making filtering relatively simple, compared to pulse-skipping types (C&D Technologies, 2000). Simple passive LC network filters were used at both inputs and outputs. For high precision motion control, noise immunity is critical. Hence the following criteria were used for the motion board design.

- The motion controller chip receives two-phase quadrature signals provided by an incremental encoder. For better noise immunity, differential signals must be used to cancel out common mode noise. A receiver chip on the board is used for this purpose.
- Low pass filters on all power lines.
- Option of using pull-up resistors for open collector and totem-pole encoders.
- Proper termination of encoder signals.

5.4.3 Board Schematics

The schematic diagrams and printed circuit board layout for final production were created with the Eagle software (CadSoft, 2003). The following were the guiding principles for laying out the tracks and components.

- No loops in supply lines.
- Wide tracks for power lines and a solid ground plane.
- Short high frequency tracks.
- Bevel or mitre track corners instead of sharp bends.

5.4.4 The LM628 Hardware Driver

For low level access to aJile's hardware, the aJile API includes a rich set of libraries such as *com.ajile.jem.rawJEM* for reading and writing primitives (Boolean, short, integer, long, float and double values). In the respective operations the read (RD) and write (WR) pins are driven to logic low. Since the aJ-100 chip has a 32-bit data bus width, it is more efficient to do word (integer) access to JStick's HSIO address space than an explicit byte write. Only the low byte of the word is actually used for the HSIO data bus – the upper bits are ignored by the HSIO hardware. The two least significant bits of the aJ100 CPU address do not go to the HSIO bus, which means that the aJ100 address space

is left-shifted by 2 in the HSIO address space. The *HSIO.class* in Fig. 5.10 below implements methods for configuring the HSIO and is the parent class for all read/write transactions with the motion controller. Apart from the last two methods of the class, all the others are direct logic and bit operations. The last two methods implement methods *rawJEM.getInt(PortAddress)* and *rawJEM.set(PortAddress*, data) for reading and writing data respectively.

```
1.  package com.IMC.drivers;
2.  import com.ajile.jem.rawJEM;
3.  import java.lang.Integer;
4.
5.  public class HSIO {
6.        public static final int HSIO_CS0_ADDRESS = 0x01400000;
7.        private int PortAddress = HSIO_CS0_ADDRESS;
8.  //constructor  creates a new HSIO
9.         // integer NewAddress; The new HSIO address of the port.
10.        // byte A19_16  is the clock divider bits of the HSIO address + 1.
11.        //integer A20 is the External Address Setup Control
12.        //Boolean SelectCS1; if true, chip select 1 is selected.  If false, chip select 0 is selected
13.       public HSIO(int NewAddress, byte A19_16, int A20, boolean SelectCS1) {
14.              ::       }
15.  // address setup
16.       public void setHsioAddress(int NewAddress) throws IllegalArgumentException {
17.          // map to hsio space, shift the bottom 2 bits that are always 0.
18.         // bits 0 to 13 are either not used or are the HSIO address. Clear them so it can be set
19.      // set the new port address.      }
20. //HSIO timing setup: bits A19:16
21.       public void setHsioTiming(byte A19_16) throws IllegalArgumentException {
22.          // Clear bits 19:16, of the HSIO address bits.
23.           // put the new clock divider in the bits that were just cleared.        }
24. // address wait setup: bit A20
25.        public void setHsioWait(int Tas) throws IllegalArgumentException {
26.         // Clear bit 20 of the HSIO address bits
27.         // Put the wait value in bit 20.      }
28. // read data from address and return value
29.       public int read() {
30.              return rawJEM.getInt(PortAddress);        }
31. // write data to address
32.       public void write(int data) {
33.              rawJEM.set(PortAddress, data);
34.       }   }//end of class
```

Figure 5.10: Pseudo-code for High Speed I/O (HSIO)

The LM628 driver class *LM628.class* illustrated below creates two objects of the *HSIO.class*. The first object, *addressA0*, configures the A0 address space on chip select 0

(CS0), while the first object, *dataAddress*, configures the HSIO CS0 base address for data I/O. Each address pin (A0-A11) holds logic high if there is read/write to it.

```
1.  package com.IMC.drivers;
2.
3.  public class LM628 extends HSIO{
4.  private static final int addr = 0x01;          //address A0
5.  private static final int addrBase = 0x00;      // base address
6.  private static final byte A19_16 = 0x01;       // clock divider)
7.  private static final int A20 = 0x01;           //address hold
8.
9.  //create HSIO.class object to configure address A0 on chip select 0
10. private static HSIO addressA0= HSIO (addr, A19_16, A20, false);
11. }
12. //create second HSIO.class object to configure base address
13. private static HSIO dataAddress = HSIO (addrBase, A19_16, A20, false);
14. }
15. //other methods
16. : : :
17. }
```

The LM628 read and write operations involve three internal registers; status-byte register where all interrupts are stored, high-byte register, and low-byte register (Hunt, 1999; LM628, 2003). The LSB of the status byte contains the busy bit which is set immediately after the host writes a command byte, or reads or writes the second byte of a data word. While the busy-bit is set, the LM628 will ignore any commands or attempts to transfer data, therefore it is imperative to query this status-byte frequently. This register is read by bringing RD and PS low – the PS pin is driven by the HSIO address pin A0. In the *LM628.class* the following method reads the status-byte:

```
31. public static byte read_Status_Byte() {
32.             return dataAddress.read() - dataAddress.getRawJStikAddress();
33. }
```

The busy bit, i.e. LSB of *status-byte* can then be checked as follows:

```
61. public static void check_busy_bit() {
62. while ( (read_Status_Byte() & 0x01) == 1){
63. }  }
```

Data written to a HSIO address space is retrieved by subtracting the initial address contents from its current contents. Since the above method does not read from the A0 address space, the A0 pin – hence PS, remains at logic low while the RD pin is driven low

to meet the condition for reading the status-byte. Similarly, when writing commands (e.g. PID filter configuration, run motor, etc) to LM628, PS and WR must be driven low as shown below:

```
64. public static void write_command(int CMD) {
65.  test.write(CMD); }
```

On the contrary, the A0 pin must drive PS high in order to read or write data to the LM628. The following methods make this possible:

```
66. public static int read_data() {
67. return testA0.read() - test.getRawJStikAddress();
68.             }
69. public static void write_data(int data) {
70. testA0.write(data);
71. }
```

5.4.5 Initializing

Immediately following power-up a hardware reset must be done before the LM628 can be programmed. This is executed by strobing the RST low for a minimum of eight clock cycles. The RST pin is hooked to JStick's reset pin, therefore both resets are executed simultaneously. Following a reset procedure the status-byte must read C4 hex or 84 hex. Subsequently, all bits in the LM628 interrupt register are reset to zero by the method *reset_interrupt_register(int)*.

5.4.6 Interrupt Service Routines and digital I/O operations

The hardware architecture drivers include classes for receiving interrupts on various general purpose I/O pins connected to the LM628 HI interrupt pin, and digital I/O such as interrupts from limit switches on the controlled device. There are also I/O pins for outputting digital signals. The aJile API provides base classes for all manner of typical microcontrollers I/O operations. LM628 interrupts are received and serviced in two ways; by polling its status-byte or receiving hardware interrupts on the HI pin. The *LM628_Interrupt.class* below shows how a motor-stall interrupt is received and serviced. The *Monitor.class* implements methods to communicate with the system coordinator computer. In this case, the coordinator is informed of an excessive position error condition.

```
1.  package com.IMC.drivers;
2.  import com.ajile.events.TriggerEventListener;
3.  import com.ajile.drivers.gpio.GpioPin;
4.  import network.Monitor;
5.  public class LM628_Interrupt {
6.          final  GpioPin pinA1 = new GpioPin(GpioPin.GPIOA_BIT1);
7.          Monitor monitor;
8.          //constructor
9.          public LM628_Interrupt(Monitor mon) {
10.                 monitor=mon;
11.                 pinA1.setPinReportPolicy(GpioPin.REPORT_RISING_EDGE);
12.                 pinA1.addReportListener(
13. //inner class implements event listener
14.                         new TriggerEventListener(){
15.                                 public void triggerEvent() {
16.                                         //poll status byte for value of its bit 5
17.                                         if ( (LM628.rdstatusLM628 & 32) == 32) {
18.                                                 monitor.sendEmergencyStop();
19.                                 }
20.                         }
21.                 }  );
22.         } //end of constructor
23. }
```

In the above class, when an interrupt is received, the LM628 status register is polled for its bit values. Alternatively, the status- byte can be cyclically polled as shown below. This routine waits for a trajectory-end bit before executing the next command.

```
1.  public static void wait_trajectoryEnd_bit() {
2.     do {
3.       check_busy_bit();
4.     while ( ((read_Status_Byte() & 4) != 4 //poll status byte for value of its bit 3
5.             }
```

5.5 Conclusion

A detailed description of the IMC hardware design has been presented in this chapter. The design consists of an embedded Java microcontroller, JStick, and a motion controller board specifically designed for this project. The motion controller board features a dedicated precision motion controller chip (LM628), I/O logic to communicate with the microcontroller, a DAC and encoder receiver logic, digital I/O, a power converter, and a clock. Six of these boards were manufactured for this research. The next chapter presents the IMC communication architecture.

6. THE IMC COMMUNICATION ARCHITECTURE

6.1 Introduction

The integrity of a reconfigurable distributed control system depends heavily on the flexibility of its communication architecture (Feng-Li et al, 2000). Even though the traditional centralized point-to-point communication architecture has proven advantages such as reliable hard real-time support, its communication infrastructure is quite inflexible. This deduction is clarified by using a microcontroller bus such as VME as an illustration: Firstly, there is the issue of the central locus of control – a computer crash brings down the entire system. Moreover downtime is likely to be aggravated by the time it takes to single out the fault. In contrast, a distributed system built on modular units is naturally molded into fault containment regions. Even if a fault pervades the entire system, it is relatively easier to diagnose and replace or fix modules. Secondly, scaling up a centralized system is met with capacity, cost and hardware problems on the computing, communication and geographic (location) domains. On the other hand, networked control systems could make reconfiguration such as scalability as easy as plug-and-play.

However, like the advent of any typical ideology that promises a holistic solution, distributed control is met by quite a colossal challenge to surmount. Since data exchange occurs between processes in different processors spatially separated, there is the need for a communication service that meets the demands of synchrony, data rate, latency and reliability. While it is easy to instantaneously sample several sensors in a centralized system, clock drifts or process execution deviating trends will have to be accounted for either dynamically or statically in a distributed system. Yet another major problem is the mechanism of the network itself: Information must be deftly delivered from producers to recipients (collision avoidance). Some data *must* be delivered reliably, while others *must* be delivered deterministically. The network protocol may also have to discriminate between messages of different priorities; for example, an emergency message such as "shutdown controllers" should preempt lower-priority messages. On a centralized system,

the latter case is as simple as asserting a high-priority interrupt line connected to slave controllers. The last problem worth mentioning is the issue of interoperability; simply stated, controllers on different networks may not be able to talk to each other without a mediator.

These myriads of problems have churned out a number of research works and interesting solutions which is gradually narrowing the performance gap between the two extreme configuration architectures. In fact, network architecture design in distributed control systems is becoming an artistry of sorts, which is resulting in a plethora of standards and paradigms (Pedreiras and Almeida, 2000; Schickhuber and McCarthy, 1997). Four main research and development (R&D) streams in distributed control have been identified. One stream deals with the aspects of the communication mechanism, such as protocols and physical infrastructure (Kopetz, 1997; Pedreiras and Almeida, 2000), and paradigms such as Time-Triggered, Event-Triggered and Cyclic communication has led to fieldbuses such as TTA, CAN, Profibus, and LonWorks – to mention a few! The next stream closely associated with the first, focuses on clock synchronization of networked controllers (Kopetz 2004; Lönn 1999), while the third stream emphasizes control algorithms that curtail network uncertainties (delays and jitter) (Wittenmark, et al., 1995; Goktas, 2000; Nilsson, 1998). The last stream addresses reconfigurable control solutions (Lian, et al, 2000; Atta-Konadu, et al, 2005) sometimes with fault tolerance (Kopetz, 1997; Benitez-Perez and Garcia-Nocetti, 2005). The architecture presented in this chapter is an integrated (hybrid) approach that incorporates the strengths of distributed and centralized architectures. Moreover, it promises a cost-effective and inter-operable solution by employing the most ubiquitous and flexible network system – Ethernet. As will be discussed in the following, there is some platitude about Ethernet not being real-time. However, an approach is presented which provides a real-time environment harnessed by the robustness of Ethernet

6.2 Requirements for Real-Time Communication

In practice, real-time networks require high efficiency, deterministic latency, operational robustness, configuration flexibility, and low cost per node. Because cost constrains the network bandwidth available to many applications, protocol efficiency is very important. Most real-time systems are characterized by predominance of short,

periodic messages. An obvious optimization is to reduce overhead bits used for message packaging and routing. The next issue is to reduce media access overhead. This may be accomplished by minimizing the network bandwidth consumed by arbitration (e.g., passing a token or resolving collision conflict). Since worst-case behavior is usually important, efficiency should be evaluated both for light traffic as well as heavy traffic. Determinacy, or the ability to calculate worst-case response time is needed to meet the real-time constraints. A prioritization capability is usually included in systems to improve determinacy of messages for time-critical tasks. Priorities can be assigned by node number or by message type. Furthermore, protocols should support local or global priority mechanisms. In local prioritization, each node is given a turn at the network in sequence and transmits its highest priority queued message (thus potentially forcing a very high priority message to wait for other nodes to transmit). In global prioritization the highest priority message in the global system is always transmitted first. This feature is highly desirable for many safety critical applications.

A protocol is robust if it can quickly detect and recover from errors (e.g., duplicate or lost tokens), added nodes, and deleted nodes. Simple protocols require less hardware and software resources and are therefore likely to be less costly. For cost-sensitive high-volume applications, these protocols are good choices. However, for scalable applications, more advanced protocols provide stronger framework.

One of the main problems in real-time communication is the scheduling of messages over the network so that messages' time constraints are met. The sort of scheduling that can (or must) be used depends on the network topology: multiple-access (e.g. shared broadcast bus or ring) or point-to-point (e.g. mesh network). An issue of utmost importance for the real-time scheduling of messages on multiple-access networks is the Medium Access Control (MAC). In real-time networks, access control protocols play a fundamental role in the timeliness of the communication system since they establish the order by which communicating nodes access the transmission medium. Therefore, they directly influence the response time of the communication system to the requests issued by the nodes. Such protocols must ensure that all nodes have the right to access the bus within a bounded time window. Otherwise, the ability of the communication system to transfer information subject to time constraints is lost. When it comes to guaranteeing a

predictable timing behavior, as required in real-time networks, the problem is not confined to the MAC protocol. In fact, such timing predictability must be a property of all protocols and services in all layers (Thomesse, 1998). When this is the case, it is possible to perform a schedulability analysis over the set of communication requirements and thus, obtain some level of guarantee concerning the timely behavior of the communication system.

6.2.1 Environment State Capturing Strategy

The computing system in a real-time network must be aware of the state of the environment at any instant. This is achieved by maintaining a data structure within the computing system that reflects the environment state. Since the computing system is distributed, the real-time database is likewise distributed among the nodes of the system. In such databases all the items have expiration times after which they are no longer valid. This property known as *temporal accuracy* is normally implemented as an Event-Triggered (ET) or Time-Triggered (TT) mechanism. In the ET approach, the computer system is alerted of any significant change in the environment state (external event) or in the controller internal state (internal event), such as a timer interrupt. Consequently, the computer system triggers the appropriate actions. If there are no events, the ET mechanism remains in an 'idle' or wait state. ET systems are susceptible to a phenomenon called *event showers*, where a node is overwhelmed by events such that not all events can be serviced; hence a scheduler has to deal with simultaneous arrival of many events, some with different priorities and deadlines. An overwhelmed scheduler might result in missed deadlines. Figure 6.1 shows a typical timing issue in an ET system. P_i is processor node i while the rectangles are events. The flag indicates the detection of events.

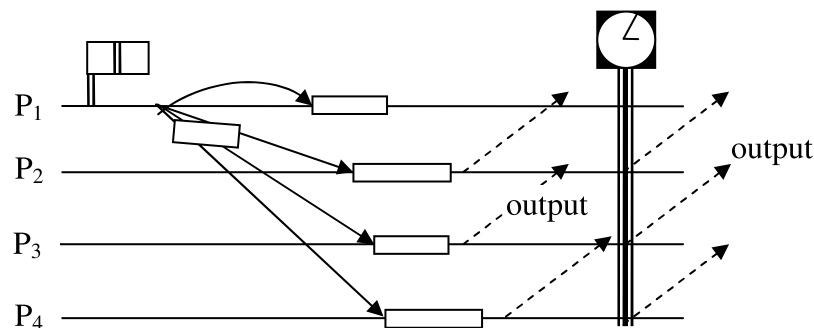


Figure 6.1: Timing in an Event Triggered System

Events are processed and dispatched as they arrive. However, the system architecture can create rules to superimpose TT behavior on top of an event communication or processing system (Verissimo and Rodrigues, 2001). In the *TT* approach, processes are started at predefined time slots, normally periodically. The environment or nodes are scanned periodically at rates pre-calculated to take into account the environment dynamics. Unlike the event-triggered approach, even when there are no state changes, actions within the computer system are continuously triggered. The advantage of the TT is that since activation instants for all actions are predefined (Fig. 6.2), it is possible to control the level of contention among actions by appropriately controlling the relative phasing of those instants.

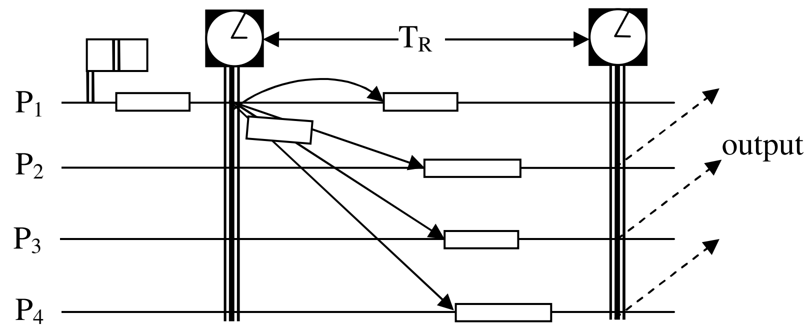


Figure 6.2: Timing in a Time-Triggered System

ET is more resource-efficient when changes in the environment state are sporadic. However, its temporal performance depends on the number of events that might arrive simultaneously at the computer system. The time-triggered approach has a more stable temporal behavior due to the aprioristic knowledge of the actions' activation instants. Hence, event-triggered approach is normally associated with dynamic scheduling, while the TT approach is usually associated with static scheduling.

6.2.2 Co-operation Models

Another important property of a communication system is the manner of interaction between *stakeholders*. There are two distinct co-operation models namely, client-server (CS) and the producer-consumer (PC) models.

In the client-server model a request from a client to the server prompts a response from the server at a later time. There are different variants of the CS model. In the one-to-many communication (clients-server) model, requests from the multiple clients are serialized and processed by the server one at a time. This situation may give rise to spatial incoherence in the real-time database (temporary inaccurate real-time images) since two simultaneous requests for the same entity in the same server might obtain different values. Another variant of the CS model is "many servers to one client", i.e., client-servers model. The simultaneous access by this client to the different servers has to be carried out sequentially and may therefore yield temporal incoherence in the real-time database. Lastly, in clients-servers model (many-to-many), all client requests need to be serialized with regard to several servers. In all variations of the client-server model, if time is to be accounted for as in real-time applications, it is necessary to bound the maximum response delay by a server to any client request. The CS model is very appropriate for supporting acknowledged data transfers.

The producer-consumers model works on a different principle: A producer node with data that might be needed by other nodes makes such data available on the network. The consumer nodes identify data relevant to them and read it from the network. Unlike the CS model, there is no explicit consumer request and the producer generally starts transactions. Since all consumer nodes have simultaneous access to data on the network, the model implicitly supports one-to-many communication with spatial consistency of the real-time database. However, where they are multiple producers, the respective transactions must be serialized, which could cause temporal coherence problems. The producers-distributor-consumers (PDC) model (Thomesse, 1993) is a solution to this problem. This model is a combination of the producers-consumers model and a master-slave architecture. All transactions are centrally managed by the distributor (master node), which facilitates the respective scheduling in order to meet the temporal constraints required to assure both the temporal accuracy and coherence of the real-time database.

6.2.3 Composability

Arguably, the most important property of a real-time system architecture is its composablility (Kopetz, 1997). "An architecture is composable with respect to a specified property if the system integration will not invalidate such property once the property has

been established at the subsystem level. Examples of such properties are timeliness or testability." (Kopetz, 1997). Kopetz (1997) outlines two factors that establish composability. The first is referred to as "Temporal Encapsulation of the Nodes": The communication system should wrap a temporal firewall around its host computer, forbidding the flow of control signals across the communication network interface (CNI). It is stated categorically (Kopetz, 1997) that such an autonomous communication system can be implemented and validated independently of the application software. Moreover, communication timing properties can be validated in isolation. The counter argument is that if flow-control[1] can be characterized, the need for autonomy which requires extra hardware will be rather questionable. Another requirement for real-time communication is the need for flexibility. For example, a scalable architecture allows functional and/or physical changes to the system during its lifetime without any predefined upper limit to such changes or elaborate software modification. The communication system must be robust, providing predictable and dependable service (Kopetz, 1997). The system should be capable of rectifying errors quite seamlessly, and if this is not possible, all participating communicators must be informed urgently. Lastly, there should be interoperability between equipment and the communication system and also with potential networks outside the immediate sphere of communication. Thomesse (1998) refers to several causes for non-interoperability such as unavailability of services, different options in protocol implementation, time behavior incompatibilities or lack of resources. The issue of obsolesce is also noted in the IMC communication architecture design. It is desired to anticipate new network apparatus which are backward-compatible with older ones.

6.3 Real Time Network Applications

Due to the absence of shared memory, communication in distributed systems is based on exchanging messages between communicating elements. Therefore, there is the need for a priori communication agreement at all levels of communication, i.e. from low level bit formats, to data semantics, error checks, etc. This led to the development of the Open System Interconnection Reference Model (OSI) (Day and Zimmerman, 1983) by the ISO (International Standards Organization). The OSI model as it is popularly called,

[1] Flow-control is the control of the speed of information flow between a sender and a receiver in such a manner that the receiver can keep up with the sender

clearly identifies the various levels involved in network communication, gives them standard names, and their respective functionalities (Fig. 6.3).

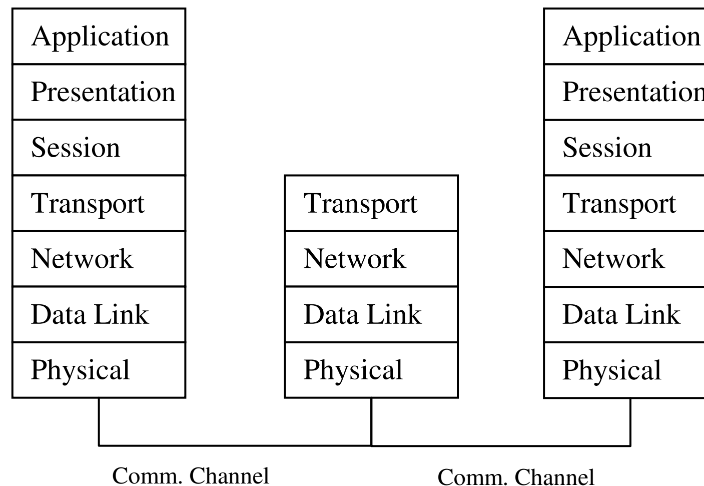


Figure 6.3: The OSI Protocol Stack

Communication systems based on the full OSI model are highly flexible and interoperable. However, the layered architecture imposes considerable processing and communication overhead. Consequently, protocols that were developed for the OSI model were never widely used (Tanenbaum and van Steen, 2002). Rather, protocols developed for applications are based on the bottom few layers and the application layer. Ethernet for example, is based on the bottom four layers and the application layer, while control networks, typically fieldbuses are based on the bottom three layers and the application layer. The latter results in *lean* communication overheads for deterministic response time but at the expense of interoperability and flexibility. The following sections discuss the protocols used in fieldbuses and Ethernet.

6.3.1 Physical Layer

The physical layer ensures the transmission of bits. The voltage levels for low (0) and high (1), bit transmission rate, and whether transmission is bidirectional are key issues in the physical layer. Other issues addressed in this layer include the interconnection topology, the physical medium, the maximum bus length, the maximum number of nodes

that can be connected to the bus, the possibility to feed power to the nodes through the bus, and immunity to EMI (Electromagnetic Interference). Most fieldbuses use a bus topology while Ethernet can accommodate other topologies such as the star-topology. The two commonly used physical media are electric cabling and optical fibers. Industries are also fast embracing wireless networks, particularly Bluetooth and wireless Ethernet (IEEE 802). Wireless networks are particularly suitable for mobile equipment and in situations where devices are spread over large distances in large plants. In many situations where cabling is used there is a limitation to the number of nodes that can be connected to the fieldbus since each connected node adds an extra capacitance to the bus line, which tends to increase the propagation delay of control signals and data.

6.3.2 Data Link Layer

The data link layer provides services and protocols required to ensure the correct transmission of data. The salient entities in this layer are the mechanism governing access to the shared communication media, identification of the destination node (or nodes), and mechanisms to assure the correct transfer of information. The first entity is called *MAC* (Medium Access Control). The second is called *Addressing*, while the third entity is called *LLC* (Logical Link Control).

6.3.2.1 Medium Access Control (MAC)

The MAC sub-layer is the workhorse for message scheduling on multiple-access networks. MAC protocols influence the response time of the communication system since they establish the order by which communicating nodes access the transmission medium. In real-time communication, protocols guarantee that all nodes have the right to access the bus within a bounded time window. One common classification of MAC protocols is whether they impose controlled access (e.g., centralized arbitration, token-passing, Time-Division-Multiple-Access (TDMA)) or uncontrolled access. In controlled access, message collisions are avoided by some control mechanism or signal. As an example, token-passing is used in Profibus, where a control message called a token circulates the network such that whoever has the token is allowed to transmit. Some protocols such as the FIP protocol depend on a master node to arbitrate communications while others regulate transmission by using time-slices (e.g. ARINC 629) or progression of real-time (e.g.

TDMA). In uncontrolled access protocols such as CSMA (Carrier-Sense-Multiple-Access), there is no external control signal; instead arbitration is performed based on the bus status and on local information. Collisions may occur since it is possible that several nodes detect silence on the bus at the same time and start transmitting almost simultaneously. The manner in which collisions are handled defines the nature or type of CSMA protocol. The arbitration mechanism is completely decentralized and independent. This implies that there is no centralized information about the present system configuration, and arbitration is carried out the same way regardless of the configuration. This lends CSMA-based systems high flexibility, for example, making it possible from a functional perspective to connect or disconnect nodes during normal on-line operation. However, the arbitration mechanism is a challenge in real-time message delivery. For instance, in the CSMA-CD (Collision Detection) protocol which is used in Ethernet, nodes involved in a collision hold back from transmission and retry after a *random* time interval. Obviously, this phenomenon becomes critical and highly non-deterministic during heavy traffic. A solution to this is to reduce or totally eliminate the probability of collisions. This will be discussed in the course of this chapter. Some CSMA protocols are designed to be deterministic, for example CSMA- CA (Collision Avoidance). In one such protocol, a set of synchronized timers at each node with predetermined values is used to guarantee that only one node transmits at any given time. An enhanced version of this scheme is used in the IEEE 802.11 wireless protocol. Another variant of CSMA-CA is used in the popular CAN (Controller Area Network) field bus where an 11-bit message identifier field is used for arbitration. In this case, the arbitration mechanism assumes that a dominant and a recessive state exist on the communication bus such that the dominant state can overwrite the recessive state (Kopetz, 1997). Assume that ‘0’ and ‘1’ are coded into the dominant and recessive states respectively. Whenever a node wishes to send a message, it puts the first bit of the message identifier on the network. In the event of a conflict, the node with a ‘0’ in its first identifier wins the right to transmit, and the other node must back off. This arbitration continues for all eleven bits in the identifier. This arbitration however limits CAN to a maximum bit rate of 1 Mbit/s on a 40 m bus. Other serial-bus protocols are emerging that guarantee bandwidth for high speed deterministic communication. FireWire (IEEE 1394) for example is capable of interleaving asynchronous and isochronous

communication at data rates likely to exceed 400 Mbps in the near future (Steinberg and Birk, 2000). This network targets home use but is gradually being fused into industrial applications requiring substantial bandwidths.

6.3.2.2 Addressing

The Data Link Layer is also responsible for identifying nodes, i.e. the origin and destination(s) of messages. There are two different ways of addressing the destination of a data transaction: directly addressing the node where the receiving application process resides or indirect addressing where an identification is placed on the data to be transmitted. The former is used in many field buses, typically in Master-Slave configurations. Destination nodes can be identified either by their physical addresses, i.e., MAC addresses, or by their logical addresses, i.e., network addresses. Group addressing is a very important property in industrial automation systems where for example the output generated by a given controller may need to be shared among several field devices. Hence many field buses and Ethernet support one-to-many communication, or multicast. Indirect addressing is typically found in protocols that support the producer-consumer communication model, where a producing node initiates a transaction to transmit a given data entity. The consumers identify the data entity of interest to them and copy them to their local buffers to be used by their respective application processes. This addressing scheme is found in CAN and WorldFIP.

6.3.2.3 Logical link control

The last component of the Data Link Layer is the Logical Link Control (LLC). This sub-layer ensures the correct transfer of information among communicating nodes by properly framing data to be transmitted, implementing error detection and correction, establishing data-link connections between nodes, and coordinating communication acknowledgements. Most real-time networks provide communication with immediate acknowledgement for real-time data. For periodic data transmission, however, communication is unacknowledged since it requires a considerable amount of bus bandwidth. Multicast communication is also typically unacknowledged. Regarding connection, most fieldbuses provide connectionless communication. Ethernet's LLC provides two types of data link control operations; LLC1 for connectionless and LLC2 for connection-oriented. With connection-oriented services, the sender and receiver first

explicitly establish a connection, and possibly negotiate the protocol they will use. The connection is terminated at the end of the transaction.

6.3.3 Network Layer

The Network layer is relevant in Wide Area Networks (WAN) for routing messages from a sender through a maze of networks to the receiver. In LAN systems, usually the sender does not need to know the geographic location of the receiver; once the message is delivered on the network, the receiver takes it off. The connectionless IP (Internet Protocol) is the most widely used network protocol. The IPv6 (IP version 6) includes many improvements over IPv4 (IP version 4) including stateless address autoconfiguration (Thomson and Narten, 1998).

6.3.4 Transport Layer Protocols

The transport protocol forms the last part of the basic Ethernet protocol stack. The job of the transport layer is to provide a reliable connection. When data is received from the application layer, the transport layer breaks it into packets small enough for transmission, assigns a sequence number to each, and sends them all. The transport layer then monitors which one has been sent, received, how many more the receiver can accept, which should be retransmitted, etc. Reliable or connection-oriented transport connections can be stacked on a connection-oriented or connectionless service. In the former situation, all the packets arrive in the same sequence that they were sent, but in the later case, it is possible for one packet to take a different route and arrive ahead of the packet sent before it. It is then the responsibility of the transport layer protocol to reassemble the packets in the right order. The Internet transport protocol is called the TCP (Transmission Control Protocol). The combination of TCP/IP is now the de-facto standard for network communication (Tanenbaum and van Steen, 2002). The Internet protocol suite also offers a connectionless protocol called UDP (Universal Datagram Protocol), which is basically IP with minimal additions.

6.3.5 Application Layer

Application layer services normally follow a certain co-operation model, e.g. Client-Server (CS) and Producer-Consumer (PC), which describes how data is exchanged between the communicating peers. Generally, the CS model is used for one-to-one

connection-oriented co-operation. This is very useful for non-repetitive asynchronous services such as remote control of tasks execution. The PC co-operation model is more suitable for periodic task with high data rates such as closed loop control. These services are typically subject to tight temporal constraints and hence need the support of connectionless services. The application layer is also used for task scheduling, since user tasks are located at this layer.

6.4 Automatic Configuration

In concept, each layer of the protocol stack described above could be automatically configured. Flexibility of configuration makes simple technology work more predictably and easier to deploy. The transport and data layers rarely require configuration in Internet hosts, unlike the application and network layers which nearly always require configuration for devices to be able to communicate. Historically, this has been a task for experienced network administrators, however emerging networking protocols are making configuration changes less difficult. For typical configurations, hosts that are permanently attached to a network are assigned static network configurations by administrators, while other hosts are assigned dynamic configurations. All necessary parameters are assigned to the host by a network configuration service, which also requires configuration. Many situations call for ad hoc reconfigurations or fault tolerance which can be impractical or impossible (Guttman, 2001). For this reason, automatic configuration protocols are becoming extremely valuable in the dynamic world of networking. There are two main strategies to bridging the gap between configuration and automatic operation (Guttman, 2001). The first approach requires transitions between local (automatic) and global (dynamic) configuration. Hosts provide local configuration for as long as there is no global configuration. A typical example is the network interface autoconfiguration protocol adopted for Apple and Microsoft operating systems. A host uses this protocol to select an unassigned IP address from a reserved range. The host then uses a Dynamic Host Configuration Protocol (DHCP) to request for IP configuration parameters (global) from the network. If a DHCP server responds (usually after some retries) and offers configuration parameters, these replace the local ones on the host. In a client-server system, this works very well especially on the client side but can become problematic when server configuration changes. Servers with dynamically changing IP parameters can

only be located by a dynamic discovery protocol. When a server configures via a DHCP, it cannot communicate with clients that have not been configured (for e.g., if the DHCP is no longer available). Likewise, configured clients cannot locate a server that has not been configured. Lastly and very pertinent in this research, some simple embedded devices support only local IP configuration and would therefore be unable to locate hosts configured with DHCP. Such problems have compelled the need for automatic configuration protocols, and have resulted in protocols such as AppleTalk, IPX and NetBIOS/SMB, which attempt to address some of these needs (Guttman, 2001). The section below discusses the Zeroconf protocol which is one of the most comprehensive solutions available.

6.4.1 The Zeroconf Protocol

The Zero Configuration Networking (Zeroconf) workgroup pioneered by Apple has defined requirements for four zero configuration network protocol areas: IP addresses autoconfiguration, name resolving without a DNS (Domain Name Service) server, decentralized service discovery, and multicast address allocation (Passmore, 2002). The last item has not been standardized yet, but the Zeroconf suite offers one of the most comprehensive solutions to avoiding dependency on infrastructure such as DHCP and DNS servers, and expert knowledge. The sections following discuss the Zeroconf specifications.

6.4.1.1 IP addresses Autoconfiguration

The Zeroconf specification for IP address autoconfiguration is different for IPv4 and IPv6. For IPv4, computers pick a random link-local address in the 169.254.0.0/16 range, and send out an ARP request to check if another host is using it. If so, they select another IP address and repeat the process. By design, IPv6 supports dynamic allocation of addresses.

6.4.1.2 Name Resolution

Zeroconf specifies a multicast DNS (mDNS) quite similar to LLMNR (Link-local Multicast Name Resolution) promoted by Microsoft. The latter however, has no implementations yet. For both protocols, a host does not need a DNS server to find the name of another host. Instead the host sends its DNS request to a unique IP multicast

address (224.0.0.251 for mDNS) on the local subnet where hosts listen and respond. Each hosts picks a hostname in the .local domain (mDNS), and publishes this on the IP multicast address. Collisions are prevented by a conflict resolution mechanism.

6.4.1.3 Service Discovery

There are two classes of service discovery protocols, namely, high-level (application or technology specific) ones and low-level generic ones. High-level protocols include Jini for Java objects, Salutation which relies on central servers, Service Discovery Protocol (SDP) for Bluetooth, which is based on the former and UDDI for web-services. Generic ones include Simple Service Discovery Protocol (SSDP) used in Universal plug-and-play (UPnP), Service Location Protocol (SLP), and DNS-SD (DNS Service Discovery) specified by Zeroconf. Generally, SSDP is regarded to be more complex than DNS-SD, and SLP has not been widely embraced. On the technical side, DNS-SD typically makes defunct the need for a central infrastructure such as a DNS or directory server. Each host offering a network service creates a DNS SRV resource record that it stores in its local mini-DNS server. Other hosts look up the service by broadcasting a DNS service discovery query. All hosts that offer the requested service then respond with their names and IP addresses (Passmore, 2002).

6.4.1.4 Zeroconf Implementations

Zeroconf specification is now adopted and implemented by many network device manufacturers. Currently, many network printers and network storage devices implement some aspect of Zeroconf-compatible networking. The most widely adopted Zeroconf solution is *Bonjour* from Apple Computer, which uses a combination of IP address autoconfiguration, mDNS and DNS-SD. Many implementations nonetheless do not implement the full specification. However, mDNS and DNS-SD are often implemented together. For example the Java implementation of Zeroconf – JmDNS, provides only mDNS and DNS-SD services. JmDNS has been modified in (Atta-Konadu, et al., 2005) for embedded Java devices.

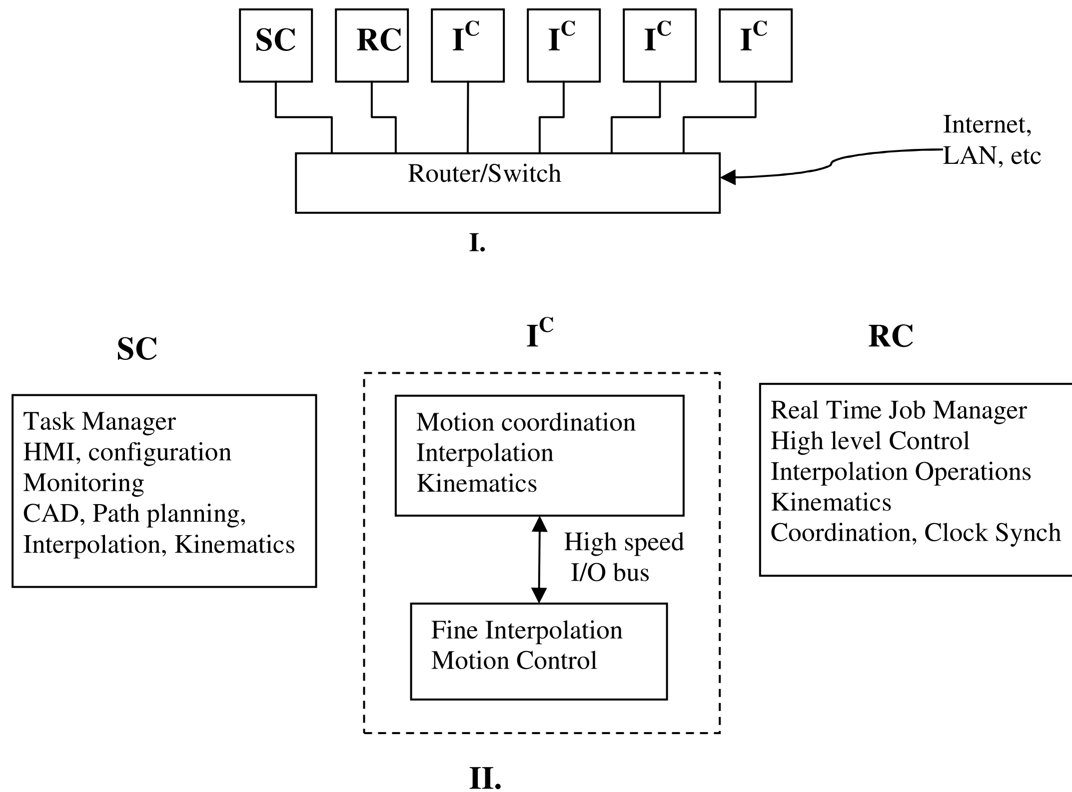
6.5 The IMC Communication Architecture

The IMC communication architecture is quite a unique concept that combines different aspects of orthogonal paradigms such as distributed and centralized systems,

client-server and producer co-operation models. The peculiarity of the architecture is reinforced by its Ethernet communication backbone – as mentioned earlier, Ethernet is not popular in real-time applications. Nonetheless, the architecture is built on the premise of using some of the most basic cost-effective commercial of-the-shelf (COTS) components. Ethernet for example is the most ubiquitous, proven and reliable of all networks with well known hardware and software characteristics. Very few networks are as cost-effective and simple to implement. The architecture addresses the issue of realizing real-timeliness. Some attempts have been made to realize a real-time Ethernet communication system. The fundamental method is to reduce message collisions on the network. The IMC architecture imposes rigorous traffic patterns and a message scheduling mechanism to create collision-free zones. Moreover, the JmDNS-CLDC protocol is fused into the architecture to create a reconfigurable paradigm. The sections following elaborate on the architecture design.

6.5.1 Communication and Computing Elements

The message collision crisis of Ethernet is curtailed by the use of switching technology. As shown in Fig. 6.4 (I), all communication nodes are connected to a 10-100 Mbps router/switch, forming a star-topology as opposed to the traditional shared bus or linear topology. The result is there is only one collision domain per port – or dedicated bandwidth segment. Moreover, all communication ports are full-duplex, hence inbound and outbound messages are kept on separate channels.



I. Nodes connected to a Router/Switch;

II. Functionalities of the SC (System Coordinator), RC (Real-time Coordinator), and I^C (IMC Controller).

Figure 6.4: Communication and Computing Elements:

The switch operates with wire-speed and is non-blocking. Wire-speed implies that all ports of the switch can simultaneously transmit and receive at their full rates. Switching technology is becoming very attractive in industrial systems for process control. Nonetheless, some further work on the protocol level is required to meet the stringent requirement for real-time communication. Typically, congestion can occur in a switch when several ports forward their traffic to the same output port while the total input traffic is greater than the output bandwidth. In this case the source bit rate should be slowed down to avoid buffer overflow within the switch (Wang et al., 2001). Another potential for congestion is when the output port is connected to a shared segment and the available bandwidth left by the traffic of the segment is smaller than the input traffic (Wang et al., 2001). The section following describes how the IMC architecture is guarded against the

above phenomena. The computing elements shown in Fig. 6.4 (II) reside in the System Coordinator (SC), Real-time Coordinator (RC) or the IMC controllers (I^C). A description of their functionalities is provided in Chapter 3.

6.5.2 Communication Flow and Control

The architecture divides communication into two zones as shown in Fig. 6.5. Three main communication protocols are established between the communicating elements: A TCP/IP connection is maintained between the SC and IMC controllers at all times for position update information. An unanticipated broken connection is assumed to be a fault on the IMC node. The second persistent connection is a multicast connection between all nodes, i.e., one multicast domain, and is used for simultaneous delivery of messages. The third connection is a datagram connection between the SC and all other nodes. This is used for control signals, and configuration information. For clarity, this connection is lumped together with the multicast connection in Fig 6.5, since both protocols are based on UDP/IP.

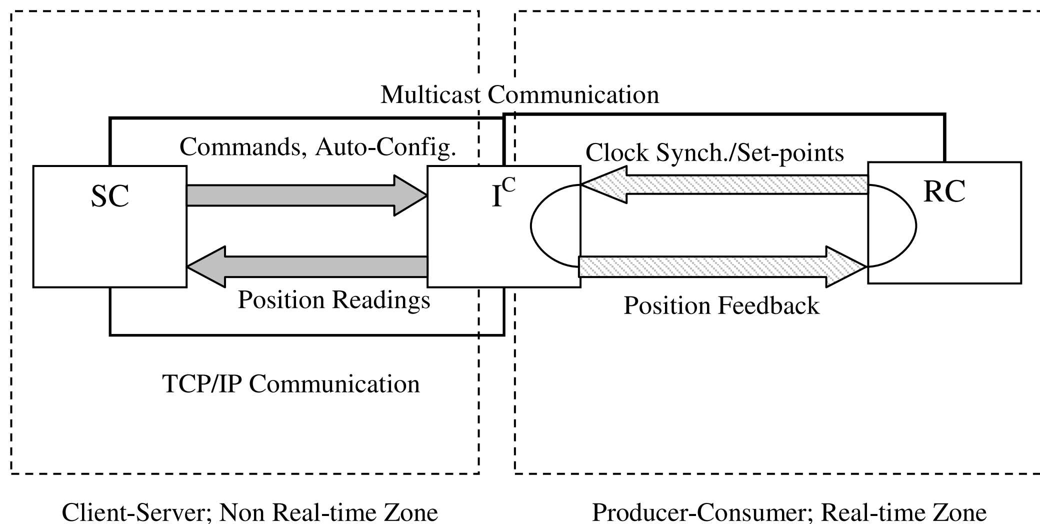


Figure 6.5: Communication Flow

Communication between the RC and I^C is designed in a producer-consumer fashion, such that messages are not explicitly addressed to receivers. At startup, the communication elements use an auto configuration protocol (to be discussed later) to

identify themselves and their services. At this point, the IMC nodes are given unique identity numbers. The configuration in Fig 6.5 is logically set-up when the user selects *coordinated-motion* from one of the four motion modes provided by the architecture (see Chapter 7 for details), and motion commands typically in NC code format are sent by the SC to the RC. The SC also multicasts task information like the number of axes and interpolation period to all nodes and measures the round-trip time to compensate for delay. As soon as the interpolation period T is received, the RC sets up a real-time periodic thread to run its interpolator. If position feedback is required for high level control, a second period thread is established according to the update rate required, to receive sensed positions from the IMC nodes. When the SC issues a *run* command to the RC, it attaches its logical clock value to a *start-of-motion* message and multicasts this to all IMC nodes, which in turn use this value to adjust their respective timers (the timers clock the motion controllers). Subsequently, each periodic data received from the RC connotes the global time and is therefore the basis for timer synchronization. The next data to be streamed at the next period is delineated by the unique node numbers and packed into a datagram packet. When data arrives at each IMC node (simultaneously), its timer is synchronized if deviation is off limits, and the relevant data addressed to that node (by the identifier) is extracted from the datagram packet for its motion controller. Figure 6.6 is a skeletal sequence diagram of the interactions between a single IMC (consumer) and the RC (producer). Data transmissions denoted by the horizontal arrows are kept atomic (no processes in-between) unless preempted by the system coordinator.

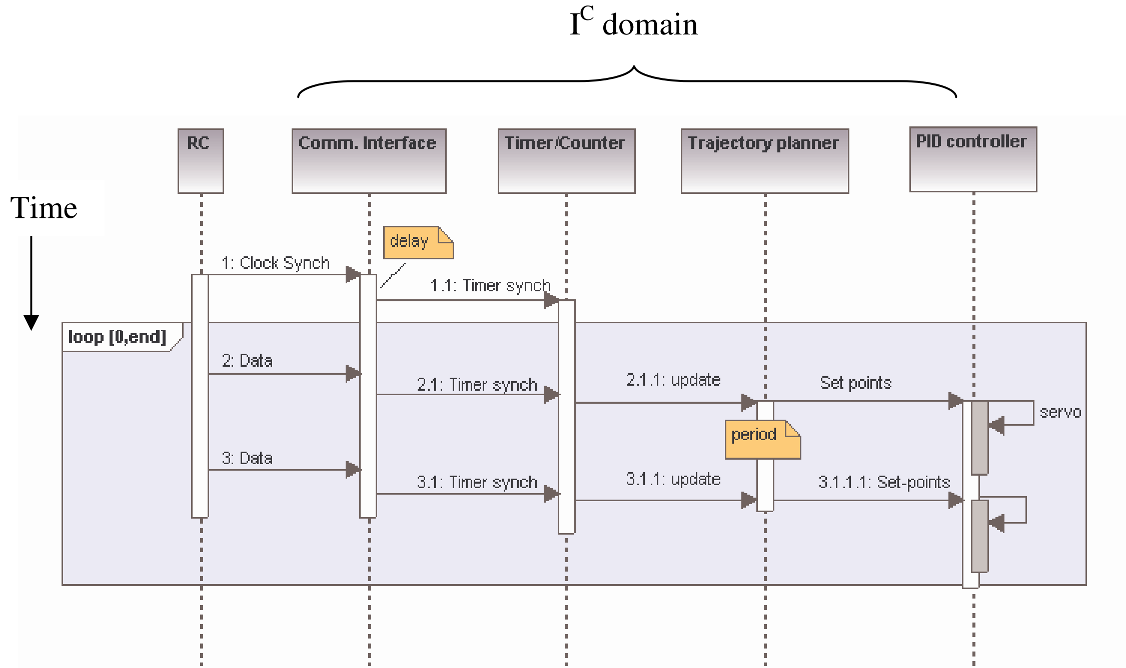


Figure 6.6: Produce-Consumer Co-operation between the RC and IMC Nodes

If the RC requires position feedback, the IMC nodes take turns transmitting their actual positions to the RC, in a TDMA (Time Division with Multiple Access) style, to avoid message collision and congestion on the switch. The protocol for this timing is determined by the identifier assigned to each node. For example if there are N nodes and the RC receiving thread is cycling periodically at T ms, node number n will have its turn to transmit data at an offset of $T \times n$ from the start of the motion transaction and with a period of $T \times N$. There are other variants of the communication architecture to support the other motion modes. For example, in *synchronized-motion-mode*, set-points are streamed directly from the system coordinator to buffers in each IMC node in a typical client-server fashion.

6.5.3 Triggering and Scheduling

In this section, an analysis of message scheduling on the communication architecture is presented. The real-time threads on the IMC processors including that of the real-time coordinator use priority-based cyclic scheduling with pre-emption; thus a Time-triggered (TT) approach is used. Since the schedule is pre-defined or derived prior to

run-time, scheduling is static, as opposed to dynamic scheduling. The communication mechanism employed, i.e., Ethernet, is typically Event-Triggered (ET). However the zero-collision derived from segmenting the network renders the communication mechanism quite like a hybrid of TT and ET. In fact the last barrier to its full qualification as a TT/ET hybrid is that the CNI (communication network interface) on the microcontrollers are not autonomous. A CNI is autonomous[2] when the decision when a message must be sent resides within the sphere of the communication system rather than the host computer (Kopetz, 1997). Otherwise, the timing for control signals to cross over the CNI to the host cannot be exactly characterized. Although building an autonomous CNI is quite simple, this research did not address this issue. Rather, communication sockets are encapsulated in interrupt-priority (high priority) threads to protect them from interference. Both ET and TT communication scenarios will be used in the analysis. Data released from the real-time coordinator, communication and trajectory updates in the trajectory planner share a given period time, T. Interpolation occurs at the beginning of the high-level control period, and trajectory update after a short and constant delay δ. Deviations from the nominal time, i.e., jitter is denoted by J. The worst case execution time C_i is the maximum processing time required by a task or for the message transmission time on the bus. The length of the activity window represents the task response time from start to completion. The activity may be completed at any time after the minimal execution time. In the worst case, task execution may be delayed by release jitter and interference from other tasks. The important control performance metrics are jitter J, control delay δ and variation in control delay $\Delta\delta$.

6.5.3.1 TT Communication with TT Processors

Global time is used to release the trajectory planner (hosted by the motion controller) update task at a time when data generation from the real-time coordinator (RC), communication and local transactions on the IMC are guaranteed to have completed (at offset O_G). Task R is scheduled to complete before the next periodic cycle appears, and tasks E and G at a time after the message has arrived (Fig. 6.7).

---

[2] This approach is used in the Time-triggered TTA architecture for tighter determinism.

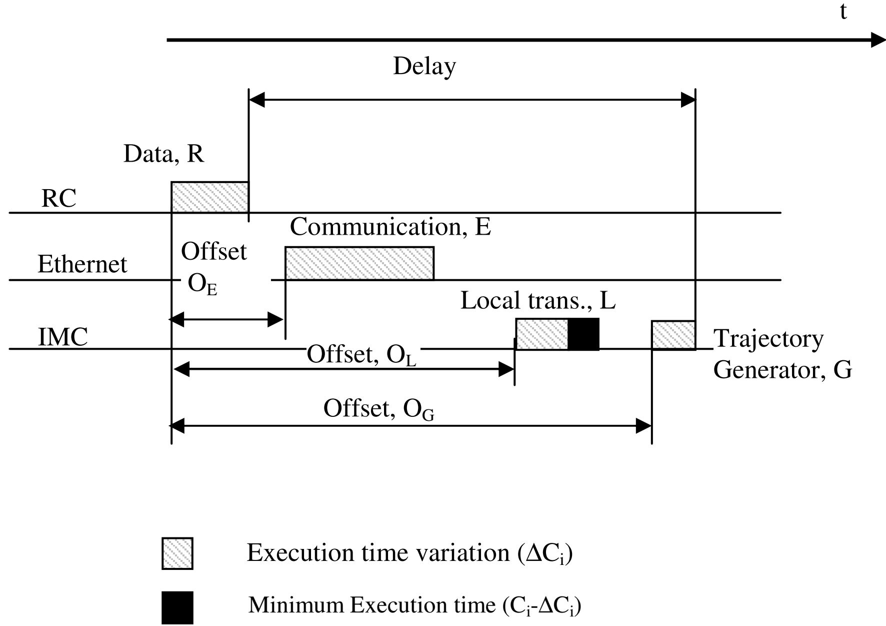


Figure 6.7: TT communication with TT processors

Assuming there are no interference and jitter, task G executes at a fixed offset relative to the data generation task on the RC, and the following hold.

$$\delta_{min} = \delta_{max} = O_G + C_G \quad (6.1)$$

$$O_G \geq C_R + C_E + C_L. \quad (6.2)$$

Supposing the high-level interpolation period is given as 10 ms; for the trajectory generators on the IMC to see this same real-time image, O_G = 10 ms. The network latency is calculated as follows. Given cable length = 3m; cable propagation delay (time for one bit to traverse the cable) = 15 ns, at 2/3 the speed of light in a vacuum; Bandwidth = 10 Mbps. The bit length is defined as the number of bits that can traverse the cable within one propagation delay, and evaluates to 0.15 bits. For a packet size of 100 bytes, the propagation time is therefore 80 μs. Of course, transaction delays within the network interfaces compound this figure. It is much easier to use a round-trip approach to measure the composite delay, i.e., C_E + C_L. A delay of 620 μs is measured for the above data. Hence the periodic execution on the real-time coordinator should be set to 10 ms minus

310 μsec. For consistent latency, data lengths are kept constant in the course of this transaction.

6.5.3.2 *ET Communication with TT Processor*

In this sub-section, we analyze a more conservative scenario where communication is ET, i.e., uncertainties are introduced by jitters in the communication mechanism. Figure 6.8 shows the time sequence diagram. Since the response time of the communication is variable, the offsets of tasks *E* and *G* will be larger, but there will not be any variations in data release to the trajectory generator.

$$\delta_{min} = \delta_{max} = O_G + C_G, \tag{6.3}$$

$$O_G \geq C_R + J_E + C_E + C_L. \tag{6.4}$$

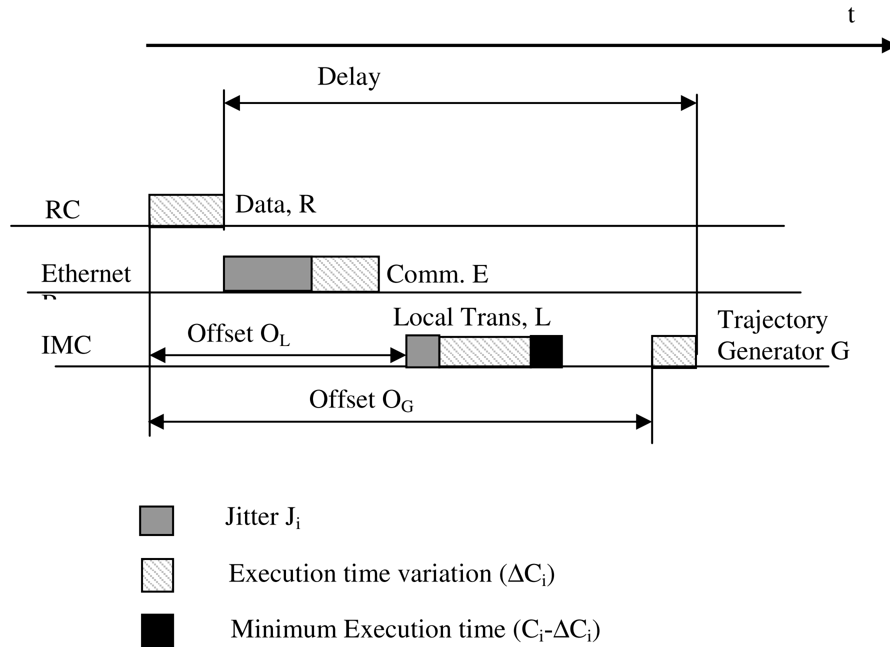


Figure 6.8: ET Communication with TT Processor

The above illustrates that the system architecture can create rules to superimpose TT behavior on top of an event communication if a global time base is observed by participating nodes. For a global time base to exist, the processor clocks need to be synchronized with each other. This topic is dealt with in Chapter 7.

6.5.4 Automatic Configuration

One of the most compelling properties of the architecture is the implementation of JmDNS (JmDNS-CLDC) to enable automatic configuration of the architecture. This implementation also serves as a decentralized watchdog for monitoring activities on the network. The resulting Plug-and-Play (PnP) feature enhances the modular characteristics of the architecture by leveraging seamless additions or removal of network nodes (e.g. an IMC) without the need to configure network protocols. On the IMC controller side, the protocol instantiation is encapsulated in servlets hosted by a min-web server (the implementation details are provided in Chapter 9). Each IMC server keeps a database on its properties such as encoder resolution, PID settings and connected I/O devices. On start-up when hosts join the network, they use the JmDNS-CLDC protocol to register their services and discover themselves. The protocol naming format is [service type, service name, port number, description]. When services are received, network parameters (IP addresses and port numbers) are extracted to enable the necessary connections described in Section 5.5.2 to be established. Figure 6.9 shows a typical interaction between two nodes while Table 6.1 shows some of the services registered and discovered on the network.

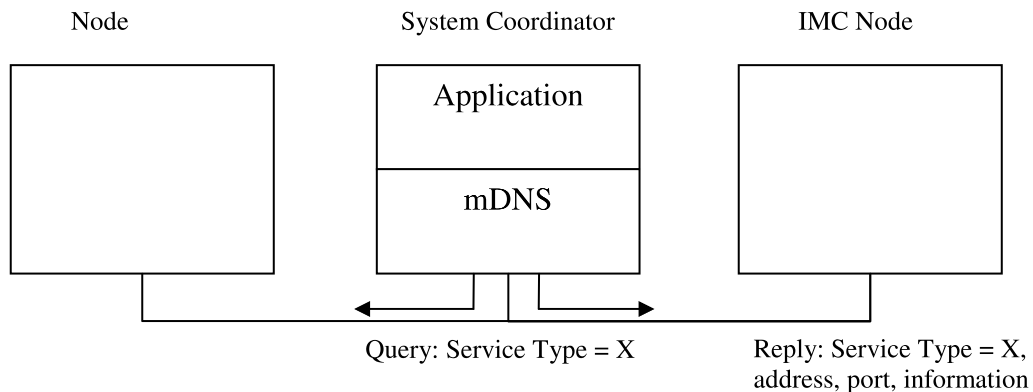


Figure 6.9: Multicast DNS Query

Table 6.1: Typical JmDNS Services on the IMC Architecture

Service Name	Service Type	Port	Description
Axis 1 Controller	_dserver._udp.local.	3000	Datagram Server receives trajectory data
Axis 1 Controller	_controller._tcp.local.	2000	TCP socket connection for streaming encoder readings
Axis 1 Controller	_http._tcp.local.	80	Web service
Multicast	_mcast._udp.local.	4000	Multicast service for receiving synchronizing signals during multi axes coordination

The advantage of the JmDNS-CLDC protocol is that evidently, the loose-coupling between nodes leverages configuration changes and code migration from one platform to another since services are outsourced on the network. Binding or tight-coupling which is essential for real-time control is established as hosts mesh their services and demands on the network.

6.6 Conclusion

This chapter began with a discussion on the requirements for real-time communication, a review of fieldbus and Ethernet protocols, and automatic configuration strategies. Key features inferred from this discussion led to the design of the IMC communication architecture. The following are the highlights of the architecture:

1. An automatic configuration protocol, JmDNS-CLDC running on each node enables nodes to automatically discover themselves and register their services. The protocol also serves as a watchdog.
2. A switched-Ethernet is used to segment the network and create one collision domain per switch port.
3. Communication flow is separated into two zones; one zone implements a producer-consumer relationship between the real-time coordinator and the IMC nodes for real-time periodic transactions; the other zone is a client-server co-operation scheme between the system coordinator and the rest of the nodes for sporadic data communication.
4. The real-time coordinator schedules tasks for the IMC nodes in a static cyclic periodic manner (time-triggered) for hard-real time control.

5. Since all real-time nodes are time-triggered (with a global clock), the event-triggered nature of the communication system does not vary delays – i.e., latency is fixed.

At the moment, device control is entirely distributed amongst the IMC nodes. Even though the architecture provides the framework for a high-level controller, this is yet to be implemented. It will certainly be interesting to analyze the effectiveness of the communication mechanism when this is done.

7. CLOCK SYNCHRONIZATION

7.1 Introduction

It is absolutely important for a distributed hard real-time system to have a global or agreed time base. The time base is composed of several clocks – one in each node, which are synchronized at regular intervals. A good clock synchronization architecture should ensure the integrity of timing in the various communication nodes which will otherwise lead to poor control performance. There are three main challenges in the design of algorithms for clock synchronization: Firstly, if there is substantial network transmission jitter, each process cannot have an instantaneous global view of every remote clock value. Secondly, modern-day quartz-driven clocks run at rates that differ from real-time by up to 10^{-6} seconds/seconds. This implies that two clocks could drift apart by 6 msec per minute even if they are started with the same clock value. The last frontier is recognizing and curtailing faulty or failed clocks, i.e., *failed* communication nodes. Typical synchronization algorithms operate on a set of clock readings collected from the other clocks in the system. When all of the clocks have collected instantaneous clock readings from all other clocks, the synchronization algorithm is applied in each node; therefore all clocks are corrected within the synchronization period. If a clock reading is beyond the boundaries set by the algorithm, it is regarded as faulty. Several fault-tolerant clock synchronization methods have been presented in literature. This chapter discusses pertinent clock synchronization issues and presents the clock synchronization design for the IMC architecture, which is based on external multicast communication with high tolerance for low-precision oscillators, i.e. large clock drift. With this method, clocks are synchronized to a master clock periodically and do not need to keep track of the clock readings of other nodes in the system.

7.2 Time

The most common way to represent time in a process is to use a local physical clock consisting of a counter, and an oscillating mechanism – typically quartz. The

oscillating mechanism generates a periodic event called the microtick (Kopetz, 1997) to increment the counter. Since this is a granular process, digitalization errors in measurement are bound to occur. The duration between two consecutive microticks is the clock granularity. Regarding notation, clocks are identified by numbers: If the property of a clock is expressed, it is identified by the clock number as superscript; the microtick or tick number is denoted by a subscript. For instance, microtick *i* of clock *k* is identified by $microtick_i^k$.

7.2.1 Properties of Physical Clocks

Physical Clock Granularity: The granularity *g* of a physical clock *k* is expressed as

$$microtick_{i+1}^k - microtick_i^k = g^k . \qquad (7.1)$$

Reference Clock: The reference clock is assumed to be a unique reference clock *z* with frequency fr^z, which is in perfect harmony with the international standard time. The granularity g^z of such as clock is $1/fr^z$. Assuming that fr^z is very large, the granularity of the clock is infinitesimally small enough for digitalization errors to be disregarded. A clock *k*, may generate a timestamp on an instantaneous event *e* denoted as *k*(*e*). If $k = z$, then since *z* is the sole reference clock in the system, *z*(*e*) is called the absolute timestamp of the event *e*. The duration between two events is determined by counting the microticks of the reference clock that occur between the two events. The granularity g^k of a clock *k*, may also be expressed as the nominal number n^k of microticks of the reference clock *z* between two microticks of this clock.

Clock Drift: The clock drift of a clock *k* between microtick *i* and microtick *i+1* is the frequency ratio between clock *k* and the reference clock *z*, at the instant of microtick *i*. This is expressed mathematically as;

$$\rho_i^k = \frac{z(microtick_{i+1}^k) - z(microtick_i^k)}{n^k} . \qquad (7.2)$$

Since a good clock has a drift close to one, for notational purpose the drift-rate or *ρ-bounded* expression is given as (Veríssimo and Rodrigues, 2001);

$$0 \leq 1-\rho^k \leq \frac{z(microtick_{i+1}^k)-z(microtick_i^k)}{n^k} \leq 1+\rho^k . \qquad (7.3)$$

A perfect clock will have a zero drift-rate. Maximum drift rates are provided in manufacturer data sheets. Typical values for real clock within their operating conditions are 10^{-2} to 10^{-7} sec/sec. Obviously, clocks which are not resynchronized leave their bounded relative time interval after a finite time.

Offset: The offset of microtick *i* between clocks *j* and *k* with the same granularity is given as

$$offset_i^{jk} = \left| z(microtick_i^j) - z(microtick_i^k) \right| . \qquad (7.4)$$

7.2.2 Global Clocks

The concept of global time is an abstract notion that is estimated by the proper selection of a subset of microticks from the synchronized local physical clocks. The granularity of the node-local perception of global time is referred to as macrotick. The number of microticks per macrotick is called the microtick-macrotick conversion factor. The following are the most important properties that depict the integrity of a global clock (Kopetz, 1997).

Precision: For an ensemble of *n* clocks, the maximum offset between any two clocks is the precision Π of the ensemble. The expression for clock precision is

$$\Pi_i = \max_{\forall 1 \leq j,k \leq n} \left\{ offset_i^{jk} \right\} . \qquad (7.5)$$

The precision is represented by the number of microticks of the reference clock. The process of mutual resynchronization to maintain a bounded precision is referred to as internal synchronization or state correction. The deviation between the different clocks of the ensemble must be within acceptable values for the valid operation of the system.

Accuracy: Accuracy characterizes how closely physical clocks are synchronized to the reference clock over a time interval of interest. The process of re-synchronizing a clock with the reference clock in order to maintain a bounded precision is called external clock synchronization. Clock rate correction can be achieved only by external synchronization.

Reasonableness Condition: The global time t is reasonable, if all local implementations of the global time satisfy the condition $g > \Pi$, where g is the global granularity. This condition bounds the synchronization error to less that one macro granule, i.e. the duration between two ticks. If the reasonable condition is fulfilled then for a single event e, observed by two clocks of the ensemble,

$$\left| t^{j}(e) - t^{k}(e) \right| \leq 1. \tag{7.6}$$

This means that the global timestamp for a single event can differ by at most one tick. This is the best that can be achieved (Kopetz, 1997).

7.2.3 Failure Mode

A physical clock may commit two types of failures: A clock commits a timing failure if it is not *ρ-bounded* (7.3), or the clock counter may become damaged by a fault so that its values are erroneous. Such error could lead to a Clock Byzantine Failure in an ensemble of clock, where local clocks receive inaccurate, untimely or conflicting information from a faulty clock. An example is a dual-faced clock which may give different values of time to different nodes. Synchronization assumes that the network connection may commit omission or performance failures but never crash (Anceaume et al., 1997).

Link Omission failure: A connection between nodes commits an omission failure if a sender's message inserted into its outgoing buffer fails to reach the incoming buffer of the recipient node.

Performance Omission failure: A connection commits a performance failure if it fails to deliver a message within its specified time.

7.3 The Synchronization Problem

Clock synchronization may be done through hardware, software or a hybrid of both methods as in the case of the IMC synchronization scheme. The former achieves very tight synchronization, but may require special hardware at each node and a dedicated network for synchronization. On the other hand, in software synchronization, nodes exchange synchronization messages to adjust their local logical clocks through special algorithms. The nature of the algorithm defines whether the synchronization scheme is

internal or external. It is important to note that internal synchronization secures precision, i.e. clock state; for any two clocks j and k and all microticks i.

$$\left| z(microtick_i^j) - z(microtick_i^k) \right| < \Pi . \tag{7.7}$$

On the other hand, external synchronization secures accuracy (clock rate) and subsequently bounds precision to within $\Pi = 2A$. Irrespective of the method employed, the synchronization algorithm in each processor has the following responsibilities (Veríssimo and Rodrigues, 2001; Anceaume et al., 1997):

- Generate a periodic resynchronization event.
- Estimate the values of remote clocks (may not be used in external synchronization).
- Provide each correct process with the value to adjust logical (virtual) clocks. At the end of a synchronization interval, (7.8) should be valid.

While there are numerous synchronization proposals in the literature, there is no holistic solution that may be applied to all situations. The following sections build on the case for the appropriate synchronization method for the IMC architecture.

7.3.1 Internal Synchronization

Internal synchronization raises a number of challenges. Firstly, state correction cannot be applied suddenly since it will introduce discontinuities (sudden jumps) in the time base. The solution is to spread the adjustment over a resynchronization time interval i.e., fast clocks become slower, and slow clocks faster, so they converge. Secondly, it has been proven by Lundelius and Lynch (1984) that given n clocks on a network with latency jitter of ε, the best internal synchronization that can be achieved even with perfect clocks is

$$\Pi = \varepsilon \left(1 - \frac{1}{n} \right). \tag{7.8}$$

Therefore, in internal synchronization, precision is affected by not only latency jitter, but also the number of good clocks in the ensemble. Thirdly, each clock sends a message to all others: In the case of averaging algorithms message contents are clock values, while with non-averaging algorithms the message is simply a signal. In both situations, a convergence function computes the value to be applied to the logical clock.

Clearly, the exchange of messages introduces communication overheads. Nonetheless, these algorithms are capable of curtailing the malicious Byzantine error described above. However, for *any* algorithm to tolerate k Byzantine errors or clocks, a total of $n \geq (3k+1)$ clocks are required (Kopetz, 1997).

7.3.2 External Synchronization

External synchronization aims at injecting the time of an external reference, the master clock, into the global time of an ensemble of slave clocks. Contrasting this with internal synchronization, clocks are synchronized individually from the master clock, rather than agreeing among each other. In a sense this method is an authoritarian process since the master imposing its view of external time on all the slaves, forcing them to either trust the master or use fault-tolerant configurations. Typical external reference clocks or time servers are a Global Positioning System (GPS) receiver and the Network Time Protocol (NTP) servers with synchronization tightness in millisecond and nanosecond ranges respectively. Alternatively, a high precision oscillator may be used as the reference clock.

Many methods have been proposed for external synchronization with traditional timeservers such as Berkeley and Christian's algorithms (Tenenbaum and van Steen, 2002). The IEEE1588 protocol is an emerging paradigm for achieving external clock synchronization on devices using regular data networks that support multicast such as Ethernet. The most precise clock on the network is elected by a simple algorithm to be the master clock. The synchronization process itself is done in two phases. In the first phase the clock offset between master and slave is corrected after the master cyclically transmits its clock value to the slaves in two second intervals. After this process the time differences between the clock and slaves is the network delay or latency. The second phase measures this delay by a round-trip process: A slave clock sends a "delay request" packet to the master at time TS_1. On reception of the packet, the master generates a time stamp, TM_2, and sends the time of reception back to the slave in a "delay response" packet time stamped with the transmission time, TM_3. Once the packet arrives, the slave records the arrival time TS_4 and calculates the delay for adjusting its clock as follows;

$$\Delta = \frac{(TS_4 - TS_1) - (TM_3 - TM_2)}{2}. \tag{7.9}$$

In order to mitigate traffic congestion, resynchronization is performed randomly between two and sixty seconds. There are a couple of challenges regarding this method. In purely software based implementations, time stamping is done by reading the clock when creating a packet for transmission. For an incoming packet, time stamping may be done by the packet reception interrupt service routine. This implies that the transmission latency may include both the network channel access uncertainty and the reception interrupt latency. When time stamping is done in the network application layer, synchronization precision is in the range of 1 ms. Precision can be improved to a 10 µs range if synchronization is implemented at the driver or kernel level of a real-time operating system (Gaderer et al, 2004). For even finer precision in the order 1 µs, hardware time stamping is required. In this method, the clock value is inserted directly into synchronization messages at the point of departure or entry into the node. Figure 7.1 shows the configuration of such a system (Mohl, 2003) with a clock hardware unit consisting of a highly precise clock and a time stamping unit (TSU).

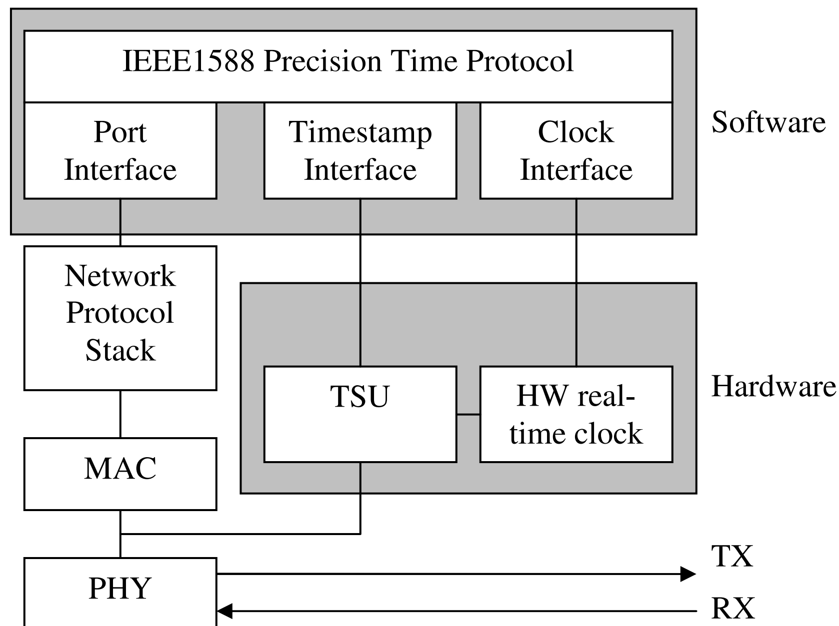


Figure 7.1: IEEE1588 Precision Time Protocol Architecture

7.4 The IMC Clock Synchronization Architecture

The sections above discussed the importance of clock synchronization in a real-time distributed network and various algorithms and protocols to achieve this. While internal clock synchronization corrects clock states, external synchronization corrects clock rates. Therefore external synchronization implicitly corrects the state of clocks. Both of these methods may require special computing hardware or networks. For example, the internal synchronization algorithms that run on time-triggered networks/protocols such as the Time Triggered Protocol (TTP) require special hardware. Therefore, it is quite difficult or impossible to incorporate such algorithms in distributed systems that do not provide the requisite hardware support. A rather radical approach is used for the IMC clock synchronization. All the methods described above make use of a logical (virtual) clock (one way or the other), and require an initialization phase. Eventually, quite a homogenous clock value is seen in all nodes and maintained by resynchronization. Instead of following the norm, the IMC clock ensemble does not care for literal clock values as much as it does for clock rate; i.e., clock accuracy implies clock precision. The reason is that the controllers on the IMC nodes are driven by hardware clock rates (i.e., edge-triggered); all related transactions such as command inputs to the controllers are either driven by the clock rate or by related events. The following section describes the synchronization architecture.

7.4.1 Assumptions and Properties

The IMC clock synchronization architecture is based on the following assumptions and attributes.

Assumption 1 (Reference Clock Integrity): *A reference clock H exists such that at time t* $|H(t) - t)| < \delta$*, where* δ *is an a priori given error.*

Assumption 2 (Bounded Transmission Delay); *the real time transmission delay is within some known bounds* $[\Delta - \varepsilon, \Delta + \varepsilon]$.

Assumption 3: *There is no direct access to the Ethernet hardware MAC, therefore hardware time-stamping of IP messages is not possible. To circumvent this liability, the highest possible thread priority is used for receiving, servicing and sending messages.*

Assumption 4: *Communication between the reference clock and the IMC nodes is by multicast. All good nodes receive simultaneous messages with time variation* t_v, *where* $t_v << 0$.

Property 1 (State Correction): *If all clocks of an ensemble are synchronized with accuracy A, then the ensemble is also internally synchronized with a precision of at most 2A*.

Property 2 (Oscillator Property): *All clocks in the ensemble have a maximum drift rate that defines a drift-window around a nominal frequency. A faulty clock oscillates outside the drift-window.*

Property 3 (Clock Hardware): *The JStick microcontroller has low-level drivers for controlling its timers and counters. The motion controller associated with each IMC is clocked by its JStick.*

Failure: *The IMC node is designed to be fail-silent, thus a faulty clock puts it in a fail-silent state*.

The operation of the master-slave synchronization process between the coordinator and the IMC nodes is as follows:

- The real-time coordinator hosts the reference clock, which is assumed to have a drift rate of less than 10^{-6} seconds/second. The real-time node is connected to an external reference time server to fulfill Assumption 1. A δ value of 1 μs is selected.
- The real-time coordinator implements high priority time-sliced real-time threads for all transactions.
- In the first phase of interaction, the communication latency Δ, between the coordinator and each IMC node is determined by measuring a round-trip message delay.
- The second synchronization phase is combined with the normal modus operandi of the real-time coordinator. The primary role of the coordinator is to serve the IMC nodes with real-time position set-points at an interpolation rate of *T* milliseconds (the maximum trajectory update rate of the motion controllers). Since this is the reference clock, it is assumed that an omniscient observer will see an event from the coordinator every *T* milliseconds. The coordinator begins this phase by multicasting a control signal to the IMC nodes at time *t1*. Upon reception of this signal at time $t1 + \Delta$, each node activates a hardware frequency counter to count its timer clock cycles. At time *t1*

+ T, and subsequently every period T, the coordinator multicasts an event (set-point). The IMC node checks the value of its counter when it receives the event and if the counter is ahead or behind, it adjusts its timer accordingly and resets the counter. If the difference is pernicious (in the order of milliseconds), the node informs the system coordinator: Depending on the fault-tolerance method selected by the user, the system coordinator may shut down all controllers, only the faulty node or do nothing.

7.4.2 Analysis

In this section, the synchronization scheme is analyzed for the precision and accuracy it provides to the clock ensemble. The implication of this method on the controller architecture is also discussed.

The drift offset Γ, of any two clocks in the ensemble depends on the length of the resynchronization period T and the maximum specified drift rate ρ of the clocks:

$$\Gamma = 2\rho T \ . \tag{7.10}$$

Due to network latency jitter, the precision of the real-time coordinator has to be correct as follows;

$$\Pi = \varepsilon + \Gamma \, . \tag{7.11}$$

Typical values are; $\rho = 1 \cdot 10^{-6}$ sec/sec; $T = 10$ ms; $\varepsilon = 0.01$ ms. From (7.10) and (7.11), $\Pi \approx 0.01$ ms. Using **Property 1** above, the accuracy of the ensemble is at least 0.005 ms. This is the best synchronization that is achievable with his scheme. Figure 7.2 shows typical synchronization analysis of three IMC clocks. More analytic results detailing the impact on motion coordination are discussed in Chapter 9 of this thesis.

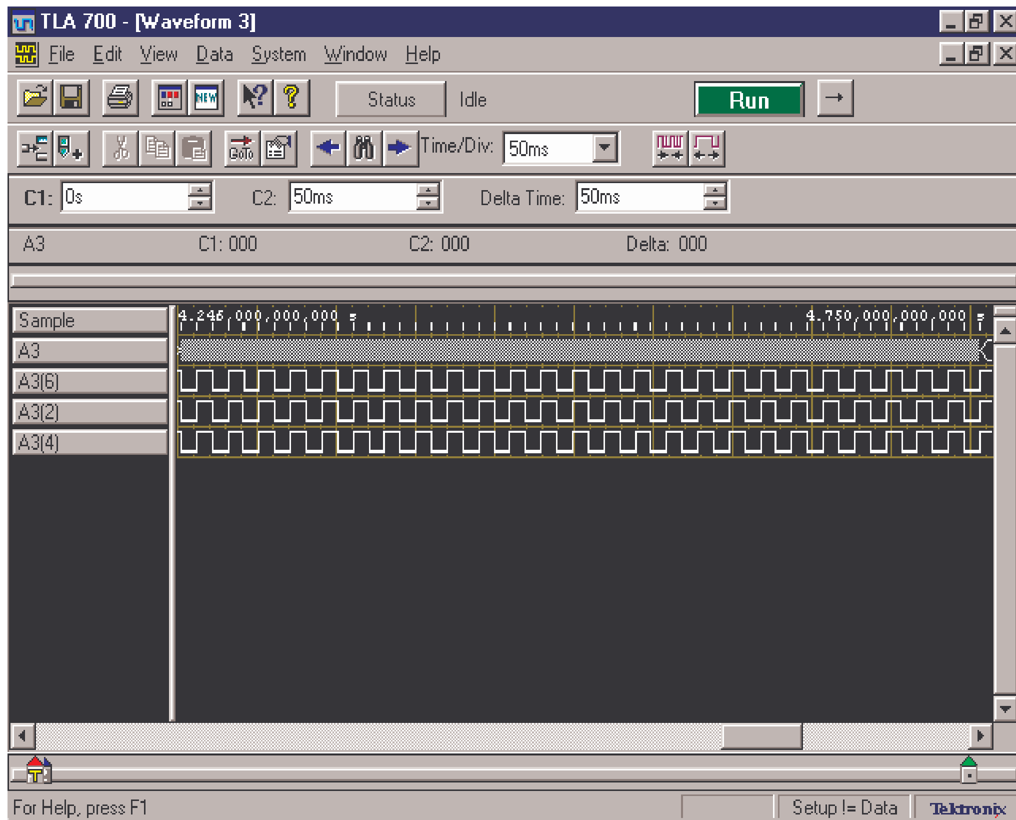


Figure 7.2: Clock Synchronization Capture on a Logic Analyzer

7.5 Conclusion

In this chapter, a number of issues relating to clock and clock synchronization methodologies were discussed. It was realized that external clock synchronization provided a convenient and simple way to synchronize the rates of a clock ensemble. Moreover, the need for logical clocks was undermined since most IMC real-time events are directly driven by the edge-triggered events of their respective clocks. Based on the characteristics governing external clock events and the properties of a global clock, a simple synchronization scheme was developed for the IMC architecture. In this scheme, the synchronization procedure is integrated with a trajectory generation on a real-time coordinator in order to avoid network traffic congestion. The best accuracy obtainable is 0.01 ms. Tests were performed with the aid of a logic analyzer.

8. TRAJECTORY PLANNING

8.1 Introduction

Motion planning for robotized processes is much more complex than that of NC machines. The prime reason is that a robot is built for transportation and/or manipulation tasks, and these require change of position in space and motion to some significant distance in comparison with its size. This raises the problem of mapping task space positions (orientation and translation) and velocities to appropriate joint space parameters. The three aspects of motion planning, i.e., path planning, trajectory planning, and trajectory tracking are indeed very broad areas. Path planning is the determination of the geometry of the motion, while trajectory planning is the determination of the time history (velocity) of the motion. The objective of trajectory tracking is to plan the control action which guarantees that the prescribed path is realized within desired accuracies. In many situations, the three aspects are highly interconnected. The IMC architecture by no means exhausts the various aspects of motion planning. Rather, the supporting underpinning allows for the implementation of high level task specifications as in the case of path planning. This chapter gives a brief overview of motion planning and describes the mechanisms provided by the architecture for trajectory planning. Since we are dealing with resource-constrained computing systems, the trajectory planning mechanism is designed around the most computationally efficient methods.

8.2 Planer Motion Trajectory Planning

The main functions of an interpolator are as follows (Weck, 1984):

- The geometric data produced by the interpolator shall approximate as close as possible to the desired path or contour.
- Since the most widely used contours are straight lines and circular curves, an interpolator should therefore be capable of at least linear and circular interpolation.
- The velocity of the axes must be kept within limitations and be independent of the contour.

- The final destination of the travel should be reached exactly as specified in order to avoid build-up (roundness) errors.

Two types of interpolator architectures identified in literature are hardware interpolators and software interpolators. Hybrid architectures or two-stage interpolators are quite common in modern designs. While a single-stage interpolator converts input data directly into colossal sequential axial co-ordinate values, the two-stage interpolator firstly provides intermediary or rough reference points to a fine interpolator. The latter subsequently determines the intermediary co-ordinate values – typically linearly, between the reference points. Such architecture permits the use of micro-processors with comparatively low capacity for the fine interpolation stage. The most common types of interpolation between coordinated axes are linear and circular interpolations. When a circular interpolation in one plane is superimposed upon a linear motion in a perpendicular axis a helical, spiral or screw-path interpolation is obtained. Higher order interpolations such as parabolic or elliptical are becoming quite common as well. Interpolation techniques are based on exact mathematical relationships of the following fundamental forms (Weck, 1984):

Implicit representation: $F(x, y, z) = 0$;

Explicit representation: $x = F(y, z)$, $y = F(x, z)$ $z = F(x, y)$;

Parameter representation: $x = F(\beta)$, $y = F(\beta)$, $z = F(\beta)$, where β is a common parameter such as time.

If the common parameter is proportional to time, then the functional dependency on time is automatically considered. Parameter representative techniques have the advantage that the resultant velocity will be constant if the resolution of the interpolator is constant with respect to time. Hence the velocity is not a function of the path being described.

8.2.1 Interpolation by Search Technique

The search technique (Weck, 1984) is derived from the implicit function representation of a plane. Each point on the prescribed contour satisfies the function equation $F(x, y) = 0$ but for all other points outside the contour the equation $F(x, y) \neq 0$. The magnitude and sign of the value is determined by the amount and direction of the instantaneous deviation of an interpolation point with respect to the contour. If the contour

form is consistently in one direction, then the sign will be sufficient to indicate which axial direction the next increment must be given. The search method can only consider the functional dependence of two variables in any one calculation. Hence for interpolation in more than one plane, a common reference axis for the determination of the positional values will be required. The search method may be applied in circular interpolation and the calculation of single-directional functions of a higher order.

8.2.2 Linear Interpolation by Digital Differential Analyzer (DDA)

The DDA technique is one of the most favored techniques for interpolation and is based on the mathematical integration of the velocity components (Weck, 1984). The linear interpolation DDA solution for a 2-D contour is as follows:

Given the starting and end points of a line to be $P_s(x_s, y_s)$ and $P_e(x_e, y_e)$ respectively, the intermediate values is determined as a function of time by the equations,

$$\begin{aligned} x(t) &= x(k \cdot \Delta t) = x_s + \int_0^t f_x dt, \\ y(t) &= y(k \cdot \Delta t) = y_s + \int_0^t f_y dt \end{aligned} . \quad (8.1)$$

For simplicity, the interpolation period T may be divided into N equal time intervals, each of duration Δt. However to account for acceleration and deceleration, the axes velocities f_x and f_y and the interpolation time interval T_i will vary with time during these periods but remain constant during the constant feed phase. A technique which is based on constant displacement increment follows (Altintas et al, 1996). The above equation may be expressed in discrete form as,

$$\begin{aligned} x(k) &= x_s + \sum_{j=1}^{k} f_x(j)T_i(j) = x_s + \sum_{j=1}^{k-1} f_x(j)T_i(j) + f_x(k)T_i(j), \\ y(k) &= y_s + \sum_{j=1}^{k} f_y(j)T_i(j) = y_s + \sum_{j=1}^{k-1} f_y(j)T_i(j) + f_y(k)T_i(j), \end{aligned} \quad (8.2)$$

or

$$x(k) = x(k-1) + f_x(k)T_i(k),$$
$$y(k) = y(k-1) + f_y(k)T_i(k). \qquad (8.3)$$

The axes velocities at time interval k are

$$f_x = \frac{\Delta x}{T_i(k)}, f_y = \frac{\Delta y}{T_i(k)}. \qquad (8.4)$$

The incremental displacement in both axes remains constant as follows;

$$\Delta x = \frac{x_e - x_s}{N}, \Delta y = \frac{y_e - y_s}{N}. \qquad (8.5)$$

Substituting (8.5) and (8.4) into (8.3) yields the recursive digital linear interpolation equations;

$$x(k \cdot \Delta t) = x_s + k \cdot \Delta x = x(k-1) + \Delta x,$$
$$y(k \cdot \Delta t) = y_s + k \cdot \Delta x = y(k-1) + \Delta y. \qquad (8.6)$$

For a trapezoidal velocity profile (Altintas et al., 1996), N is divided into acceleration (N_1), constant velocity (N_2) and deceleration (N_3) regions. Supposing acceleration A is from feed f_0 to f, the following can easily be proven for N_1:

$$N_1 = \frac{f^2 - f_0^2}{2A\Delta u}, \qquad (8.7)$$

where Δu is the displacement step which is kept constant. Similarly if deceleration D, is from feed f to f_l, N_3 can be deduced as

$$N_3 = \frac{f^2 - f_l^2}{2D\Delta u}. \qquad (8.8)$$

The interpolation period T_i for each interpolation interval varies in the acceleration and deceleration zones as follows:

$$T_i(k) = \frac{2\Delta u}{f(k) + v \cdot f(k-1)} \Bigg|_{\substack{v = 1 \text{ for accleration,} \\ v = \text{-1 for deceleration}}}, \qquad (8.9)$$

but remains constant in the constant velocity zone as

$$T_i(k) = \frac{\Delta u}{f}. \quad (8.10)$$

The incremental size is calculated based on a given minimum interpolation interval. A real-time implementation of this technique is fairly simple and computationally efficient. The implementation details are discussed in Section 8.5.

8.2.3 Circular Interpolation

Circular interpolation by second-order recurrence (Weck, 1984) divides an arc into N small chords each of length Δu and corresponding angular segment $\Delta\theta$. A chord error C (Fig. 8.1) is introduced as a result, which is the distance between the arc and the chord.

$$C = R\left(1 - \cos\frac{\Delta\theta}{2}\right), \quad (8.11)$$

where R is the radius of curvature. Hence the angular segment evaluates to

$$\Delta\theta = 2\cos^{-1}\left(1 - \frac{C_{\max}}{R}\right). \quad (8.12)$$

If the maximum chord error is constrained to one encoder count, then

$$\Delta\theta = 2\cos^{-1}\left(1 - \frac{1}{R}\right). \quad (8.13)$$

The corresponding chord segment is,

$$\Delta u = R\Delta\theta. \quad (8.14)$$

As in the case of linear interpolation, the tool path length N is divided into N_1, N_2 and N_3 segments for the acceleration, constant velocity and deceleration zones. The feed speed f is tangential to the arc and the travel distance is the segment Δu in every interpolation period T_i. It can easily be shown that the velocities in the x and y axis are

$$\begin{aligned} f_x &= -\frac{f}{R}\sin\left(\frac{f}{R}t\right) = -\frac{f}{R}\cdot y(t), \\ f_y &= \frac{f}{R}\cos\left(\frac{f}{R}t\right) = \frac{f}{R}\cdot x(t). \end{aligned} \quad (8.15)$$

Since the velocities are coupled with position, digital integration of the above equations will yield errors. A recurring circular interpolation (Altintas, 2000) has been formulated to overcome this. Considering the arc in Fig. 8.1, the coordinates for P_n can be expressed as:

$$x_n = R\cos\left(\theta_s + n\Delta\theta\right),$$
$$y_n = R\sin\left(\theta_s + n\Delta\theta\right). \tag{8.16}$$

It follows that

$$x_{n+1} = R\cos\left(\theta_s + (n+1)\Delta\theta\right),$$
$$y_{n+1} = R\sin\left(\theta_s + (n+1)\Delta\theta\right). \tag{8.17}$$

By using trigonometric functions to manipulate the above, the following recursive equations are derived:

$$x_{n+1} = 2x_n \cos\Delta\theta - x_{n-1},$$
$$y_{n+1} = 2y_n \cos\Delta\theta - y_{n-1}. \tag{8.18}$$

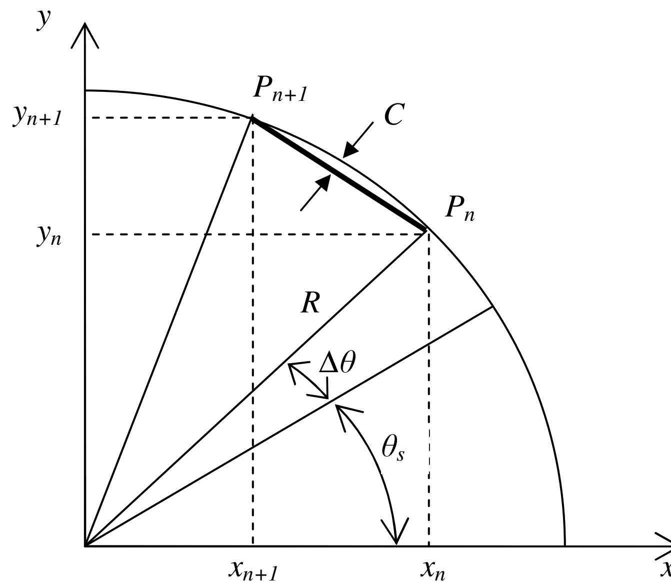


Figure 8.1: Circular Interpolation

This method is particularly suitable for real-time computing since there are minimal multiplications, additions, and subtractions. Moreover, $\Delta\theta$ evaluated from (8.13) may be stored in memory before interpolation begins. The different computing configurations described in section 8.2.2 apply in this situation as well.

8.3 Robot Motion Planning

As mentioned earlier, the essence of robotized processes is transportation and manipulation tasks. There are two cases very different from the motion planning view: These are continuous path (Cartesian path) and point to point (joint space path) with corresponding Continuous Path Control (CPC) and Point to Point (PTP) control (Somló, et al., 1997). With CPC, in every point of the robot motion the velocities and positions are computed. Moreover, it may be required that the orientation of working tools attached to the end-effector have given orientations. On the contrary PTP control requires only the initial and final points – the motion between these is determined by the kinematics and dynamics of the robot motion. Obviously, PTP control is computationally less demanding than that of CPC. Continuous path motion deeply involves all aspects of motion planning, i.e., path planning, trajectory planning, and tracking.

Trajectory planning is in itself a challenge in robot motion because apart from mapping task space motion (end-effector path and tool orientations) to joint space, the technological constraints (e.g. velocity, torque, etc) on actuators must not be exceeded. In other words, there is a possibility that the planned path may pass through singularities or unreachable workspace. Moreover, actuators do not reach their limits at the same time due to the dynamics of the motion. In situations where the motion path is known, complicated methods are available for optimal control or trajectory generation. However, in practice, simpler approaches are normally sufficient (Somló, et al. 1997). One of these is the use of trapezoidal velocity profile discussed earlier. In this approach, the working point moves on the path with given constant velocity after going through a given acceleration phase. The joint velocities are computed by inverse transformations. Another approach is to use spline curves to define the motion. In this case, the coordinates of a series of points in Cartesian coordinate system are given and the corresponding joint coordinates are determined by inverse kinematics. The trajectory planning problem is to determine the joint positions, velocities and possibly acceleration/deceleration values and of course, the time of motion

from point to point. The desired paths for joints satisfying the given boundary conditions in the given point can then be determined using proper order splines. In all approaches to trajectory planning, the last phase involves generating in real-time, finely interpolated trajectory set-points for or by the controller:

Let

${}^{S}_{T}T(k)$ for $k=1,..,N$ denote the initial pose, via-points (poses) and final goal (pose); S is the stationary frame: ${}^{S}_{T}T(1)$ is the initial pose and ${}^{S}_{T}T(N)$ is the final goal. For $k=1,.., N-1$, a smooth path in $SE(3)$ is generated (for example by splines) that connects ${}^{S}_{T}T(k)$ and ${}^{S}_{T}T(k+1)$. After a CPC trajectory,

$$ {}^{S}_{t}T \in SE(3), \quad t[t_i,\ t_f] \tag{8.19} $$

is generated, it must be converted into a sequence of set-points via inverse kinematics for the controller.

The time axis is digitized as

$$ t_s = t_i + s \cdot \delta,\ s = 1,\ldots,N,\ \delta = \frac{t_f - t_i}{N}, \tag{8.20} $$

with intermediary or via-points

$$ T_s = {}^{S}_{t_s}T = {}^{S}_{t_{i+s\delta}}T . \tag{8.21} $$

The via-points are converted into joint set-points

$$ \theta_s = K^{-1}\,{}^{S}_{t_{i+s\delta}}T,\ s = 1,\ldots,N, \tag{8.22} $$

where K^{-1} represents the inverse kinematics.

As mentioned earlier, in PTP or joint space control the actual Cartesian position of the end effector is only given at the specified initial, end-point and intermediary points (way-points or via-points). The path is converted into joint coordinates using the inverse kinematics of the manipulator and a smooth time trajectory for each joint position θ_i is calculated based on the given initial and final values. As in the case of CPC, trapezoidal or spline interpolation may be used for the trajectory planning. Set-point generation for the

joint controllers is then simply a matter of finely digitizing sequence of points on the trajectory.

8.4 Other Interpolation Methods

There are several other interpolation methods used to realize appropriate application-dependent trajectories. Nonholonomic constraints is a phenomenon which occurs when the generalized velocity vector of a mechanical system are non-integrable to equivalent configuration space constraints. The effect of this is that the instantaneous velocity is limited to certain directions. Nonholonomic interpolation usually involves motion in a plane with three degrees of freedom constrained to two in translation and one in rotation (Divelbiss, 1997). This type of constraint occurs in mobile robots, automobiles, orbiting satellites and space-based robot manipulators. Another class of controlled mechanical systems that exhibits nonholonomic behavior is under-actuated robots, i.e., robots with passive degrees of freedom. These mechanisms range from nonprehensile manipulation to robot acrobatics, from legged locomotion to surgical robotics, from free-floating robots to manipulators with flexibility concentrated at the joints or distributed along the links (De Luca, 2002). Motion planning solutions for nonholonomic systems require intuitive geometrical approaches and optimization techniques for prudent computational efficiencies. Some approaches include methods that emphasize optimality and those that emphasize feasibility (Divelbiss, 1997).

In situations where motion is in a three-dimensional space with three degrees of freedom or six degrees of freedom, holonomic spatial interpolation is used. Examples include Spherical Linear Interpolation (SLERP) and screw axis interpolation. SLERP is derived from representing the relative rotations of two rigid bodies by unit quaternions. Quaternions are a generalization of the complex numbers that can be used to represent three dimensional rotations. The set of all unit quaternions form a 4D unit sphere. Consequently, the problem of interpolation can be seen as the problem of finding the great-circle arc between two points on the 4D sphere. It has been proved that SLERP corresponds to rotation around a fixed axis with constant angular velocity (Strandberg, 2004). A screw motion is a combination of two simultaneous motions of an object; a linear translation and a rotation around a constant axis parallel to the translation vector. The trajectory of any point on the moving object is a helix and the velocity vector of the point

remains constant with respect to the object's local coordinate system. Thus, screw motion can be decomposed into a pure rotation or a pure translation, since they produce consistently the same relative trajectory – regardless of the choice of the coordinate systems (Rossignac, 2001).

8.5 Implementation on the IMC

The IMC framework supports a variety of motion modes to support the aforementioned trajectory planning schemes: This include jog-mode, position-mode, synchronized-position-mode, velocity-mode, and coordinated-motion. Figure 8.2 shows the flow chart for the different modes, except coordinated-motion.

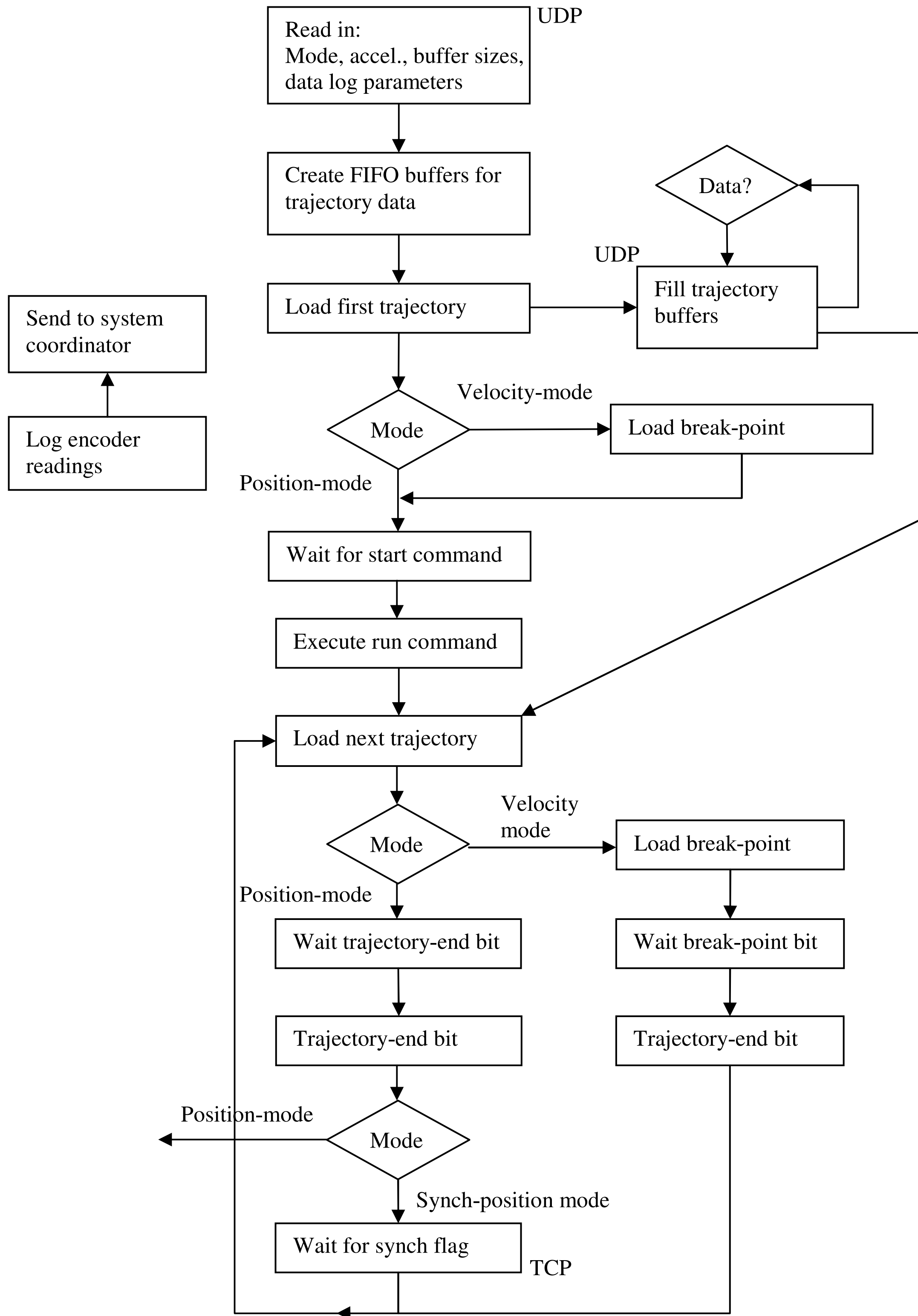


Figure 8.2: Flowchart for Different Trajectory Configuration Modes

In jog-mode, a single set of user-defined trajectory data (final position, maximum velocity and acceleration) is required to jog an axis to the commanded position. On the other hand, in position-mode, each IMC receives a continuous stream of set-points from the system coordinator into a FIFO buffer. The stream is temporarily interrupted by an object-lock operation when the buffer is full, thus enabling a much smaller buffer size to be used. Only the start of motion is synchronized, thereafter each IMC coordinates the operation of its own motion controller. The motion controller has a register that holds only one set of trajectory data at a time; therefore the IMC host continually updates this registry until motion is completed. Position is controlled along a trapezoidal trajectory profile from start to completion. On the contrary, in velocity-mode, the controller tracks velocity along the profile without coming to rest until a stop command is issued. The velocity may be varied on the fly or any other action may be taken upon the emergence of a breakpoint interrupt. This interrupt is triggered when a preloaded position reference is reached in the trajectory. Synchronized-position-mode is similar to position mode; except that each IMC node signals the system coordinator each time a commanded position is reached, loads its controller's trajectory register with the next data, and holds to receive a multicast "go" signal from the coordinator.

In coordinated-motion (Fig. 8.3), the architecture may be configured such that the real-time coordinator multicasts set-points to the IMC nodes at each interpolation period. Each IMC node extracts its data from the multicast package according to its given axis ID and commands its controller which in turn fine-interpolates at a minimum of 0.341 ms. The trajectory planer may be hosted either by the real-time coordinator or by the system coordinator. If the latter has real-time services, set-point data may be multicast directly to the IMC nodes; otherwise, data is channelled through the real-time IMC coordinator by a buffer-send technique. The advantage of this configuration is that large files (e.g., NC code), and computationally expensive motion planning can be handled outside the resource-constrained IMC nodes on a more powerful computer platform: A typical example is the high computational cost of the inverse kinematics of complex serial robots. In situations where trajectory generation (including the inverse kinematics) is decomposable, each axis is automatically configured to compute its own incremental displacements; for example, in linear interpolation, an algorithmic representation of

equation 8.6 is used. During the interpolation process, the real-time coordinator manages all the IMC nodes with a real-time periodic thread; the periodic value T is supplied by the system coordinator[3]. In one configuration style, the coordinator calculates the number of interpolation iterations, the step size for each axis displacement, the remainder for each axis step size, and sends them to the IMC controllers. The coordinator then proceeds to compute the interpolation time intervals, which may be higher than T; at each scheduled period T, its real-time thread multicasts the difference δ, between the interpolation time and T to the controllers. On the IMC ensemble, the following cyclic process takes place until the trajectory segment is completed:

1. Each node calculates the next displacement and velocity from equations (8.6) and (8.4) and loads its controller trajectory registers with these values.
2. Each node sees a real-time image (value) of δ at every period T; this value is loaded into a timer counter that counts down.
3. A countdown to zero triggers a *run* command to be sent to the motion controller. The motion controller then proceeds to finely interpolate the loaded trajectory, and simultaneously control the actuator.

In situations where the kinematics of the mechanism demand a more centralized computational structure, the coordinator generates all set-points and transmits them (including the interpolation time interval) to the IMC nodes at each scheduled period.

[3] The user-selected interpolation period is decremented by a pre-computed value to account for network latency.

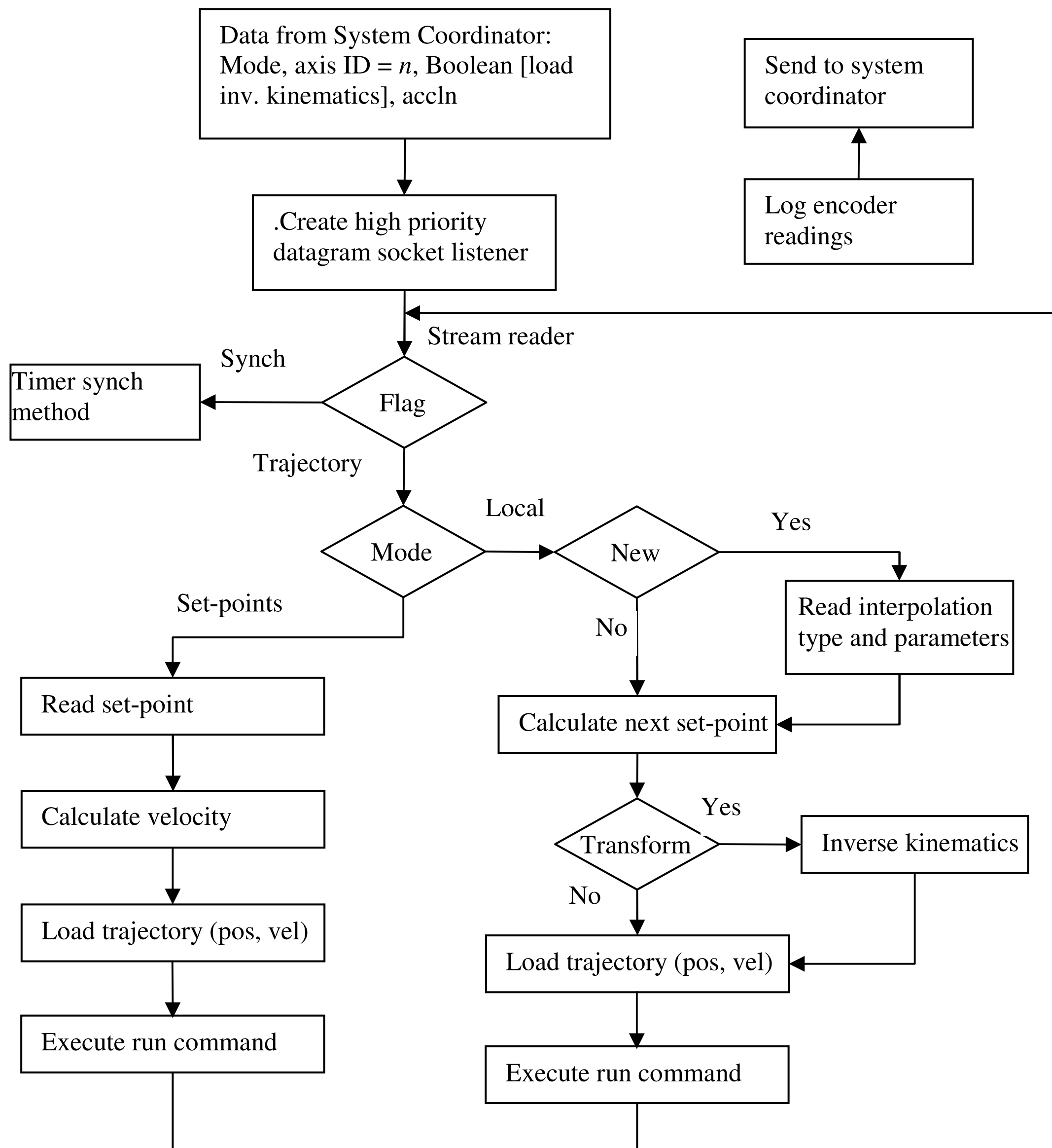


Figure 8.3: Flowchart for Coordinated-Motion

8.6 Conclusion

In this chapter, an overview of motion planning and trajectory generation techniques has been discussed. Subsequently, trajectory planning schemes have been adopted and developed for the IMC architecture. Two fundamental interpolation schemes

used on the IMC system are linear and circular interpolations. The next chapter presents details of the IMC software architecture.

9. THE IMC SOFTWARE ARCHITECTURE

9.1 Software Development Phases

The software architecture is based on Java's rich object-oriented style of programming. This greatly leverages building software components with standard interfaces and reuse capability. Reuse is achieved by generality and extensibility. For example, it is possible to have a generalized forward kinematics component for different robots. Extensibility can be achieved through *inheritance*, where one piece of code extends the functionality of another. Object-oriented software allows for the building of components with standard interfaces and reuse property. There are three major steps in the design of object-oriented components (Kapoor, 1996). The first step is the analysis of the problem domain and sub-domains. This results in a set of entities in the form of classes or objects, the relationship between these entities and their functionality. The next step is the design phase where decisions are made based on the execution platform, the programming language and the operational constraints. The last step is the software implementation.

9.1.1 Analysis

The purpose of the analysis phase is to provide a model for the behaviour of a system. This means identifying the entities of the system. Generally, the analysis process can be decomposed into the following steps:

Identify the entities in the application domain. The identified entities generally lead to defining classes and objects in the design and implementation phases. It should be easy for the designer to identify and name the behaviour of an entity. Moreover, the size of the abstraction should be appropriate.

Identify the responsibilities of the entities. In this second stage, the responsibilities of the entities are characterized by the services and behaviours of classes. The objective here is to create efficient interfaces that provide the maximum possible functionality.

Identify the relationship between entities: Object-oriented software architecture leverages the definition of crisp relationship between entities or classes. Two or more classes have an IS-A relationship if there is a parent-child or inheritance connection. On the other hand, a HAS-A relationship indicate containment of a class within another class, or of an object within a class, or of an object within an object. Another type of connection is the USES-A relationship. This relationship is present if the function interface of a class takes an instance or object of another class as a parameter. Generally USES-A relationship comes in handy when two or more classes need to collaborate to accomplish a task.

9.1.2 The Design Phase

The design phase results in the definition of classes and objects identified in the analysis phase. At this stage, names are given to classes and these names should reflect the semantics of the application domain. Furthermore in this phase, relationships identified in the analysis phase are transformed into inheritance hierarchies or containment relationships.

9.1.3 Implementation

The implementation phase involves filling in the details of the class data structures, adding internal functionality to support overall class functionality, and writing member functions. Testing and validation take place during implementation. Generally, a software designer will have to juggle the analysis, design and implementation activities a few times before arriving at a satisfactory architecture. Therefore these activities are not necessarily sequential.

9.2 The IMC Architecture Software Abstraction Development

Categorically, the goal of this research is to design a distributed reconfigurable controller for robotic applications. Generally, researchers have made many strides in developing specific robot application programs that are generic and hence can be reconfigured for various platforms. Similarly there are a plethora of distributed control systems and even distributed and reconfigurable I/O interfaces. Our focus in this research departs from these classic design approaches by employing a software and hardware infrastructure that supports a wide range of services. The overall architecture was described in Chapter 3. We concluded that most computationally intensive functions

needed to be hosted by a system coordinator running on a Workstation. Software components had to be developed for the IMC in the following areas:

- Low level control requiring hard real-time services.
- Well defined mechanisms for communicating and collaborating with the system coordinator.
- Mechanism for configuring device and auto-configuration.
- Partial kinematics.
- Interpolation.
- Interface for high level control.
- Kinematics interfaces.
- Graphical user interfaces.
- Configuration and auto-configuration interfaces.

9.2.1 Analysis

Analysis had to be performed on these domains and main components had to be identified. Subsequently, software components had to be designed and tested. Three main domains were developed to support the IMC architecture. Figure 9.1 shows the IMC Domain which represents the IMC modular controllers – one per machine axis, the Real-Time IMC Coordinator which handles all system-level real time tasks and the System Coordinator which is the main interface between the user and the rest of the system. After defining key domains and their sub-domains, the next task was the analysis of the sub-domains or classes. The analysis led to the specification of the key entities in the sub-domain. In this phase, related domains were grouped into packages to enhance software reusability.

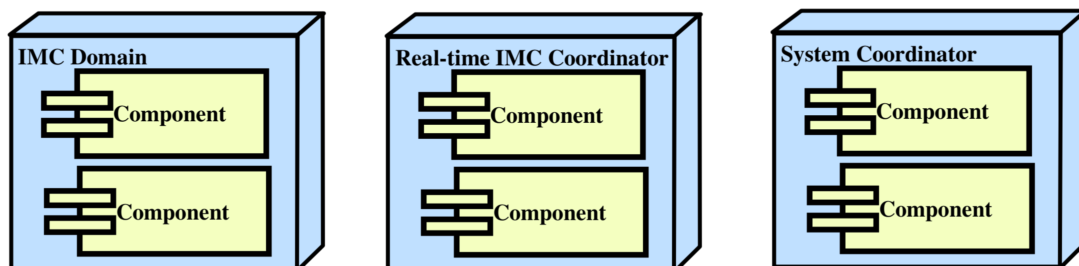


Figure 9.1: The IMC Architecture Software Components

9.2.2 Design

Design issues are more relevant at the sub-domain or component level. Some of the key design issues that were employed are as follows.

- All classes developed had to have meaningful names and placed in clearly defined packages. An example is *DatagramServer.class* which belongs to the *com.IMC.network* package.
- Real-time threads were defined on the IMC platform for scheduling real-time tasks, while non-real-time threads handled non-real-time tasks.
- The priority levels of threads were clearly defined.
- Tasks demanding hard real-time were assigned a higher priority than the automatic garbage collector.
- Minimum garbage was generated in classes. Therefore string writes were minimal.
- Efficient coding was used since resources are limited on the IMC JSticks. This involved for example avoidance of unnecessary copying of objects and suspension of superfluous thread loops.

9.3 The IMC Domain

The IMC domain is responsible for axis-specific activities such as joint control. In addition, the domain contains many components for intelligent interaction with the rest of the system. Table 9.1 shows the packages in this domain.

Table 9.1: Software Packages

Package	Description
com.IMC.database	Network, configuration data and data access methods
com.IMC.coordination	Protocols for negotiating with system coordinator
com.IMC.drivers	Low level drivers for the communicating with the LM628, interrupt services, and digital I/O drivers
com.IMC.network	Ethernet protocols
com.IMC.servlets	Web interface for viewing/editing configuration and PnP mechanisms

9.3.1 Database Abstractions

The database abstraction provides a data warehouse for static and dynamically generated data, and also data access policies. The subcomponents are listed in Table 9.2. Interface definitions are provided in Appendix B1.

Table 9.2: The IMC Database Components

Database Component	Description
Data	Network and configuration data
PushPullData	Data access mechanisms for collaborating threads
FIFO	First-In-First-Out Buffer
JmDNS_Coordinator_Data	Contains data for publishing and subscribing services o the network
FileServer_ConfigFiles	Controller parameters

9.3.1.1 Data

The Data class is used to store key static parameters needed by the IMC controller. The analysis and design issues are presented below.

Analysis

A data warehouse provides persistent data to the domain. Its attributes and desire functionality are as follows:

- The data class should distinguish between modifiable static data and *final* data.
- The class should contain all network and configuration parameters
- It should not place a limitation on inheriting.

Design

Based on the analysis above, the class specification was designed and implemented as described below.

- *Data*.class is made to be a member of the *com.IMC.database* package.
- The class is given the modifier *abstract* to enable other classes inheriting its properties to have the flexibility of inheriting other properties.
- All parameters are *static* and default parameters are qualified with *final*.

Example

An example of the implementation of this class is as follows.

1. *Myclass implements Data{*

```
2.  int datagramPort = PRIORITY_DATAGRAM_PORT_NUMBER; //inherited from Data }
```

9.3.1.2 PushPullData

Sometimes two or more threads need to collaborate to accomplish a task. For example a thread may need to know if another thread has finished its task. However, when two threads share the same data, complexities may arise due to a phenomenon called *race condition* (Oaks and Wong, 2004). Java uses a concept called synchronization to solve this problem. When a method is declared synchronized, the thread that wants to execute the method must acquire a token or a lock. Once the method has acquired the lock, it executes the method and releases the lock. There is only one lock per object so if two separate threads try to call synchronized methods of the same object, only one thread can execute the method; the other thread must wait for the lock to be released before it can execute the method. The *PushPull* class implements synchronized methods for various tasks and also stores short-term data.

Analysis

- The class should support collaboration between two threads calling on the same methods in this class.
- An object of this class should be available to all threads that need its services.

Design

- *PushPullData* is made to be part of the *com.IMC.database* package and declared *public.*
- Methods used by two or more threads are designed to be synchronized methods.

Example

This method below controls the start of motion.

```
1.  /**
2.  if velocity is 0 thread calling this method will be put in a wait
3.  state until notified. This is used to synchronize start of motion.
4.  */
5.  public synchronized double get_vel() {
6.  if (vel == 0) {
7.  try {
8.  wait();          //wait for all  trajectory data before starting  motion
9.  }
10. catch (InterruptedException ex) {}    }
11. return vel; }
```

9.3.1.3 FIFO

This class provides a first-in-first-out buffer for two threads. One thread fills the buffer from its *tail* whiles the other takes data from its *head.*

Analysis

- The class should provide methods to create any buffer size.
- This class should provide a method to fill the buffer from its tail and wait for the buffer to be partially empty before resuming the fill operation.

Design

- The *FIFO* class is part of the *com.IMC.database* package.
- A method is implemented to create an array of any given size.
- Synchronized methods fill and acquire data from the buffer.
- The *PushPullData* class contains instances of the *FIFO* class for creating velocity and position buffers. Therefore this connotes a *USES-A* relationship between the two classes.

Example

In the example below, buffers are created for position and velocity data.

```
1. public static void createPosVelFIFO(int size) {
2. positionBuffer = new FIFO(size);
3. velocityBuffer = new FIFO(size);
4. }
```

9.3.1.4 JmDNS_Coordinator_Data

The IMC architecture implements an auto-configuration network protocol called *JmDNS*, to enhance modularity and reconfigurations (see Chapter 5). The protocol is used to register configuration information and discover services such as the connection detail of nodes on the network. The *JmDNS_Coordinator_Data* contains data required by a *JmDNS* coordinator to register and detect these services.

Analysis

This class should contain data to compose *JmDNS* information. Some of the data such as network information is in the *Data*.class.

Design

- The class was place in the *com.IMC.database* package.

- The *JmDNS_Coordinator_Data*.class bears an *IS-A* relationship with *Data*.class since it inherits data from this class.

Implementation

The example below shows one of the methods in this class for creating a *JmDNS* register.

```
1.  public static String registerSingle(String type, String name, int port,
2.  int weight, int priority, String text) {
3.  StringBuffer register = new StringBuffer();
4.  register.append(type);        // type of service
5.  register.append(",");
6.  register.append(name);//name of the service
7.  register.append(",");
8.  register.append(port);      //port number for this service
9.  register.append(",");
10. register.append(weight);  //degree of persistence
11. register.append(",");
12. register.append(priority); //priority level
13. register.append(",");
14. register.append(text);             //text message
15. return register.toString();
16. }
```

9.3.1.5 FileServer_ConfigFiles

File creation, storage and retrieval are important for storing and modifying persistent data. The *FileServer_ConfigFiles* class contains various methods for storing configuration data such as PID filter values and *JmDNS* data.

Analysis

- The class should have methods to prepare data for storage such as adding delimiters to separate data.
- The class should have methods to create, delete and perform file I/O.

Design

FileServer_ConfigFiles inherits data from *JmDNS_Coordinator_Data*, which also inherits data from *Data*.class. Therefore this is an *IS-A* relationship. The class embodies various methods to add delimiters such as commas and end-of-line to data.

Example

In the example below, the method retrieves configuration parameters from *Data*.class and converts them to a String separated by the end-of-line delimiter.

```
1.  public String getConfig(String[] value) {
2.  StringBuffer buffer = new StringBuffer();
```

```
3.  for (int i = 0; i < value.length; i++) {
4.  buffer.append(value[i]);
5.  buffer.append('\n');          // parameters separated by \n ie "new line"
6.  }
7.  return buffer.toString();
8.  }
```

9.3.2 Coordination Abstractions

The coordination component abstracts an interface for the system coordinator to issue commands and receive feedback from the controllers. The coordinator also handles clock synchronization signals and real-time data from the real-time coordinator. As discussed in Chapter 8, the architecture supports various motion modes, including jogging, position-mode, synchronized-position-mode, velocity-mode, and coordinated-motion. Each IMC receives either a stream of set-points, or/and motion synchronization signals, depending on the mode selected by the user. These various tasks are coordinated by the components listed in Table 9.3. Their interface structure is provided in Appendix B2.

Table 9.3: Coordination Components

Coordination Component	Description
StateCoordinator	Receives datagram events from system coordinator
MultiCasted_States	Receives high priority multicast events signals
Device	Invoked by *StateCoordinator* to execute motion profiles
StateBuffer	Creates a temporary buffer for multicast signals received
Interpolation_Server	Receives set-point from an interpolator and commands the controller to move to set-point positions
Monitor	Establishes a TCP connection with system coordinator
EncoderReader	Periodically reads and sends encoder data to the system coordinator
Counter	Logs sensor data
MainClass	Main class for initializing the controller

9.3.2.1 StateCoordinator

The *StateCoordinator* class uses a network connection to receive events from the system coordinator, and call the right method to execute the requested command. All datagram packets received consist of a header and a body. The header contains a flag which indicates the type of incoming event, for example “set PID filter” and may contain

other parameters such as data packet size, etc, depending on the value of the flag. The body may contain associated data such as PID parameters or may be null. For some motion modes**,** trajectory data are received into FIFO buffers. When motion commences, i.e., when FIFO buffer consumption begins, the system coordinator is alerted to stream in the next data batch.

Analysis

- The class should have a datagram connection with the system coordinator. This is will enable packet sizes to be pre-allocated. Since a packet is an indexed buffer, its contents can be discharged into FIFO buffers systematically.
- Methods should be executed without blocking incoming commands; i.e., the class should run concurrently with the classes implementing these methods.

Design

- The class is implemented as a high priority thread with its "run" method embodying a perpetual loop. The loop waits forever for messages from the coordinator and makes functions calls on methods in other classes as dictated by the command flag received. Since threads run concurrently, these methods execute outside the thread's own execution block.
- The class inherits a datagram socket connection from *DatagramServer*.class in the *com.IMC.network* package. Therefore all connection related issues including exceptions are handled outside this class.
- In order to have access to the FIFO buffers, an object from the *PushPullData*.class is created within the class at runtime. The received data packet is used to fill the buffers by virtue of a *synchronized* method in the *PushPullData*.class which ensures that the buffers are not flooded. When the packet is emptied a flag is sent to the coordinator to resend the next packet. The back-and-forth communication continues until an end-of-data flag is received or a high priority interrupt pre-empts the operation.

Example

The example below shows a skeletal implementation of the *StateCoordinator* class.

```
1.  public class StateCoordinator
2.              extends DatagramServer implements Runnable {
3.              .. ..
```

```
4.   public synchronized void run() {
5.     .. ..
6.    while (true) {
7.     in_packet.setLength(inbuffer_length);    //reset buffer
8.      dgconn.receive(in_packet);            //receive packet
9.      state_Flag = in_packet.readInt();          //read first number of packet
10.     switch (state_Flag) {
11.             case 0:
12.           // receive trajectory data for jogging axis
13.                .. ..
14.             case 4:
15.            //position mode
16.                .. ..
17.             case 9:
18.           //Receive PID Filter value
19.            case n:
20.                          }
21.      }
22.   }
23.  }
```

9.3.2.2 MultiCasted_States

Multicast signals are very useful when all nodes on a network need to receive the same information simultaneously. The architecture employs this protocol for most synchronized behaviour such as synchronized start, stop, suspend, etc. The section below describes the multicast class.

Analysis

- Since synch signals are urgent commands, the class implementing this should be protected from pre-emption by most operations.
- The class should not generate garbage or run computationally intensive functions

Design

- The class is derived from *MulticastServer*.class in the *com.IMC.network* and is implemented as a thread with the highest priority in the Java API specification.
- The class receives only 4-byte packets and immediately dispenses them in a *storage* class in order to minimize the risk of losing an in-coming packet.

Implementation

The class is instantiated as follows.

```
1.   StateBuffer buffer = new StateBuffer()            ;//storage class
2.   Thread thread =new Thread(new MultiCasted_States(buffer));
3.   Thread.setPriority(10);       //10 is the highest thread priority
4.   thread.start();
```

9.3.2.3 Device

Methods were required for direct calls on controller drivers such as "load trajectory", "run trajectory", etc. Simultaneously, the class implementing these methods needed to cooperate with other classes in direct contact with the system coordinator to create a producer-consumer relationship.

Analysis

- The class should be implemented as a consumer and have methods to control the peripheral controller in the manner dictated by the system coordinator.

Design

- The class is implemented as a thread to run concurrently with the producers.
- The class retrieves data through the *PushPullData* class and frequently monitors the *StateBuffer* class for urgent signals.
- A loop is implemented in its *run()* method to periodically read instructions from the coordinators (producers) through the *StateBuffer* class. It was more computationally efficient to temporarily halt the loop when there was no immediate instruction therefore the instruction-read method was cast as a *synchronized* method with a *wait()*.

Implementation

The following is the section of its method for executing a "find home position" command.

```
1.     case 8: {
2.  //mask all interrupts except excessive position interrupt
3.          LM628.mask_reg(0x20);
4.            //reset loop flag. This will cause the main loop to wait until next command
5.          statebuffer.put_deviceRunFlag(false);
6.          load_PID_Filter(); //call PID filter load method
7.            //call reference switch driver
8.          Reference_Switch_Driver home = new Reference_Switch_Driver("home");
9.          break;
10.     }
```

9.3.2.4 StateBuffer

A class was required to hold high priority state information such as stop, run and suspend.

Analysis

- The class needs to employ safe methods which prevent race conditions or loss of data integrity.

Design

- The *StateBuffer* class is designed with the same concept as that of *PushPullData* class. Command flags (as opposed to data in the latter) are held in its buffers. Secondly, emergency stop commands are implemented directly in this class.
- The *StateBuffer* extends or inherits the properties of *PushPullData*, making it possible for instances of this class to have direct access to the latter.

9.3.2.5 Interpolation_Server

This class coordinates with the real-time coordinator to receive set-points, trajectory signals and clock synchronization signals.

Analysis

- The most important requirement is for the class to capture the real-time data stream and command the controller without any infringements. This means that method executions should be atomic and be guaranteed to commit.
- The real-time coordinator multicasts all set-points to all nodes in the multicast group. Therefore the class should have an efficient means to identify and extract set-points addressed to its platform.

Design

- The class is designed to inherit a datagram socket connection from the *DatagramServer* class.
- The class is given a high thread priority – above that of the Java Garbage Collection thread.
- When this class is instantiated by the *StateCoordinator* class, a set-point index generated by the system coordinator is passed to it. This index is transformed into a read-pointer to mark the position of the set-point data in the multicast packet.

Implementation

The class is instantiated and started by the *StateCoordinator* as follows:

```
1. Thread thread = null;
2.  thread = new Thread(new Interpolation_Server(mode, true, statebuffer));
3. rawJEM.setJEMPriority(thread, 26); // thread priority;    thread.start();
```

9.3.2.6 Monitor

Most communication across the network was designed to be connectionless in order to allow easy changes on the network. However, the system coordinator needed at least one connection-oriented link to serve as the life-line for emergency calls from the IMC, encoder readings and synch signals.

Analysis

The Monitor class should have the following attributes and functionality:

- The class implements TCP socket communication with the system coordinator and has methods to send and receive data.
- The priority level of an object of this class should be adjustable for high and low priority messages.

Design

The following describes the design of the *Monitor.class*.

- The Monitor class inherits its server socket communication method from *TCPServer.class* in the com.IMC.network package.
- The class is not implemented as a thread therefore its priority is not static. Rather, threads invoking a synchronized method of its object use a low-level method in the aJile API, to raise their own thread priority to a specified ceiling level. After returning from the synchronized method, the priority is restored to the priority prior to invoking the method.

Examples

The following examples show how the Monitor class methods may be applied.

```
1.  Monitor monitor; //Monitor object
2.
3.  // This procedure is  used by an interrupt listener to send an emergency signal
4.  com.ajile.jem.rawJEM.setCeiling(monitor,25);
5.  monitor.sendEmergencyStop();
6.
7.  //This is used by the Device class during synchronized- position-mode operations
8.  monitor.send_flag();          //send a synch flag
9.
10. monitor.send_data(int data);          //send encoder data
```

9.3.2.7 EncoderReader

Analysis

- The EncoderReader periodically reads the encoder signal from the controller's decoder registry and dispatches it to the system coordinator.

Design

- The class uses the *send_data* method in the Monitor class to send signals by TCP.
- The class is implemented as a thread. Before invoking the *send()* command, it calls on a synchronized method in the *StateBuffer.class*. If a higher priority operation is in process, the *send()* procedure is temporarily suspended.

Implementation

The following shows part of the body of *EncoderReader* class.

```
1.  public class EncoderReader extends Thread{
2.  .. ..
3.  public synchronized void run(){
4.   while(true){
5.  try {
6.  statebuffer.get_monitorBusyFlag();  //call synchronized method
7.  monitor.send_data(LM628.readPosition());    //send data
8.  Thread.sleep(Data.encoderRepInt);  //frequency provided by user
9.  }
10. catch (Exception ex) {}  }}}
```

9.3.2.8 Counter

The Counter class logs encoder positions in a buffer. The size of the buffer and frequency of logs is determined by the user before motion commences.

Analysis

The class has the following attributes and functionality:

- Its integer array buffer should be protected from index or illegal exceptions to prevent garbage generated by uncaught exceptions.

Design

- The array size is determined by the calling method.
- The logging method catches array exceptions locally.

Implementation

The following is the logging method of the class.

```
1.  public class Counter {
2.  .. ..
3.  public static void posCounter() {
```

```
4.          try {
5.         position[index] = LM628.readPosition();
6.          index++;
7.      }  catch (ArrayIndexOutOfBoundsException ex) {   } }}
8.    A typical invocation of the above method is as follows.
9.    int logCounter = in_packet.readInt();           //read user-defined buffer size
10.   Counter.position = new int[logCounter];        //set counter size
```

9.3.2.9 MainClass

The MainClass is the first to be executed. Its functionalities and attributes are as follows:

- It is responsible for instantiating and starting most of the coordinator threads, including *StateCoordinator.class*, *MultiCasted_States.class* and *Device.class*.
- It sets the controller's timer or clock.
- It starts a web server.
- It logs on to a time server to set JStick's wall clock.

9.3.3 Driver Abstractions

The driver component is responsible for all low-level operations related to hardware switches, I/O and the motion controller. Most of the functionalities of its sub-components have already been described in Chapter 4. Table 9.4 lists the drivers and a brief description of each. The interfaces are given in Appendix B4.

Table 9.4: Driver Component

Driver Component	Description
LM628	Contains methods to directly control the motion controller
HSIO_Driver	Driver for High Speed I/O operations between the controller and the JStick
Board_Clock	Enables/disables the board clock
JStickTimer_tc2	Programmable timer
LM628_Interrupt	Receives and services hardware interrupts from the motion controller
LimitSwitch	Limit switch drivers
Reference_Switch_Driver	Reference or home switch driver
GPIOPinA3	Digital I/O for external purposes

9.3.4 Network Abstractions

The network sub-components are designed to be base classes for all required Ethernet connections. A description of each class is given in Table 9.5.

Table 9.5: Network Component

Network Component	Description
DatagramServer	Creates Datagram Server Sockets
MulticastServer	Creates Multicast Server Sockets
TCPServer	Creates TCP Server Sockets

9.3.4.1 DatagramServer

The DatagramServer class provides the tools for all classes seeking to build a datagram server connection.

Analysis

The following are the attributes and functionality of the class.

- Connections made should be memory efficient.
- The class should leverage the creation of multi-server connections.

Design

- The class uses a memory efficient *DatagramConnection* method from aJile's API, and optionally provides standard connection methods in the J2ME network API.
- The class is instantiated with a server port number to allow multi-server connections.

Implementation

The interface is outlined in Appendix B5. A multi-server connection is simply established as follows.

1. *DatagramServer server1 = new DatagramServer(int port_1);*
2. *DatagramServer server2 = new DatagramServer(int port_2);*
3. *DatagramServer server3 = new DatagramServer(int port_3);*

9.3.4.2 MulticastServer

The MulticastServer class provides tools for multicast connections.

Analysis

The following are the attributes and functionality of the class;

- Connections established are memory efficient.

- The class method handles the protocol for joining multicast groups.

Design

- The class uses a memory efficient *MulticastConnection* method from aJile's API as opposed to the standard connection method in the J2ME network API.
- The class implements methods to join or leave a multicast group.

Implementation

The interface is outlined in Appendix B5. A multi-server connection is established as follows;

1. *MulticastServer server = new MulticastServer (String address, int port);*

9.3.4.3 TCPServer

The TCPServer class provides the tools for building TCP multi-socket connections.

Analysis

The following are the attributes and functionality of the class;

- Connections established are memory efficient.
- The server connection waits to accept and open a connection with a client before releasing control of its socket to the implementing class.

Design

- The class uses a memory efficient *StreamConnectionNotifier* method from aJile's API and also provides standard connection methods in the J2ME network API.

9.3.5 User Interface and Plug-And-Play Abstractions

> Man-Machine Interface: If an embedded system has a man-machine interface, it must be specifically designed for the stated purpose and must be easy to operate. Ideally, the use of intelligent product should be self-explanatory, and not require any training or reference to an operating manual (Kopetz, 1996).

The architecture provides a rich set of tools for PnP and man-machine interactions, transforming the otherwise black-box-like IMC modules into intelligent open modules. Using Java servlet technology, users can login to each IMC and view or change configurations and network services with a web browser. All sensitive transactions are password protected. The architecture uses Tynamo (Silverman, 2004) as the platform for building all servlets. Tynamo is one of the few ultra-concise web-servers designed

explicitly for embedded Java platforms. Before describing the sub-components, the following sections explain the underlying protocols leveraging these services.

9.3.5.1 Servlet Technology

Servlets are reusable Java applications which run on response/request-oriented web server. The server loads and executes the servlets, which accept zero or more requests from clients and return data to them. Functionally, they are similar to CGI scripts, but more platform-independent. A few of the many applications of servlets include the following;

- Servlets can process data posted over Https using an HTML form.
- Servlets can accommodate multiple requests concurrently to allow collaboration between multiple users.
- Servlets can forward requests to other servers and servlets in order to balance load among servers or partition a single logical service over several servlets.

There are two types of servlets:

1. Generic servlets are protocol independent, implying that they contain no built-in support for any transport protocols such as HTTP.
2. HTTP servlets support the HTTP protocol and are more relevant in web browser environments.

When a server loads a servlet, it runs the servlet's initializer method, *init()*, only once. The init method may be overridden when developing the servlet. Afterwards the servlet may handle client requests in its service method. The service method supports standard HTTP/1.1 requests by assigning each request to a designated method. When designing servlets, the following methods in the *HttpServlet* class may be overridden.

- *doGet*, for servicing HTTP GET, conditional GET and HEAD requests.
- *doPost* for handling HTTP POST request.
- *doPut* for handling HTTP PUT request.
- *doDelete* for handling HTTP DELETE requests.

The above methods take two arguments: The first, *HttpServletRequest*, encapsulates the data from the client, while the second, *HttpServletResponse*, embodies the response to the client. An *HttpServletRequest* object provides access to form data, cookies, session information, and URL name-value pairs. See Sun Microsystems (1999),

Berners-Lee et al (1996), and HTML 4.0 (1998) for more information on servlet technology, HTTP/1.1, and the HTML specification respectively.

9.3.5.2 The JmDNS Protocol

The *JmDNS* protocol was designed for standard Java (J2SE). Hence, over fifty method calls and procedures had to be modified or replaced in order to port it to the Java-CLDC profile running on the IMC controllers. All classes that dealt with network issues derived from *java.network* package, were substituted with CLDC equivalents or low-level detour methods. Many other classes like *ThreadGroup*, *Iteration*, etc not found in the CLDC package were substituted with other methods. The protocols were embedded in the IMC servers (servlets to be specific) to enable a web-based approach to register and discover services, and allow users to view and change configurations. The sub-components constituting the user interface and PnP (automatic configuration) are listed in Table 9.6. The interfaces are outlined in Appendix B3.

Table 9.6: Servlets Component

Servlets Component	Description
JmDNS_Coordinator	Implements JmDNS to register IMC services and discover specific services
ConfigureDevice	Creates a web-based interface for configuring the IMC
EditJmDNS	Creates a web-based interface for configuring the JmDNS services
PositionDump	Creates a web-based interface for viewing position logs
ControllerInfo	Creates a web-based interface for viewing info on the IMC
ShutdownServlet	Web-based interface for shutting down the web server

9.3.5.3 JmDNS_Coordinator

The *JmDNS_Coordinator* encapsulates the *JmDNS* protocol and is one of the first programs to be executed at runtime. Below are the attributes and functionality of this class.

Analysis

- It is a servlet class and auto-started by the server.
- The class uses *JmDNS* to register services and discover specific services.
- *Listeners* continue running to detect changes on the network.

- The class removes registered services (local) when the IMC stalls or shuts down.
- The servlet posts information on a web service.

Design

- The class is designed as an HTTP servlet.
- The server is pre-configured to auto-start or load this servlet when the IMC node is started.
- Its *init* method embodies methods to register and discover services; therefore this is done automatically at runtime.
- A listener thread is kept alive to detect services added or removed.
- Discovered services are posted on to the *com.IMC.database* package.
- The servlet provides a *doGet* method to post information in HTML format on a web browser.

Implementation

The init method of the servlet is as follows:

```
1.  public void init(ServletConfig config) throws ServletException
2.    {  super.init(config);
3.      try{
4.        //set HOST name in JmDNS
5.        JmDNS.DEFAULT_HOST_NAME="JStick-"+Data.DEVICE_NAME;
6.        jmdns = new JmDNS();
7.        listenerVector =new Vector();     // stored info. on services to be discovered
8.      }
9.      catch (IOException e) {    }
10.     discoverJmDNS();          //invoke method to discover services
11.       //don't register services if controller needs to be initialized
12.     if(!Data.initializeController){
13.         registerJmDNS();
14. }
15.  }
```

The servlet has an inner class that implements the JmDNS listener. The skeletal illustration is as follows:

```
120.  static class Listener implements ServiceListener {
121.          .. ..
122.    public void serviceAdded(ServiceEvent event) {
123.          .. ..
124.      //detect supervisor
125.      if (type.startsWith("_supervisor._tcp.local.")) {
126.        .. ..
127.      }
128.      //detect registered devices
129.      if (type.startsWith("_device._pid")) {
```

```
130.        .. ..
131.    }
132.//detect services removed
133.    public void serviceRemoved(ServiceEvent event) {      .. ..    }    }
```

9.3.5.4 *ConfigureDevice*

A class was needed to access and display configuration parameters and allow users to make changes. The attributes and functionality of the *ConfigureDevice* class are described below.

Analysis

- The class implements a servlet to publish the configuration of the IMC in HTML format.
- The IMC configuration parameters such as screw-pitch factor, encoder resolution, etc can be changed by users by means of this servlet.
- The servlet is protected to allow only authorized users to configure the device.

Design

- The class is designed to read configuration parameters from configuration files, present them in HTML format, and save changes to these files to persistent memory.
- The class is structured to extend *com.qindesign.servlet.AuthenticatedHttpServlet* within the Tynamo API to provide security methods.
- The class design overrides the following methods:
  - *doGet* to auto-generate HTML forms and handle *doPost* requests
  - *doPost* to auto-generate HTML forms and handle *doGet* requests
  - *doUnauthorizedGet* to handle unauthorized *HTTP GET* requests.
  - *doUnauthorizedPost* to handle unauthorized *HTTP POST* requests.
  - *getRealm* to get the realm based on the request.
  - *isAuthorized* to check if the given user/password is authorized in the given realm.

Example

Typical configuration browser displays when a client user queries the *ConfigureDevice* servlet are shown in the Figs. 9.2 and 9.3.

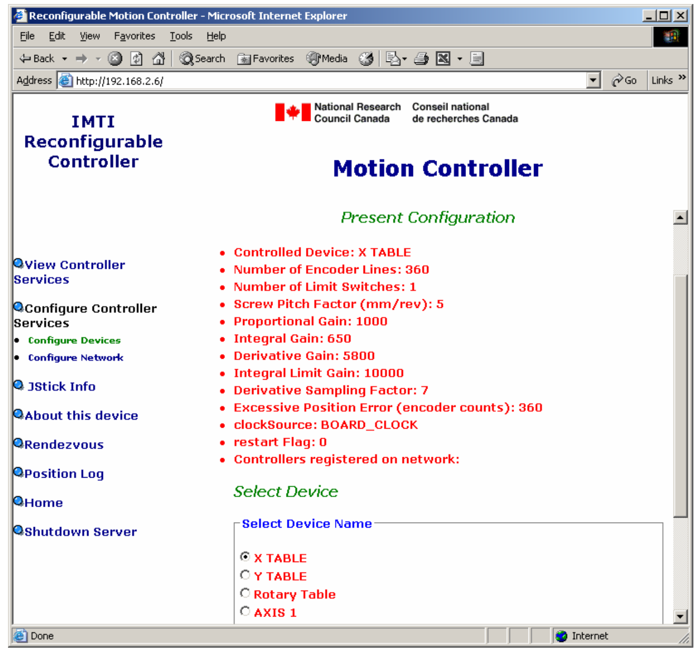


Figure 9.2: IMC Configuration Servlet II

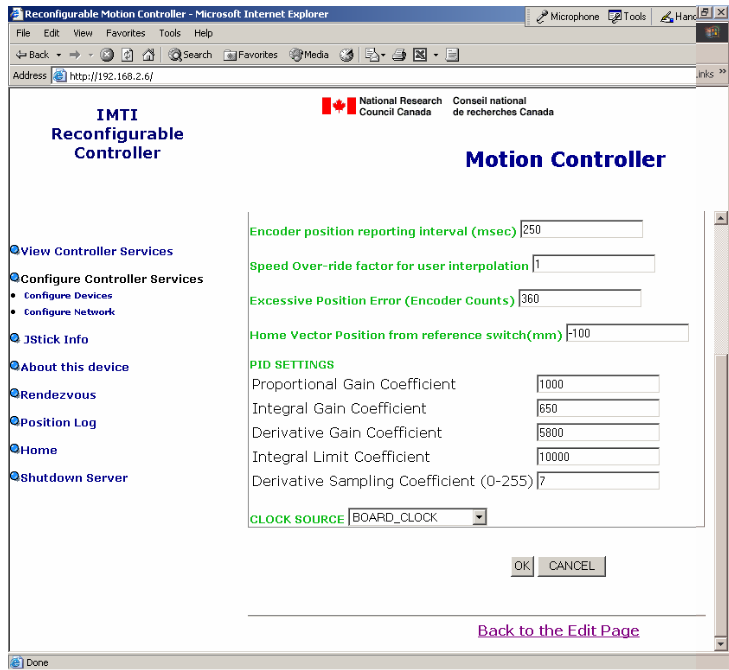


Figure 9.3: IMC Configuration Servlet II

9.3.5.5 EditJmDNS

When a new service is added to an IMC programmatically, and this has to be announced to the network domain, the services have to be included in the list of services to be registered. Similarly, new services to be discovered have to be included in the service discovery list. The class implemented to handle this responsibility is described below.

Analysis

The class has the following attributes and actors:

- The class implements a servlet to publish registered and discovered *JmDNS* services in HTML format.
- An HTML based form is provided for users to edit, delete or add services.

Design

- The class is designed to read *JmDNS* data from a file and present them in HTML format and also save changes to files.
- The class uses collaborative executions of *doPost* and *doGet* to display a hierarchy of HTML forms at the prompt of the user.

Example

Examples of the interfaces generated by the *EditJmDNS* servlet are shown in Fig. 9.4 and 9.5 below.

Figure 9.4: JmDNS Service/Discovery Browser

Figure 9.5: JmDNS Service/Discovery Editor Webpage

9.3.5.6 PositionDump

The servlet technology employed makes it possible to remotely monitor devices on a web browser. This section describes the design of a class for this purpose.

Analysis

The attributes and functionality of the class are as follows:

- Encoder readings are periodically posted to the servlet.
- The servlet posts a history (log) of encoder data.
- The servlet encapsulates readings in HTML format in its *doGet* method.

Design

- Objects of *EncoderReader* and Counter in the *com.IMC.coordination* package are created to capture real-time and logged encoder data.
- Encoder readings are wrapped in HTML format in the servlet's *doGet* method.

9.4 The System Coordinator Domain

The System Coordinator domain handles all supervisory activities to guarantee the appropriate execution of tasks on the IMC controllers. All high-level tasks and commands are generated in this domain. These include human-machine interactions, system configuration, and Meta tasks such as "complex" inverse kinematics, which cannot be

handled by the IMC nodes or the real-time coordinator. Table 9.7 shows the packages in this domain.

Table 9.7: System Coordinator Software Packages

Package	Description
com.coordinator.GUI	Abstracts the Human-Machine Interface
com.coordinator.coordination	Protocols for commanding and coordinating activities
com.coordinator.database	Temporary and Permanent Global Repository
com. coordinator.interpolation	Abstracts interpolation and kinematics algorithms
com. coordinator.network	Abstracts protocols for communication

9.4.1 The Graphical User Interface (GUI)

The GUI component contains programs which construct graphical interfaces for viewing, editing and sending commands to the IMC hosts. Table 9.8 lists the sub-components in this package and brief descriptions. The interfaces are outlined in Appendix C4.

Table 9.8: GUI Component

GUI Component	Description
MainApplication	Main program for starting the User Interface
MainGUIFrame	Presents the main GUI
TrajDataFrame	Constructs the GUI for viewing and editing motion parameters
TrajTable	Displays a table for constructing trajectories
PIDTable	Presents a table for viewing and editing PID parameters.

9.4.1.1 MainApplication

Since there can be only one "main method" in a Java application, the implementing class becomes the trigger point for the execution of the entire application. This section describes the design of the class starter.

Analysis

The attributes and actors of the *MainApplication* are as follows:

- The *MainApplication* starts all network applications and a *JmDNS* implementation to discover and register services on the network.
- This class also instantiates the main graphical user frame.

Design

- The *MainApplication* constructor is designed to create objects of classes encapsulating network connectionless protocols such as *DatagramSender* and *MulticastSender*. At the time of the execution of this class, it is assumed that the connection parameters of the IMC hosts are not known.
- The *JmDNS* coordinator is instantiated in this class before its lifecycle ends.

9.4.1.2 MainGUIFrame

A graphical user frame provides a user-friendly man-machine interaction. This section describes the main GUI frame for this purpose.

Analysis

The features and functionality of the class are as follows:

- The main GUI provides a self-explanatory graphical frame for users to explore the various functionalities of the system.
- The graphics show all registered IMC devices and graphical methods to logon to their web servers.
- The graphical frame shows real-time encoder readings sent by the registered IMC devices.
- The graphics provide tools and links to send control signals and parameters and also shutdown specific or all IMC devices.

Design

- The class is designed with Java graphical API tools in the *javax.swing* and *java.awt* packages and layout features from the Borland JBuilder's, *com.borland.jbcl.layout* package (JBuilder, 2005).
- The interface amalgamates all registered IMC hosts into one console, thus emulating a centralized system.
- The design provides links to other graphical tools to edit or create motion commands.

Implementation

The diagram (Fig. 9.6) below illustrates the main GUI browser.

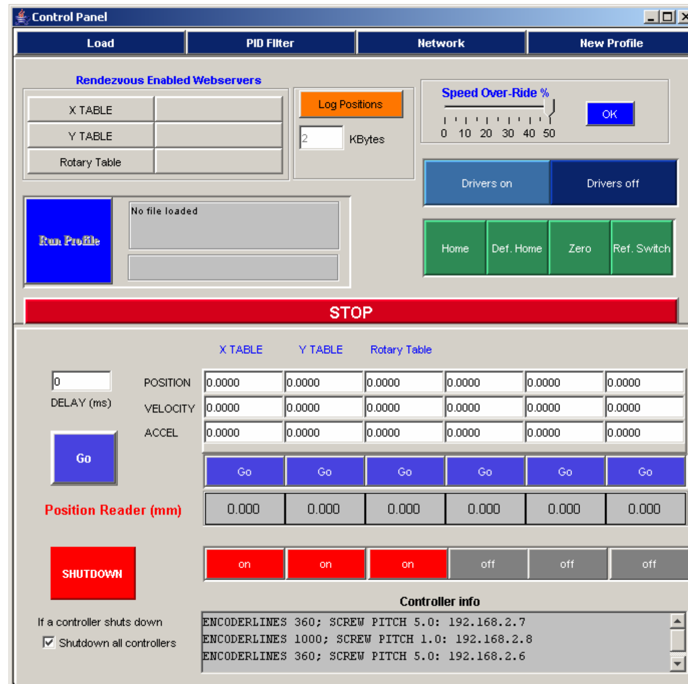


Figure 9.6: Main GUI Browser

9.4.1.3 TrajDataFrame

The *TrajDataFrame* class provides a graphical tool for editing motion trajectory parameters from saved files or creating new trajectories.

Analysis

The *TrajDataFrame* class embodies the following features and attributes.

- The class projects a user-friendly interactive environment for selecting the trajectory type, configuration, and mode of interaction with the IMC hosts.
- The class provides an editor for creating or editing motion trajectory or commands.

Design

- The design of the class is based on Java graphical API tools.
- The class reads or writes user-provided data to the system coordinator's database.

Implementation

An example of a graphical frame generated by the class is shown in the figures below. Figure 9.7 shows the configuration window for NC Code. Figure 9.8 illustrates trajectory set-points including via-points generated offline by the user.

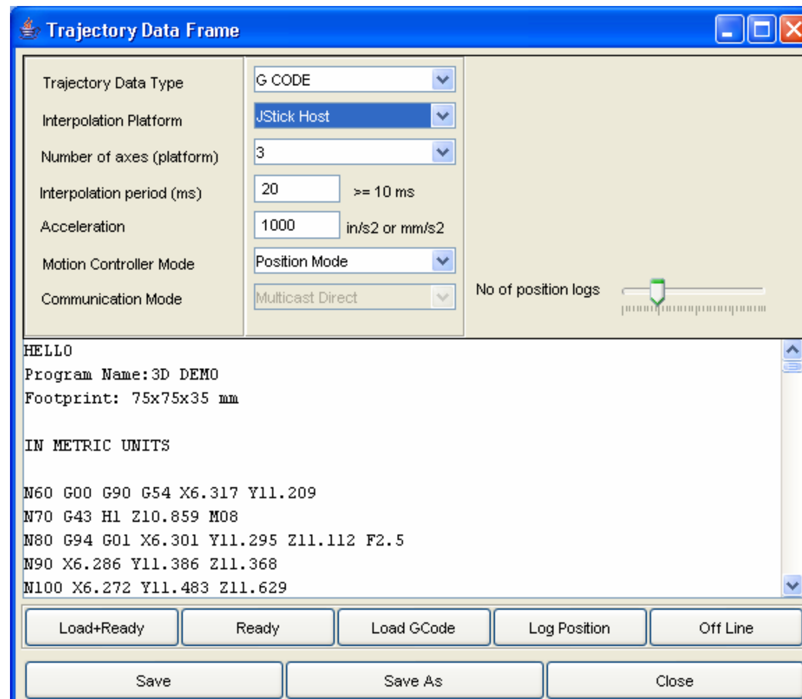


Figure 9.7: Trajectory Editor I

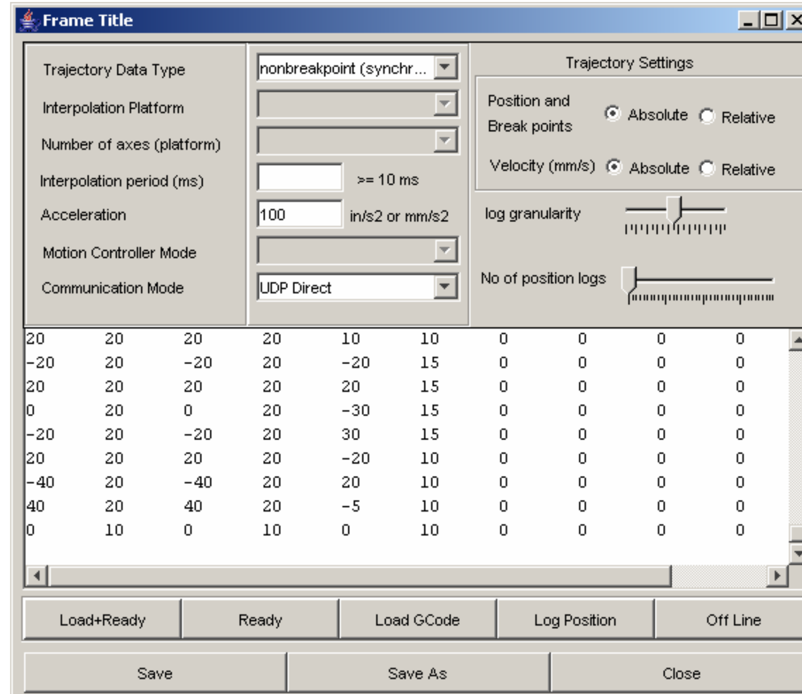


Figure 9.8: Trajectory Data and Configuration Browsers

9.4.1.4 *TrajTable and PIDTable*

The *TrajTable* and *PIDTable* classes have similar attributes; both add methods from the *javax.swing.table* package to standard graphical methods described above to generate editable tables. *TrajTable* may be used to create new trajectories for position, velocity and synchronized position modes. Figure 9.9 shows a trajectory table for creating motion profiles for up to six end-device axes. *PIDTable* on the other hand may be used to generate new PID filter values for the IMC devices. Figure 9.10 illustrates the graphical table for editing PID settings.

Figure 9.9: Trajectory Editor Frames

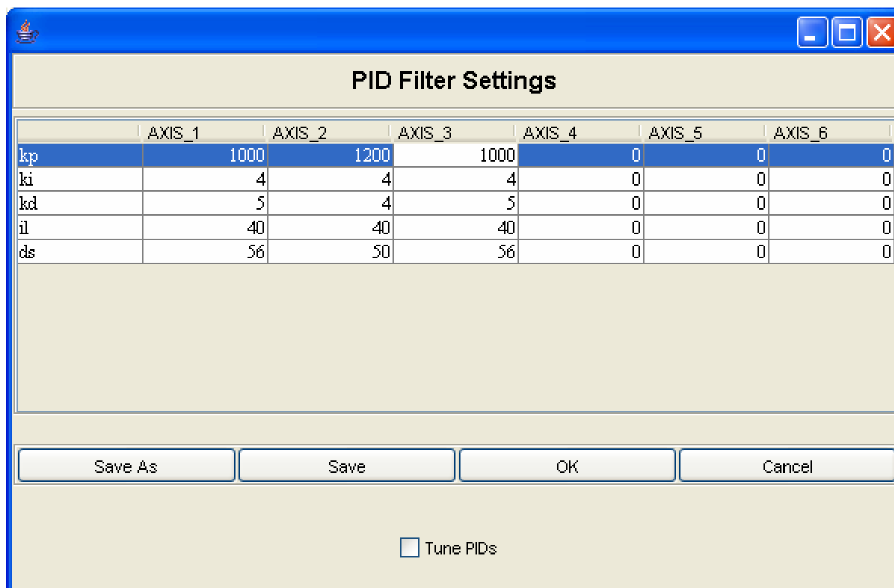


Figure 9.10: PID Filter Parameter Editor

9.4.2 Coordination Abstractions

The coordination component mirrors that of the IMC nodes. The sub-components in this package are listed in Table 9.9 and the interfaces are provided in Appendix C2.

Table 9.9: System-Coordinator Coordination Components

coordination Component	Description
JmDNS_Coordinator	Implements JmDNS to register services and discover specific services
JmDNS_Event_Server	JmDNS_Coordinator calls this class to service events received
ControllerIO	Relays user commands in the GUI to the communication layer
Monitor	Receives IMC sensor data and messages by TCP
SynchFlag	Coordinates trajectory in synchronized-position mode
Trajectory_Server	Streams trajectory commands to IMC hosts

9.4.2.1 JmDNS_Coordinator and JmDNS_Event_Server

The *JmDNS* coordinator collaborates with *JmDNS* counterparts running on the IMC to create a unified system configuration with no user-intervention. At the start of execution, the system coordinator makes no assumption about the network configurations and services provided by the IMC hosts. It multicasts its own service parameters and executes a listener for IMC services. When service parameters are received, encapsulated network data such as IP addresses are extracted and utilized for connections with the IMC hosts.

Analysis

The functionality and attributes of the classes are as follows:

- The *JmDNS_Coordinator* class uses *JmDNS* to register the system coordinator on the network.
- The *JmDNS_Coordinator* class calls *JmDNS_Event_Server* to fire its listeners to detect incoming *JmDNS* events on the network and service them accordingly.

Design

- Like in the case of the IMC, the *JmDNS_Coordinator* and *JmDNS_Event_Server* are designed to implement *JmDNS* to register and discover services respectively.
- The *JmDNS_Event_Server* is designed to service the following events:

- o When an IMC controller service is detected, the device name is posted to its spot on the main graphical user interface and its IP configuration is mailed to a database. Subsequently, a client TCP thread is activated to connect with its TCP server.
  - o When an IMC web service is detected, a corresponding “web” button is enabled on the user interface to allow the user to access its web-pages.
  - o IMC multicast information detected are mailed to the database.
- IMC devices disconnected from the network are removed automatically.
- The *JmDNS_Event_Server* listener thread is kept alive to detect service changes on the network.

9.4.2.2 ControllerIO

The *ControllerIO* class interprets signals and commands from the graphical user interface and prepares them to be sent to the IMC hosts through the communication interface. The *ControllerIO* therefore plays the role of a resource controller.

Analysis

The attributes and functionalities of the class are enumerated below.

- The class transforms all real-world representations to forms which can be understood by the IMC hosts. For example, commands like “run”, “home position” make sense in the concrete world but not in the IMC realm.
- The class connects or matches commands and data to the appropriate communication channel.

Design

The class is designed as follows:

- During the controller commissioning process, the system coordinator’s database is rehashed with connection and command information about the IMC subsystem. The IMC names are used to retrieve specific command keys.
- The class is designed to inherit a datagram client socket to connect with the various IMC datagram servers.

Implementation

The following is an example of how a jog command is sent to the controllers. The *jog* method below is called when a jog button is clicked in the graphical user interface. For example, the jog button for AXIS 1 executes the method following which in turn calls the corresponding method in the *ControllerIO* class;

```
1.   void axis_1Go_actionPerformed(ActionEvent e) {
2.        controllerIO.jog("AXIS_1", "run");
3.     }
4.   public class ControllerIO extends DatagramSender{
5.   .. .. ..
6.   public void jog(String device, String flag) {
7.
8.      for (int i = 0; i < DEVICES.length; i++) {
9.        try {
10.         if (device.equals(DEVICES[i]) | device.equals("ALL")) {
11.           trajectory(AXIS_TRAJ[i][0], AXIS_TRAJ[i][1], AXIS_TRAJ[i][2],
12.                      getDeviceIP(DEVICES[i]),
13.                      getDevicePort(DEVICES[i]), getControlFlag(flag));
14.         }
15.       }
16.       catch (Exception ex) { }   }}
17.     .. .. }
```

The method *getControlFlag(flag))* above is invoked from *Data.class* to retrieve the command key from a hash-table.

9.4.2.3 Monitor

The *Monitor* class is the client version of the *Monitor* class on the IMC hosts. A Monitor thread is created when an IMC is detected on the network. The thread then makes a request for a TCP connection with the server in order to receive emergency signals, encoder readings and synch signals.

Analysis

The *Monitor* class has the following attributes and functionality:

- The class implements a TCP client socket communication.
- Encoder readings are made available to the graphical user interface and any high level controller implemented.

Design

The following describes the design of the *Monitor*.class.

- The Monitor class derives a client communication socket from *TCP_Client* class in the *com.coordinator.network* package and is implemented as a thread.

- Messages received are channelled to the data repository, graphical display or controller depending on the value of the flag prefixed to the data.

Examples

The example below shows how the *Monitor* thread coordinates synchronized-position-mode motion within its thread body.

```
1.  public void run() {
2.  .. ..
3.  switch (readFlag) {
4.  case 1:   {
5.  //increment synch counter when a trajectory-end flag is received
6.             Synch.put_flag(1);
7.  //if number of flags = number of threads (IMC hosts) send run command
8.              if (Synch.get_flag() % Synch.get_threads() == 0) {
9.               ControllerIO.setMulticast("run");    }
10.            break;          } } }
```

The *Synch* class is used as a repository for flags received from all *Monitor* threads during this type of motion s follows;

```
1.  public synchronized void put_flag(int i) {
2.     synch = synch +i;
3.      notifyAll();
4.    }
5.  //When a Monitor thread is created a thread-counter is incremented in the Synch class
6.     public synchronized void put_threads(int i) {
7.    threads += i;      //thread counter
8.     notifyAll();
9.   }
10. //Method returns the number of synch flags received
11. public synchronized int get_flag() {
12.    return synch;
13.   }
14. //Method returns the number of Monitor threads
15. public synchronized int get_threads() {
16.    return threads;
17.   }
```

9.4.2.4 Trajectory_Server

The *Trajectory_server* class communicates trajectory data to the *StateCoordinator* of each IMC.

Analysis

The attributes and functionalities of the class are as follows:

- Large chunks of trajectory data are sliced into conveniently small pieces, packaged in datagram packets and sent over the network sequentially.

- The class negotiates with the IMC coordinators in order to ensure the integrity of data congruity.

Design

- The class is designed to have a handshake policy with each IMC. Therefore one class thread is created for each IMC. When a packet is sent, the thread switches to a blocking receive-state to receive a ready signal from the IMC. The next packet is then sent. This cycle continues until all the data is sent or the process is interrupted by a high level command.

9.4.3 The Database Abstraction

The database abstracts sub-component which are designed to store persistent and most transient data. Table 9.10 lists the sub-components and their descriptions. Their respective interfaces are outlined in Appendix C1.

Table 9.10: System Coordinator Database Sub-Components

Database Component	Description
Data	Network, configuration and program data and access mechanisms
JmDNS_DATA	Repository for JmDNS services
Traj_Configuration_Data	Stores trajectory configuration data
GCodeParser	Parses NC code into machine readable format
GCodeSender	Sends NC code to the real-time interpolator

9.4.3.1 Data

The analysis and design of the Data class is described in this section.

Analysis

- The *Data* class stores all persistent data except trajectory and *JmDNS* data.
- The class contains robust mechanisms to store and retrieve data, even when changes are unanticipated.

Design

- The *Data* class is designed to use the static modifier for all persistent data.
- System commands and IMC information are placed in hash-tables and retrieved with keys. For example, when a device is registered, its IP address is put in an IP *hashtable*

and the device name becomes the key to retrieving the address. Since this credential is not hard-coded, changes have no ramification on calling methods.

Example

An example of a method in the Data class is illustrated below;

```
1.  public static void putDeviceIPInfo(String deviceName, String IP_Address, int port)
2.   throws Exception {
3.         hashIP.put(deviceName, IP_Address);
4.         hashPort.put(deviceName,new Integer(port));
5.    }
```

9.4.3.2 JmDNS_DATA and Traj_Configuration_Data

All *JmDNS* data to be registered and discovered are stored in the *JmDNS_DATA* class as arrays. The *JmDNS* coordinator extends this class to get access to this data. The *Traj_Configuration_Data* class handles user-defined trajectory configuration parameters such as acceleration, interpolation periods, etc. These are held in arrays and stored in a configuration file defined by the user.

9.4.3.3 GCodeParser

NC machine code is usually written in a text format. The *GCodeParser* performs a syntax check on an NC G code file and transforms it into a data structure, which is suitable for an interpolator. This section describes the design of the parser.

Analysis

The attributes and functionalities of the *GCodeParser* are as follows:

- The class reads a typical G code file and eliminates all non-motion related ASCII characters and comments.
- Meaningful tokens are sequenced and end of command lines are properly delimited.
- The results are saved in binary format.

Design

- The class is designed to use a combination of a file read pointer (*DataInputStream.readLine*) and a read marker (*DataInputStream.mark*) to search from the beginning of the file to the first command line, i.e., the first occurrence of the character *N, n, G*, or *g* followed by two digits. When the command line is detected, the marker places the read pointer at the beginning of this line.

- The G code is tokenized by this technique; the G commands are logically grouped into motion commands (G00, G01, G02, G03, F, etc), G commands (G52, G91, etc), and coordinates and federates (e.g., X2, F2, etc). When a motion command is read, it is buffered and prefixed to every coordinate. When another motion command is read, the previous command is replaced by the current, and so on.
- All white spaces are removed and end-of-lines are delimited. The file is saved in ASCII binary big-endian format.

Example

The following typical G code for linear interpolation,

1. *NRC: Footprint: 70x70x30 mm, IN METRIC UNITS*
2. *N60 G00 G90 G54 X6.317 Y11.209*
3. *N80 G94 G01 X6.301 Y11.295 Z11.112 F2.5*
4. *N100 X6.290 Y11.301 Z11.152*

is tokenized into the format below.

1. *g90*
2. *g54*
3. *g0*
4. *x6.317*
5. *y11.209*
6. *g94*
7. *g1*
8. *x6.301*
9. *y11.295*
10. *z11.112*
11. *f2.5*
12. *g1*
13. *x6.290*
14. *y11.301*
15. *z11.152*

9.4.3.4 GCodeSender

The *GCodeSender* is used to send G code commands to the real-time coordinator. It extends the functionality of the *GCodeParser* and in addition calls a TCP client socket.

9.4.4 The Interpolator Component

The component described in this section is essential for all coordinated motion profiles. It is responsible for generating set-points through interpolation and transmitting these set-points. Set-points may be transmitted directly to the IMC hosts or through a real-time coordinator. The *Transmission* and *Transmission_ACK* classes (Table 9.11) are

responsible for the communication transaction. The rest of the section describes the design and implementation of the interpolator.

Table 9.11: System Coordinator Interpolator Sub-Components

Interpolator Component	**Description**
Interpolator	Trajectory interpolation
Transmission	Transmits set-points to controllers or real-time coordinator
Transmission_ACK	Receives acknowledgement from real-time coordinator
Transmission_Flag	Services acknowledgement flags

9.4.4.1 Interpolator

The interpolator design is based on the linear and circular interpolation algorithms discussed in Chapter 7. The original code was written in C++, and later ported to the Java-based IMC architecture for this research. The flow diagram of the Interpolator class is shown in Fig 9.11.

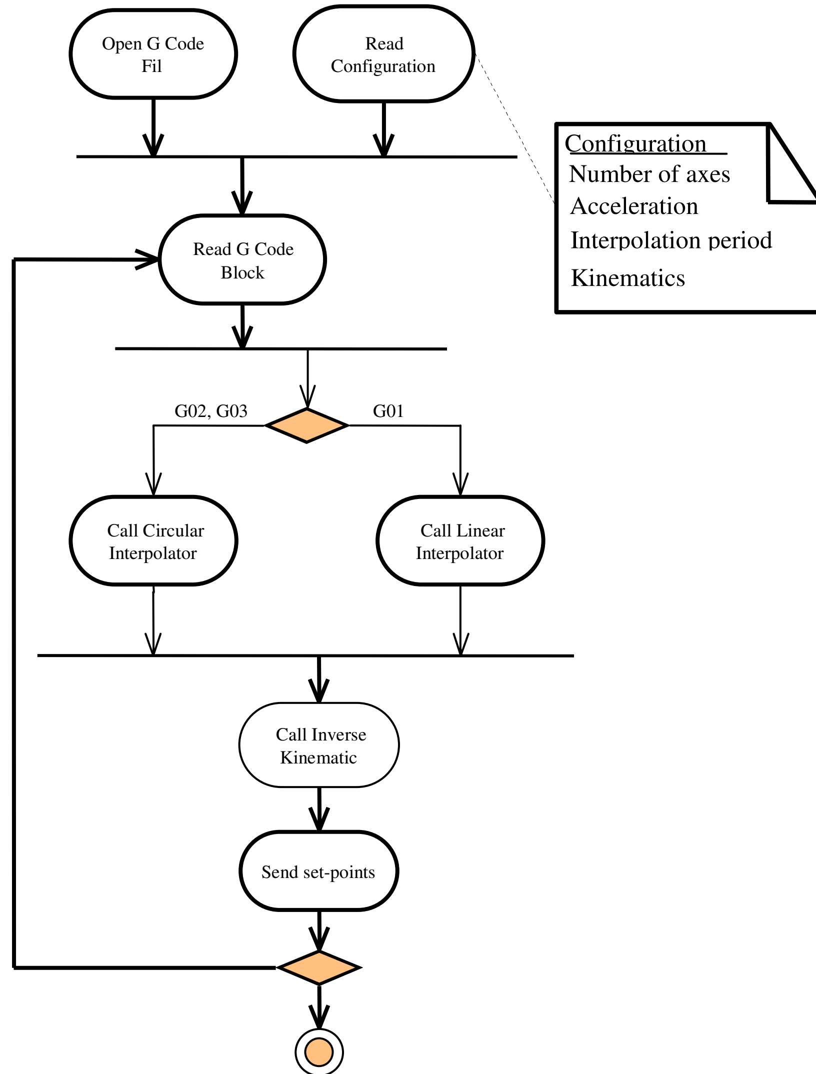


Figure 9.11: Interpolator Flow Diagram

The Interpolator class is instantiated with user-defined configurations. Configurations include, the interpolation minimum period, the number of controlled axes, acceleration/deceleration values, a Boolean to indicate whether motion profile should be ramped (acceleration/deceleration regimes), and an object of the class implementing the inverse kinematics of the controlled device. The system coordinator architecture, which hosts this class, is designed to run on Microsoft Windows XP. However, since Java is platform-independent, it is possible to run the system on other platforms – ideally a real-time operating system. In the case of a real-time environment, set-points can be

transmitted deterministically to the IMC hosts. However, in the absence of such an operating environment, the IMC architecture provides a workaround by dedicating one of the IMC hosts for real-time coordination such as streaming set-points.

9.4.5 Network Abstractions

The network abstraction is the complement of the IMC network abstraction. Hence, communication components abstract client connections from this package to request connections with corresponding servers on the IMC hosts. The abstraction contains other classes to communicate with the real-time coordinator. The list and description of the sub-components of this package are shown in Table 9.12.

Table 9.12: Network Package

Network Components	Description
DatagramSender	Transmits control flags and data to IMC nodes
McastDirect	This is used by Interpolator.class to multicast set-points directly to IMC nodes
MulticastSender	For sending state signals by multicast to nodes
TCP_Client	Creates a TCP/IP data I/O stream socket
UDP_Client	Creates datagram client socket and has methods to transmit/receive data

9.5 The IMC Real-Time Coordinator

The real-time coordinator complements the system coordinator by providing services which the latter may not be able to provide. Table 9.13 shows the packages in the real-time coordinator. Only components in the first package will be discussed since the last two are similar to aforementioned ones.

Table 9.13: Real-Time Coordinator Software Packages

Package	Description
com.RTcoordinator.RTservices	Contains real-time protocol services for coordinating with IMC nodes
Kinematics.NRCTripod	Contains inverse kinematics of the NRC Tripod
com.RTcoordinator.network	A group of standard IP protocols

9.5.1 Real-Time Services

The real-time services package includes abstractions for real-time streaming of data, and clock synchronization. The package contents are listed in Table 9.14. The classes with asterisks bear resemblances with ones already discussed, so a concise description of them will ensue.

Table 9.14: Real-Time Service Components

Real Time Services	Description
CommandReceiver	Receives commands from the system coordinator
*GCodeReceiver**	Receives tokenized G Code from system coordinator and stores values in local directory
*Multicaster**	Multicasts data to IMC nodes
*Interpolator**	Real-time interpolator
*NetworkParameters**	IP parameters
*Network_Analyzer**	Measures network latency
RTReceiver	Receives data streams from System Coordinator
RTDistribute	Streams data in real-time to IMC nodes
TCPReceiver	Base class for TCP connections
TimeStamper	Uses hardware timer to timestamp events

9.5.1.1 *CommandReceiver*

The *CommandReceiver* class establishes a server-client relationship with the system coordinator through which various command flags are received. The command structure is shown in Table 9.15. Descriptions of the analysis and design follow.

Table 9.15: Real-time Coordinator Command Structure

Command Flag	Action to be Taken
0	Suspend interpolation thread
1	Release interpolation thread from suspend-mode
2	Reset/restart interpolation
3	Execute method/class to receive tokenized NC code, and store in persistent file.
4	Prepare interpolation thread to execute locally stored NC code. Interpolation time, number of axes and acceleration value attached to packet.
5	Interpolation values streamed by system coordinator. Prepare threads to receive data for real-time streaming to IMC nodes. Period attached to packet.
6	Start threads in 5 to receive and distribute data
7	Send contents of NC file
8	Invoke method to measure network latency

Analysis

The attributes and functionalities of the *CommandReceiver* class are as follows:

- The class is made as generic as possible to allow reusability on other platforms.
- It responds to the various commands from the system coordinator listed above.

Design

For a generic design, implementation-specific methods are made abstract; the class inherits a datagram socket from the network package, and is implemented as a standard thread with the main loop in its *run()* method. Implementing classes may override hardware-specific methods such as the creation of real-time threads.

Example

The prominent feature of the implementation of this class is the real-time thread creation. The aJile CPU uses a cyclic executive model with preemption for real-time periodic threads with a context-switch of 1 μs. The data structure for this model is a cyclic array which is like a piano roll. The *PianoRoll* constructor provided in the aJile API, establishes the Piano Roll's length (duration) and beat. The cyclic data structure is used by the aJ-100 scheduler to keep the different periodic threads. An index pointer to the structure is incremented for every beat time and if a periodic thread is scheduled for that beat, it is executed. Below shows how the real-time threads for receiving (*receiver* object) and streaming data (*distributor* object) are created.

```
1.  void prepareDataStreamThreads(){
2.  int n = 2;
3.  int k = 0;
4.  int T = period;
5.  int beat = 1 * T;
6.  int duration = n * beat;
7.  int period = 1 * beat;
8.  // max priority of the periodic threads:
9.  int priority = 14;                          // use: 0 <= priority <= 14
10. int receiveroffset = 0;
11. int distributroffset = beat;
12. //setup FIFO buffers
13. InterruptSafeIntFifo buffer = new InterruptSafeIntFifo(num_axes * 5000);
14. InterruptSafeIntFifo periodbuffer = new InterruptSafeIntFifo(5000);
15. receiver = new RTReceiver(period, priority - 1, receiveroffset, buffer,
16. periodbuffer);
17. distributor = new RTDistribute(period, priority, distributroffset,
18. buffer, periodbuffer,msender);
19. pianoRoll = new PianoRoll(duration, beat);
20. }
```

The network is analyzed once for a particular configuration.

9.5.1.2 GCodeReceiver

This class is instantiated to receive tokenized NC code from the coordinator via a TCP/IP connection. The data stream from the connection is wrapped in a file output stream for simultaneous data transfer to a created file.

9.5.1.3 Interpolator, Multicaster and Network_Analyzer

The *Interpolator* is similar to that of the coordinator, except that it is scheduled for periodic real-time executions. The *Multicaster* class is invoked by the interpolator to package and multicast data to the IMC nodes. The *Network_Analyzer* uses a datagram socket connection to carry out several request-response transactions with each IMC nodes. The average latency is then deduced and used to correct periodic schedules.

9.6 Conclusion

Details of the IMC software architecture have been presented in this chapter. An object-oriented architectural style based on Java is used in order to benefit from the inherent modular component properties this style provides. The architecture is implemented on the three separate domains of the IMC architecture; i.e., the IMC nodes, the system coordinator, and the real-time coordinator. Analysis, design, and implementation details are provided for the components of each domain. The architecture is designed from the onset to be modular, reusable, and easy to maintain. Hence component-style techniques are used such as separation of hardware-specific or implementation-specific components from generic components. Since Java is platform-independent, with little modification, the implementation can run on different computing platforms.

10. SYSTEM EVALUATION

10.1 Introduction

System evaluation (and fine-tuning) is a very broad exercise which could pass for another major research projects. Therefore, this chapter provides a brief evaluation based on CNC requirements. This is one of the most stringent areas of application for IMC architecture. A demonstration on a parallel kinematic machine is also illustrated.

Machine tools may be classified according to their time period for evaluating and generating axes commands. This period is commonly referred to as the servo cycle period. High performance controllers have servo cycle periods of less than 1 ms, while that of medium performance controllers lie between 1 ms and 7 ms. Low performance controllers have servo cycles greater than 7 ms. Most CNC machines for metal cutting fall within the medium to high categories (Bellini, et al., 2003). In addition, a controller's block processing time reveals its ability to process geometric entities. For general machining, 10 ms (or less) is adequate. In contrast, high-speed machining requires a block processing time of less than 2 ms. In view of these requirements, the distributed reconfigurable controller is evaluated based on the following criteria:

1. Servo loop cycle time.
2. Communication latency.
3. Block processing time and real-timeliness.
4. Synchronicity and positioning accuracy.
5. Architectural flexibility.

10.2 Sampling Time and Communication Latency

The PID motion controller chip (LM628) on the IMC performs high speed trajectory generation (including ramping and slewing) at a maximum sampling speed of 341 µs. This is the last and finest interpolation leading to commanding the axes drives. At this speed, trajectory calculation takes 120 µs, PID filter calculation adds 66 µs delay, and sample output latency is 13.3 µs. However, the output integrity of the motion chip may

deteriorate if its trajectory generator is updated (externally) faster than 10 ms. The JSticks which serve as the host to the motion control board communicates with the LM628 chip via its 8-bit high speed I/O (HSIO). The HSIO's timing is reconfigurable enabling us to clock down from its maximum speed of 38.8 ns to 77.5 ns in order to meet the timing requirements of the motion control chip. It takes five 8-bit words to command and load up a set of trajectory parameters (velocity and position) into the LM628's buffers. Considering read/write delays, the total execution time for this is at most 500 ns. It is therefore technically feasible for one JStick to host more than one LM628. At the moment only one JStick is mapped to each of our motion boards. The hosts receive set points from the interpolator every 10 ms and propagate them to the motion chips accordingly. Communication is via Ethernet's multicast protocol (UDP/IP). Since UDP has an overhead of only 8 bytes, communication is very fast. With a payload of 8 bytes per each set-point and a bandwidth of 10 Mbps, it takes theoretically 1.6 μs for a packet to reach the communication interface of an IMC node. Accounting for jitter and processing time, the maximum latency is approximately 4 μs.

10.3 Block Processing Time and Real-Timeliness

The interpolator runs on a computing platform (JStick) dedicated for real-time coordination. The JStick platform executes real-time threads in a cyclic deterministic manner. Thread context switch takes only 1 μs, thereby providing acceptable real-time outputs. ISO G-code programs are loaded to the interpolator's flash memory by Ethernet from a GUI program running on a regular PC. The JStick's flash speed is 90 ns and its execution speed is 15 Mega bytecodes/s. To measure the average block processing speed, G1 and G2 codes were executed on the platform. An average time of 1.5 ms was obtained for 3-axis linear interpolation, and 2.0 ms for 2-axis circular interpolation. However, the block processing time is constrained by the hardware limitation of the motion controller chip to 10 ms as mentioned in the previous section.

10.4 Synchronicity

Synchronization is a critical issue with decentralized controllers. Three different timing analyses were conducted on a 3-axis machine with the aid of a logic analyzer. Each controller toggled one of its I/O pins just after executing its trajectory commands. The

respective pin activity timings were logged by the logic analyzer. In the first test case, there was no compensation for timing variations caused by network and processing delays. The worst case delay, as shown in Fig. 10.1 was about 1 ms, and over 20% of the data fell in this region. In the next test case, the clock synchronization scheme developed for the architecture (see Chapter 7) was implemented. The worst case delay fell appreciably to 0.1 ms (Fig. 10.2). For even finer synchronization, the real-time coordinator was hardwired to the IMC controllers by an interrupt line. Thus the sequence of events was driven by interrupts from the real-time coordinator. The delay results are illustrated in Fig. 10.3. Delays are highly repeatable in this case, but this configuration is not a true distributed approach. This is illustrated to show the versatility of the architecture.

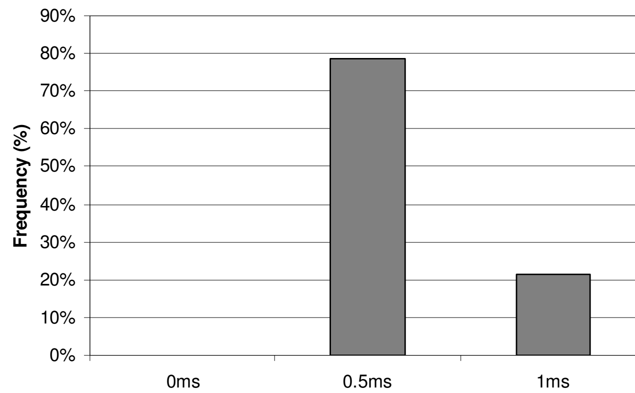


Figure 10.1: Timing Variations-Uncompensated Delays

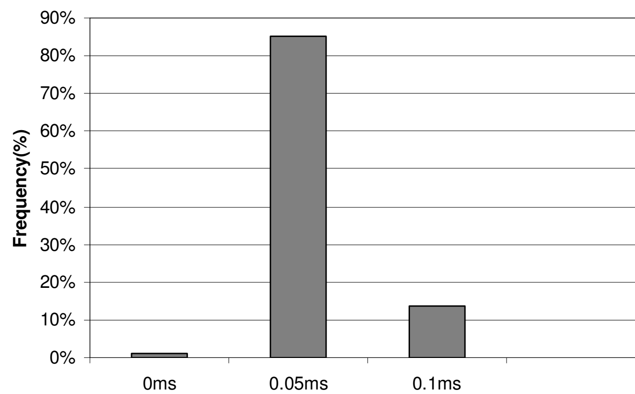


Figure 10.2: Timing Variations- Compensated Delays

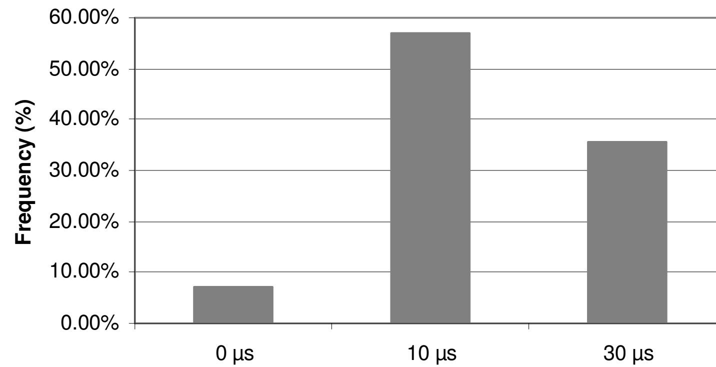


Figure 10.3: Timing Variation- Synchronization by Interrupts

10.5 Positioning Accuracy

Some position measurements are presented below to illustrate the performance of the controller on a 3-axis machine. The machine is equipped with d.c. motors directly connected to 5-mm pitch feed-screws and 360-line encoders (360 × 4 counts/rev). Thus the BLU (Basic Length Unit) is approximately 0.003 mm. Figure 10.4 shows a zigzag pattern move realized with a feed-rate of 900 mm/s. Figure 10.5 illustrates circular interpolation with a feed-rate of 350 mm/s and a radius of 5 mm (1440 counts). The trajectory errors associated with a 1-mm radius (288 counts), 5-mm radius and 25-mm radius circular paths are shown in Figs. 10.6 to 10. 8.

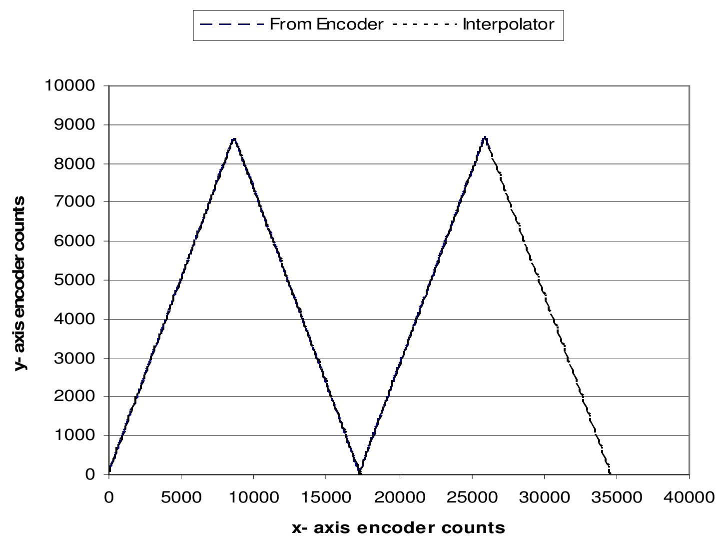


Figure 10.4: Linear Trajectory

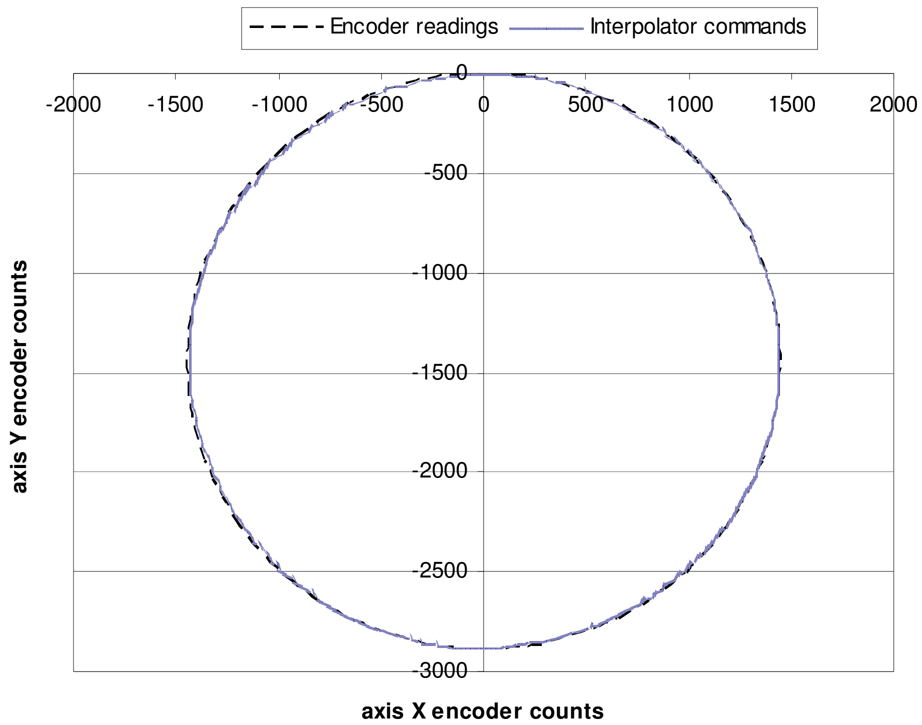


Figure 10.5: Circular Trajectory – 5-mm radius

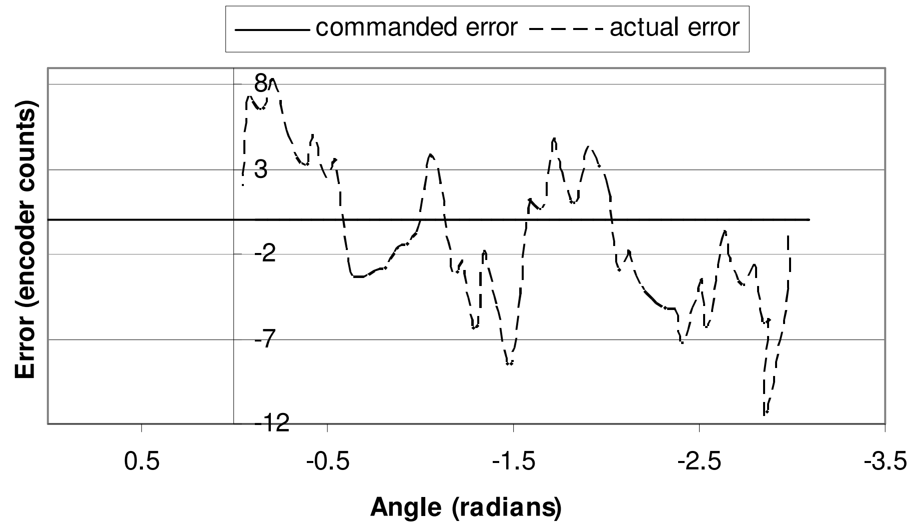


Figure 10.6: Radial Error – 1-mm radius

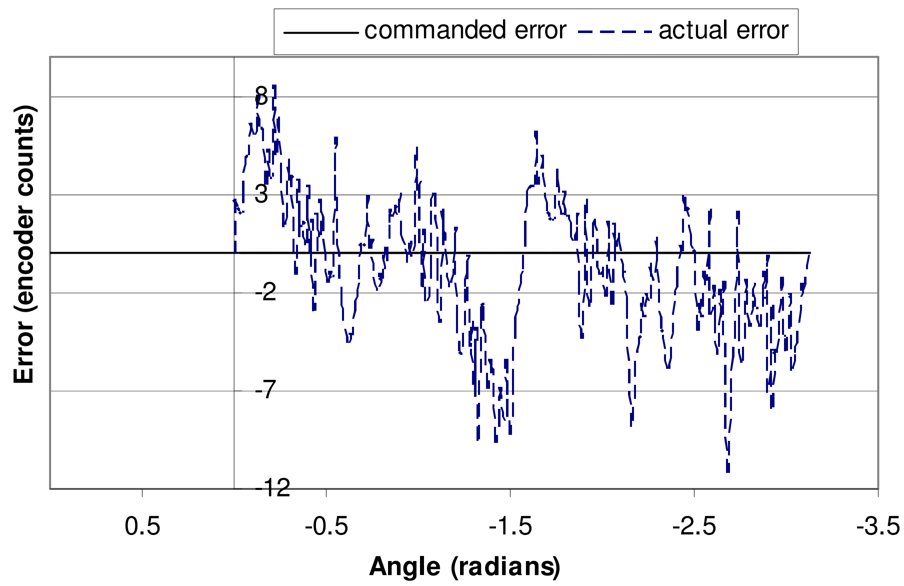


Figure 10.7: Radial Error – 5-mm radius

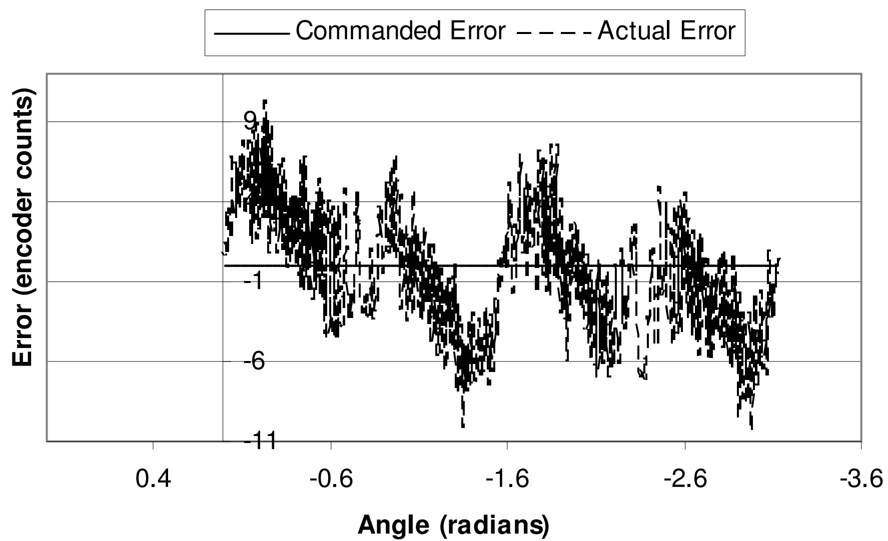


Figure 10.8: Radial Error – 25-mm radius

The error patterns are very similar and bound by a maximum of about 11 counts. Dry friction in the mechanical drives and interpolation approximations contribute significantly to these errors. Blended moves are also very acceptable. In Fig 10.9, two

linear and one circular interpolations followed by another linear interpolation are executed in one trajectory path.

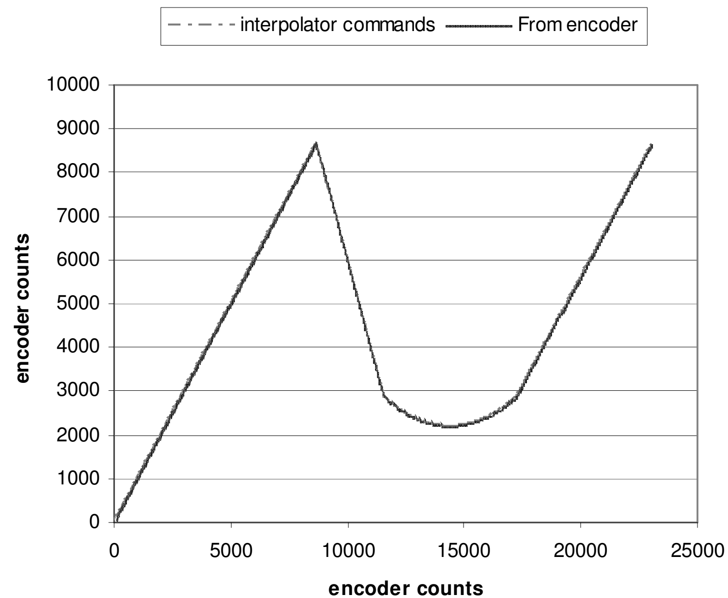


Figure 10.9: Combined Linear and Circular Paths

10.6 Architectural Flexibility

By virtue of the intelligent communication protocols and the modular nature of our hardware and software architecture, flexibility is greatly enhanced. When machine (or workstation) configurations such as the number of axes need to be scaled, control modules are added to the network to match the number of axes. The protocols described in Chapter 6 automatically configure the network for such changes. The modular nature of the software architecture makes it easy to add modules to accommodate changes. For example, through software interfacing new algorithms for kinematics or interpolation can be readily integrated. Figure 10.10 illustrates the simulation of the guide (slider) movements of a parallel kinematic mechanism on a 3-axis table.

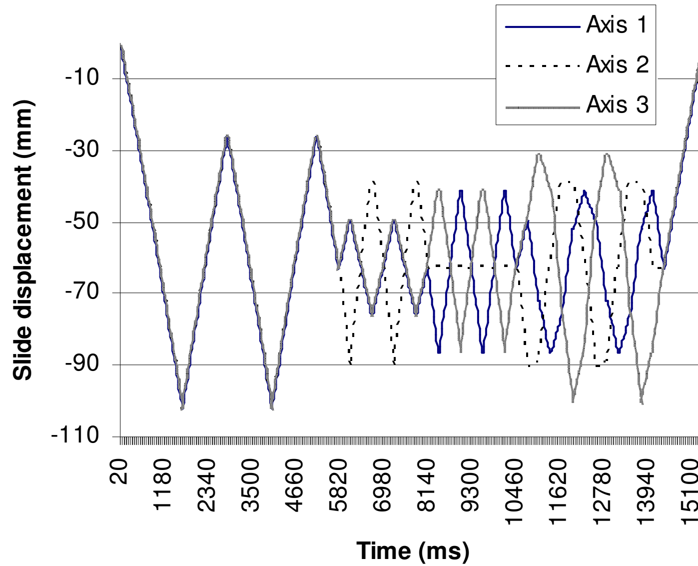


Figure 10.10: Tripod Slider Displacements

10.7 Conclusion

The architecture and development of a real-time modular controller architecture for easy reconfiguration has been described and analyzed. The work demonstrates the use of COTS components for cost-effectiveness and easy integration. The architecture allows devices to discover themselves and search for services or broadcast their services using the JmDNS protocol described in Chapter 5. The architecture may also be changed during initialization for optimal performance depending on the decision of a master controller. This broadens its range of application to other devices such as serial robots. Performance results show a close match with medium performance controllers while providing rich support for network connectivity.

11. CONCLUSION AND DISCUSSIONS

11.1 Overview

This research has presented the design and implementation of a modular controller for robot applications. The work progressed along two main directions. First, the field of controller architectures was reviewed to determine the necessary philosophy and components for a good design. Moreover, the requirements vis-à-vis shortcomings of current designs were investigated. Second, a controller was designed based on modular distributed concepts, off-the-shelf components, and reconfigurability. The design process itself was a little more chaotic than it appears in this thesis. In retrospect, most of the intermediary steps in the course of this project were reached before the requirement analysis was exhausted. This rationale, also called rapid prototyping in some literature, is quite suitable when dealing with a general problem space that is hard to model analytically or has not been previously investigated. Quite certainly, many system designers will intimate the same sentiments after attempting to mesh together seemingly conflicting goals. Nonetheless, a good design is an inherently creative activity with a sense of continuing finality.

A summary of the thesis chapters is provided in the following. In addition, the major contributions are underscored. Finally, recommendations for future research directions in the area of reconfigurable controller design are provided.

11.2 Summary of Results

The overall objective of this research was to design and implement a modular reconfigurable controller prototype based on COTS components. The research objectives were outlined in Chapter 1 and are repeated in summary form in Table 11.1.

Table 11.1: Summary of Research Objectives

Primary Objectives
• Design a generic framework for a modular reconfigurable control architecture.
• A simplistic design that fits into embedded low-cost systems.
• A working prototype.
Secondary Objectives
A critical review of the state-of-the-art in control architecture, distributed communication paradigms, and reconfigurable networked systems.
A synchronization algorithm and protocols to enable Ethernet to be used for real-time control.
An operational software architecture based on modularity and reusability.
Implementation and evaluation of the strengths of the prototype.

11.2.1 Control Architecture Review Summary

Chapter 1 emphasized the need for a well-conceived architecture and provided key issues for designing control architectures. A broad review of some state-of-the-art software and hardware architectures was provided. The review unveiled the ongoing research of designing controller architectures with superior flexibility and greater robustness to obsolescence than traditional ones. Certain concepts stood out in the discussion as the key enablers of this vision. First modularity should be pervasive throughout the design, in order to relax many constraints associated with traditional controllers such as complexity. Among the many hardware and software approaches, modularity based on off-the-shelf components and object-oriented techniques apparently have many advantages. Second the architecture should possess a well-designed communication framework that can support configuration changes. There is a dichotomy between enhanced flexibility and performance. Therefore, this necessitates a well-designed communication architecture to provide a good balance. Third, intelligence should not be limited to the upper layers of a controller architecture hierarchy but distributed as much as possible – ultimately to sensors and actuators. This approach improves flexibility, fault-tolerance, and versatility.

11.2.2 Recapitulation of the IMC Architecture

In Chapters 3 through 9, the IMC controller architecture was conceived and developed. The architecture defined the functionalities of various levels of a distributed layered-model. The concept also proposed embedding or dedicating a controller for each machine axis, a loosely-coupled coordination scheme that can auto-configure to tight-coupling depending on the need, and a network system that can accommodate these requirements. Below is a summary of the salient properties of the architecture.

- A distributed architecture based on a reference (abstract) architecture that defines the various hierarchical decompositions.
- Loosely-coupled architectural elements (software and hardware) for easy system development and flexibility. This is primarily enabled by a modular hardware design, networked elements, and an object-oriented software architectural style.
- Controller elements or nodes are designed to be plug-and-playable.
- Cooperation protocols were developed to separate real-time from non real-time data flows.
- A clock synchronization model was developed to provide a global time base.

11.2.3 Prototype Development Summary

In Chapter 4, the strengths and weaknesses of Java for real-time control were discussed. It was realized that Java potentially has several advantages over traditional C++ and Ada approaches. The main weakness of Java is that it was not originally designed to service real-time systems, hence its relatively slow and non-deterministic response. However, many measures are currently under way to develop real-time Java platforms. Of particular interest to us is the advent of COTS-embedded Java hardware processors which promise real-time computation. Consequently, in Chapter 5, a combined COTS Java-processor based microcontroller (JStick) and motion controller boards were developed for each machine axis and code-named IMC (Intelligent Modular Controller). Details of the design procedure and timing analyses were discussed in this chapter.

11.2.4 Summary of the IMC Communication Architecture

Traditionally, Ethernet is used for non real-time communication because of the non-deterministic nature of its underlying protocols. Nonetheless, it provides superior

robustness to almost all other network communication standards. In Chapters 6, a case for using Ethernet for real-time communication was presented. The primary reason was to address the need to propagate this standard communication mechanism further to the controller arena. Currently, this field is inundated by keen competition between rival fieldbuses. Therefore, a typical industrial communication system (from enterprise level to field devices) needs several mediator or middleware devices to create a virtually homogenous environment. Obviously, this technology aggravates the complexity of the system. In order to create a truly homogenous environment, a flexible Ethernet-based real-time communication architecture with implicit clock synchronization was developed for the IMC architecture. The following were the enabling concepts and technologies.

- An automatic configuration protocol based on JmDNS was developed to run on each node. The protocol enables nodes to automatically subscribe for services they need and also publish their own services
- A switched-Ethernet was used to segment the network and create one collision domain per switch port.
- There are two main execution flows which run concurrently. A client-server cooperation scheme is used for non real-time communication over the network, particularly monitoring operations. In contrast, a producer-consumer model is used for real-time communication. In addition, the latter provides greater flexibility than the former.
- A dedicated real-time coordinator module was created to schedule hard-real time tasks for the IMC nodes in a static cyclic periodic manner (time-triggered computations).
- External clock synchronization is not a singular activity but is concurrent with regular cyclic communication flows.

11.2.5 Summary of Computational and Software Models

Chapter 8 provided an overview of trajectory generation schemes for different mechanical platforms and the execution methodology developed for the architecture. The discussions focused on algorithms that can be decomposed according to the number of machine axes. Two fundamental interpolation schemes developed for the IMC system were linear and circular interpolations. In Chapter 9, the development process of the IMC

object-oriented software architecture was discussed. The architecture was designed and implemented on a conceptual framework of reusability and modularity, and this was empowered by component building blocks. The rationale behind this was to realize a system that is easy to maintain and configure for different end-devices. The development of components consisted of three major steps executed in a back and forth manner. First, the problem domain and sub-domains were analysed. This resulted in a set of entities (components) in the form of classes or objects, the relationship between these entities, and their functionalities. The next step was the design phase, where decisions were made based on the execution platform, the programming language and the operational constraints. The third step was the software implementation. The entire framework consisted of the IMC (axis-controller) domain, the real-time coordinator domain, and system coordinator domain.

11.2.6 Summary of the Controller Performance

An evaluation of the performance of the controller architecture was presented in Chapter 10. Table 11.2 shows a summary of the main evaluation results obtained. The controller met the expectations of a medium performance controller with latitude for tremendous improvement. The main limitation identified was the trajectory update rate of the motion controller chips employed in the design of the IMC.

11.3 Research Achievements

The main contributions of the work presented in this thesis are as follows:

1. An Ethernet-based real-time communication architecture with implicit clock synchronization. The technology also enables the controller sub-component design to incorporate embedded web-servers for remote monitoring and system configuration.
2. The development and demonstration of a protocol for automatic configuration (i.e., PnP) of controllers and other embedded shop floor devices.
3. The design and implementation of a medium-performance flexible controller incorporating the above on a homogenous Java software and hardware processor environment.

Table 11.2: Performance Evaluation Results

Evaluation Criteria	Value
Maximum servo loop cycle time	341 µs
Communication latency	4 µs
Block processing time	3-axis linear interpolation: 1.5 ms 2-axis circular interpolation: 2.0 ms
Synchronicity	Synchronization through network: maximum jitter ≈ 1 ms Synchronization by interrupt line: maximum jitter ≈ 30 µs
Positioning accuracy	Max. radial error on 25 mm circle: 0.03 mm (BLU ≈ 0.003mm)
Architectural flexibility	Easy to add/remove axis Quick adaptation to different mechanical platforms

11.4 Discussions and Future Research Direction

The Java-processor (aJile) used in the research is still in its infancy stage. We would like to see further development of this system and also other Java-based systems. Currently, the stripped-down version of Java (J2ME) that aJile uses makes programming a bit cumbersome. Moreover, a complete implementation of the real time specifications for Java (RTSJ) will enable designers to have a richer suite of scheduling methods at the kernel level. Presently, only fixed priority scheduling with preemption is supported by aJile.

An implementation of the IEEE 1588 clock synchronization protocol on aJile-based systems such as JStick is also greatly desired. This technology will require a more sophisticated Ethernet system that is capable of autonomous communication: In this case, control signals can be serviced directly by low-level communication protocols. This capability will free up computing resources for high-level tasks.

There are research potentials in improving and extending the Zeroconf protocol to networked factory-floor equipment and even network-centric field devices. This protocol could mitigate the technical and time cost of reconfiguring machines.

Demonstrative work performed to evaluate the architecture was not very thorough due to time constraints. A full evaluation is necessary to characterize and tune various aspects of the architecture such as the ones outlined below:

- For fault tolerance, an internal clock synchronization algorithm has to be incorporated into the current external synchronization method.
- The software architecture has to be optimized for performance and openness.
- Implementation of generalized kinematics for a wide-range of robotic devices should be investigated.
- At the moment, high-level control algorithms such as adaptive control or cross-coupling control have not been implemented. Fault tolerant independent control is definitely an interesting research area to be exploited.
- The motion control chips used on the controller board are fixed PID types. We will like to see an upgrade with modern ASIC chips capable of handling more sophisticated algorithms. However, these chips are quite expensive and susceptible to obsolescence. Ultimately, we would like to develop a Field Programmable Gate Array (FPGA) system that can be configured for different motion control requirements.

REFERENCES

aJ-100 Reference Manual, aJile Systems, Inc., 2001.

Altintas Y, *Manufacturing Automation Metal Cutting Mechanics, Machine Tool Vibrations, and CNC Design*, Cambridge University Press, 2000.

Altintas Y., N. Newell and M. Ito, *Modular CNC Design for Intelligent Machining. 1. Design of a Hierarchical Motion Control Module for CNC Machine Tools*, Journal of Manufacturing Science and Engineering, 1996; 118 (4), pp. 506-513.

Analog Devices, *Microprocessor-Compatible 12-Bit D/A Converter*, REV A, 2003.

Anceaume E. and I. Puauta, *Taxonomy of Clock Synchronization Algorithms*, Research report IRISA, NoPI1103, July 1997.

Andrews G., *Paradigms for Process Interaction in Distributed Programs*, ACM Computing Surveys, 23(1), Mar. 1991, pp. 49-90.

Atta-Konadu R., S. Y. T. Lang, P. Orban and C. Zhang, *Design and Implementation of a Modular Distributed Control Architecture for Robot Control*, 14th International Conference on Flexible Automation and Intelligent Manufacturing FAIM2004, July 12-14, Toronto, Canada, 2004, pp 441-48.

Atta-Konadu R., S. Y. T. Lang, C. Zhang and P. Orban, *Design of a Robot Control Architecture*, Mechatronics and Automation, 2005 IEEE International Conference, Vol. 3, 2005, pp. 1363- 68.

Atta-Konadu R., S. Y. T. Lang, P. Orban and C. Zhang, *Performance Evaluation of a Distributed Reconfigurable Controller Architecture for Robotic Applications*, ASME International Mechanical Engineering Congress and Exposition, Orlando Florida, 2005, pp. 1627-33.

Bellini P., M. Buonopane and P. Nesi, *Assessment of a Flexible Architecture for Distributed Control*", Programming and Computer Software, Vol. 29, No. 3, 2003, pp. 147-160.

Benítez-Pérez H and F. García-Nocetti, *Reconfigurable Distributed Control*, Springer-Verlad London Ltd, 2005.

Berners-Lee T., R. Fielding, H. Frystyk, *Hypertext Transfer Protocol HTTP 1.0*, RFC 1945, May 1996.

Birla S., D. Faulker, J. Michaloski, S. Sorenson, G. Weinert and J. Yen, *Reconfigurable Machine Controllers using the OMAC API*, Proceedings of the CIRP 1st International Conference on Reconfigurable Manufacturing , Ann Arbor, MI - May 01, 2001

CadSoft Computer, Inc., Eagle Version 4.16r1, 2003.

C&D Technologies, *DC-DC Converters Application Notes*, C&D Technologies (NCL) Ltd, 2000.

C&D Technologies, Product Data Sheet, HL02RrevD, 10/97.

Chen D. J., *Architecture for Systematic Development of Mechatronics Software Systems,* Licentiate Thesis, Department of Machine Design, Royal Institute of Technology, KTH, Sweden, 2001.

Chesney C., *Which Bus Architecture Is Best for You?* EE:-Evaluation Engineering, V 37 N 9 1998, pp. 12, 14, 17, 19.

Coste-Maniere E. and R. Simmons, *Architecture, the Backbone of Robotic Systems*, Proceedings of the 2000 IEEE International Conference on Robotics & Automation, San Francisco, CA., vol.1, 2000, pp. 67-72.

Chadha B. and J. Welsh, *Architecture Concepts for Simulation-based Acquisition of Complex Systems*, Summer Computer Simulation Conference, 2000.

De Luca A. and G. Oriolo, *Trajectory Planning and Control for Planar Robots with Passive Last Joint*, The International Journal of Robotics Research Vol. 21, No. 5-6, 2002, pp. 575-590.

Dhayagude N and Z. Gao, *A novel Approach to Reconfigurable Control System Design*, Journal of Guidance, Control and Dynamics, Vol. 19, N04, 1996, pp 963-967.

Divelbiss A. W. and J. T. Wen, *A Path Space Approach to Nonholonomic Motion Planning in the Presence of Obstacles***,** IEEE Transactions on Robotics and Automation, Vol. 13, No. 3, June 1997, pp. 443-51.

Dugenske A, A. Fraser, T., Nguyen and R. Voitus, *The National Electronics Manufacturing Initiative (NEMI) plug and play factory project*", International Journal of Computer Integrated Manufacturing, 2000, Vol. 13, No. 3, pp. 225-44.

Feng-Li L., J. R. Moyne and D. M. Tilbury, *Implementation of Networked Machine Tools in Reconfigurable Manufacturing Systems***,** Proceedings of the 2000 Japan-USA Symposium on Flexible Automation, Ann Arbor, MI, July 2000.

Fielding R. T., *Software Architectural Styles for Network-Based Systems*, PhD Thesis, University of California, Irvine, 1999.

Finkemeyer B., M. Borchard and F. M. Wahl, *A Robot Control Architecture Based on an Object Server*, Proceedings of the IASTED International Conference, Robotics and Manufacturing, IASTED/ACTA Press, Calgary, Canada, 2001.

Fuggetta A., G. P. Picco and G. Vigna. *Understanding code mobility*, IEEE Transactions on Software Engineering, 24(5), May 1998, pp. 342-361.

Gaderer G., Höller R., Sauter T. and Muhr H., *Extending IEEE 1588 to Fault Tolerant Clock Synchroniztion*, IEEE, 2004.

Garlan D. and M. Shaw. *An Introduction to Software Architecture*, Ambriola & Tortola (eds.), Advances in Software Engineering & Knowledge Engineering, vol. II, World Scientific Pub Co., Singapore, 1993, pp. 1-39.

Goktas F., *Distributed Control of Systems over Communication Networks*", Ph.D Thesis, University of Pennsylvania, 2000.

Guttman E., *Autoconfiguration for IP Networking: Enabling Local Communication*, IEEE Internet Computing, 2001, pp. 81-88.

Hong K. S., K. H. Choi, J. G Kim and S. Lee, *A PC-Based Open Robot Control System: PC-ORC*, Robotics and Computer Integrated Manufacturing 17 (2001) 355–365

HTML 4.0 Specification, W3C Recommendations, REC-html40 – 19980424, 1998.

Hunt S., *LM628 Programming Guide*, National Semiconductor Application Note 693, 1999.

JBuilder 2005, Borland Software Corporation, 2005.

James J. and McClain R., *Tools and Techniques for Evaluating Control Architecture*, Proceedings of the 1999 IEEE International Symposium on Computer Aided Control System Design, Kohala Coast-Island of Hawai'i, Hawai'i, USA, August 22-27, 1999.

Kapoor C., *A Reusable Operational Software Architecture for Advanced Robotics*, PhD Thesis, The University of Texas at Austin, 1996.

Kim K. H., Im C. and Prasad A., *Realization of a Distributed OS Component for Internal Clock Synchronization in a LAN Environment*, Proceedings of the 5th 1EEE International Symposium on Object-Oriented Real-Time Distributed Computing, 2002.

Kopetz H., Ademaj A. and Hanzlik A., *Integration of Internal and External Clock Synchronization by the Combination of Clock-State and Clock-Rate Correction in Fault-Tolerant Distributed Systems*, Proceedings of the 25th 1EEE International Real-Time Systems Symposium, 2004.

Kopetz H., *Real Time Systems: Design Principles for Distributed Embedded Applications*, Boston, MA, USA: Kluwer Academic Publishers, 1997.

Kramer T. and M. K. Senehi: *Feasibility Study: Reference Architecture for Machine Control Systems Integration*, NISTIR 5297, National Institute of Standards and Technology, Gaithersburg, MD, 1993.

Lee C. J. and C. Mavroidis, *PC-Based Control of Robotic and Mechatronic Systems Under MS-Windows NT Workstation*" IEE/ASME Transactions on Mechatronics, Vol. 6, No. 3, 2001., pp. 311-321.

Lee E. A., *Embedded Software*, Advances in Computers (M. Zelkowitz, editor), Vol. 56, Academic Press, London, 2002.

Lian F. L., J. R. Moyne and D. M. Tilbury, *Implementation of Networked Machine Tools in Reconfigurable Manufacturing Systems*, 2000 Japan-USA Symposium on Flexible Automation July 23-26, 2000, Ann Arbor, Michigan, USA

Lin C. and C. S. G. Lee, *Fault-Tolerant Reconfigurable Architecture for Robot Kinematics and Dynamics Computations*, IEEE Transactions on Systems, Man, and Cybernetics, Vol. 21, No5, 1991, pp. 983-99.

LM628/LM629, *Precision Motion Controller*, National Semiconductor Corporation DS009219, 2003.

Lönn H, *Synchronization and Communication Results in Safety-Critical Real-Time Systems*, PhD thesis, Department of Computer Engineering, Chalmers University of Technology Göteborg, Sweden 1999.

Lönn H., Synchronization and Communication Results in Safety-Critical Real-Time Systems, Chalmers University of Technology, 1999.

Lundelius J. and N. Lynch, *An Upper and Lower Bound for Clock for Clock Synchronization*, Information and Control, 62(2/3), 1984, pp. 190-204.

Medvidovic N. and R. N. Taylor, *A Classification and Comparison Framework for Software Architecture Description Languages*, IEEE Transactions on Software Engineering, Vol. 26, No. 1, Jan. 2000.

Mehrabi M. G., A. G. Ulsoy and Y. Koren, *Reconfigurable Manufacturing Systems and Their Enabling Technologies*, International J. of Manufacturing Technology and Management, 2000.

Mohl D. S., *IEEE 1588 - Precise Time Synchronization as the Basis for Real Time Applications in Automation*, Available at www.industrialnetworking.com, 2003.

Nesnas I., A Wright, M. Bajracharya, R Simmons, T. Estlin and W. S. Kim, *CLARAty: An Architecture for Reusable Robotic Software*, 2003, pp. 253-264.

Nilsson J., *Real-time Control System with Delays*, PhD thesis, Dept. of Automatic Control, Lund Institute of Technology PhD thesis, 1998.

Oaks S. and H. Wong, *Java Threads*, O'Reilly Media, Inc., 2004.

Oldknow K. D. and I. Yellowley, *Design, implementation and validation of a system for the dynamic reconfiguration of open architecture machine tool controls*, International Journal of Machine Tools & Manufacture 41, 2001, pp. 795–808.

Orban P., R. Atta-Konadu, S. Lang, M. Verner and C. Zhang, *Java-based Distributed Control System for Reconfigurable Manufacturing*, 3rd CIRP International Conference on Reconfigurable Manufacturing, Ann Arbor, Michigan, 2005.

Pardo-Castellote G., *Experiments in the Integration and Control of an Intelligent Manufacturing Workcell*, PhD Thesis, Dept of Electrical Engineering, Stanford University, June 1995.

Passmore D., *Zero Configuration Networks*, Burton Group Research, Business Communications Review, 2002, available at http://www.burtongroup.com/promo/columns/column.asp?articleid=122&employeeid=56

Pedreiras P. and L. Almeida, *Combining Event-Triggered and Time-Triggered Traffic In FTT-CAN: Analysis of the Asynchronous Messaging System*", Proceedings of the 2000 IEEE International Workshop on Factory Communication Systems (WFCS'00). Porto, Portugal: IEEE Industrial Electronics Society, 2000, pp. 67--75.

Reference.com, *Zeroconf*, Available at http://www.reference.com/browse/wiki/Zeroconf.

Rossignac J. R. and J. J. Kim, *Computing and Visualizing Pose-Interpolating 3D Motions*, Computer Aided Design, 33(4):279–291, 2001.

Schickhuber G. and O. McCarthy, *Distributed Fieldbus and Control Network Systems*, Computing & Control Engineering, Vol. 8, no. 1, pp. 21--32, Feb. 1997.

Schöberl M., *JOP: A Java Optimized Processor for Embedded Real-Time Systems*, PhD Thesis, Vienna University of Technology, Austria, 2005.

Shaw M., *Beyond Objects: A Software Design Paradigm Based on Process Control*, ACM Software Eng., Engineering Notes, Vol. 20, No 1, January 1995.

Shaw M., *Comparing Architectural Design Styles*, IEEE Software, Volume: 12 Issue: 6, Nov. 1995, Page(s): 27 –41.

Shaw M. and D. Garlan, *Software Architecture. Perspectives on An Emerging Discipline*, Prentice- Hall Inc., 1996.

Shaw M., R. DeLine, D. V. Klein, T. L. Ross, D. M. Young, and G. Zelesnick, *Abstractions for Software Architecture and Tools to Support Them*. IEEE Transactions on Software Engineering, 21(4), Apr. 1995, pp. 314-335.

Silverman S., *Tynamo-aJile Version 1.0-RC1*, 2004.

Somló J, B. Lantos and P. T. Cat, *Advanced Robot Control*, in Advances in Electronics, Vol. 14, Budapest, Akademiai Kiado, 1997.

Sorensen A. S., *Modular Control of Industrial Mechanics*, PhD Dissertation, University of Denmark, 2003.

Steinberg D. and Y Birk, *An Empirical Analysis of the IEE-1394 Serial Bus Protocol*", IEEE-Micro. Vol. 20 N 1 2000, pp 58-65.

Strandberg M., *Robot Path Planning: An Object-Oriented Approach*, PhD Thesis, Automatic Control Department of Signals, Sensors and Systems Royal Institute of Technology (KTH) Stockholm, Sweden, 2004.

Sun Microsystems, Java(TM) Servlet API Specification 2.2, 1999.

Systronix, *JStik High Speed I/O Timing Diagram*, Systronix, Inc., Rev 1.0, 2001.

Systronix*, JStik™ Power & the JSimm Bus*, Systronix, Inc. 2003.

Systronix, Technical Reference for Systronix JStik Realtime Native Java Network Module, Systronix Inc., 2003.

Tanenbaum A. S. and R. van Renesse, *Distributed Operating Systems*, ACM Computing Surveys, 17(4), Dec. 1985, pp. 419-470.

Tanenbaum A. S. and M. van Steen, *Distributed Systems: Principles and Paradigms*", Prentice Hall, 2002.

Thomesse J.-P, *A review of the Fieldbuses*, Annual reviews in Control 22, pp35-45, 1998.

Thomson, S. and T. Narten, *IPv6 Stateless Address Autoconfiguration*, RFC 2462, December 1998.

Timing Designer, Pro v5.3 Datasheet, Forte Design Systems, 2003.

Topley K., *J2ME in a Nutshell*, O'Reilly and Associates, Inc, 2002.

Vyatkin V. V., J. H. Christensen and J. L. Martinez Lastra, *OOONEIDA: An Open, Object-Oriented Knowledge Economy for Intelligent Industrial Automation*, IEEE Transactions on Industrial Informatics, Vol. 1, No. 1, pp. 4-17, 2005.

Veríssimo P. and L. Rodrigues, *Distributed Systems for System Architects*. Boston, MA, USA: Kluwer Academic Publishers, 2001.

Wang S. and Shin K. G., *Reconfigurable Software for Open Architecture Controllers*, In Proceedings of the 2001 IEEE International Conference on Robotics & Automation, 2001, pp. 4090-94.

Wang Z., Y. Song, J. Chen and Y. Sun, *Real Time Characteristics of Ethernet and Its Improvement*", Proceedings of the 4th World Congress on Intelligent Control and Automation, China, 2002, pp. 1311- 1318.

Weck M., Handbook of Machine Tools Vol. 3, Automation and Controls, Wiley Heyden Publication, 1984.

Wills L., S. Kannan, S. Sander, M. Guler, B. Heck, J.V.R. Prasad, D. Schrage and G. Vachtsevanos, *An Open Platform for Reconfigurable Control*, IEEE Control Systems Magazine, Vol. 21, No. 3, 2001, pp. 49-64.

Wittenmark B., J. Nilsson and M. Törngren, *Timing problems in real-time control systems*, In Proceedings of the 1995 American Control Conference, Seattle, Washington Control Conference, Seattle, Washington.

Yook J., D. Tilbury, K. Chervela, N. Soparkar, *Decentralized, Modular Real-Time Control for Machining Applications*, In: 1998 American Control Conference, Philadelphia, PA, June 24-26, 1998, Proceedings. Vol. 2 (A99-14618 02-63), Institute of Electrical and Electronics Engineers, P 844-849, 1998.

Zimmerman H., *OSI Reference Model - The ISO Model of Architecture for Open Systems Interconnection*, IEEE Transactions on Communications, 28, Apr. 1980, pp. 425-432.

APPENDIX A. THE IMC HARDWARE

A1 IMC Motion Control Board Schematics

This section illustrates the IMC hardware schematics.

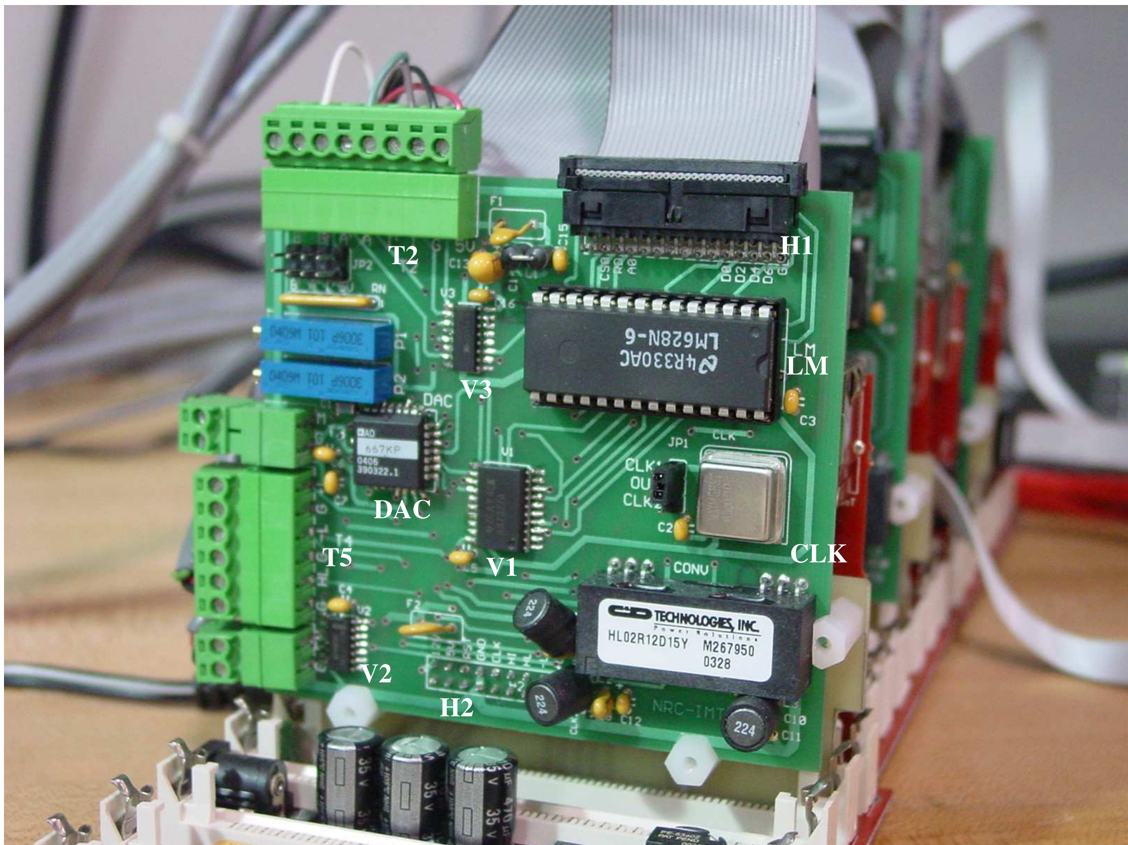
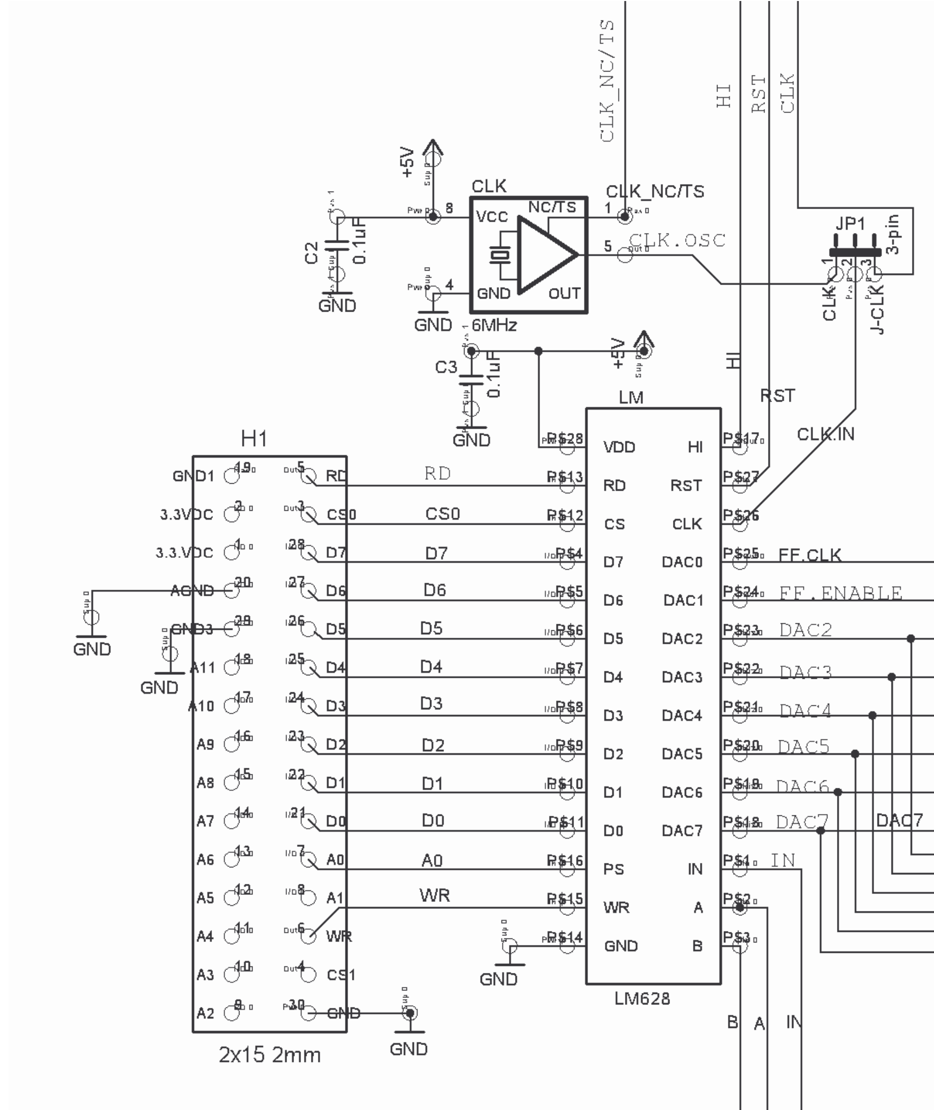


Figure A1: Motion Control Board (3.5mm x 3.2mm)



Date: Oct 2004	LM628 Interface
Title: IMC	Sheet: 1/5

Figure A2: LM628 Pin-out to Header (H1), and Clock

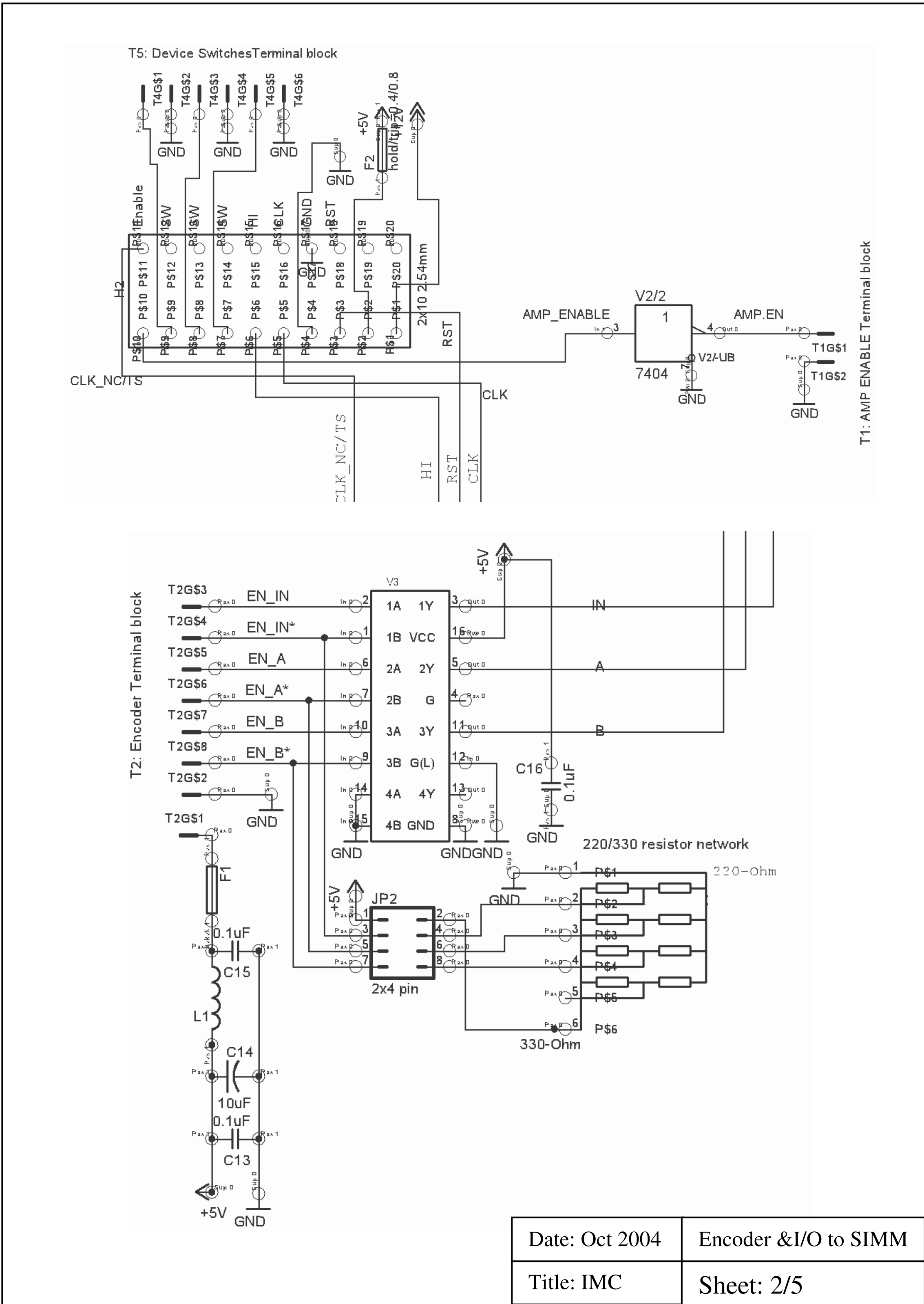


Figure A3: Encoder Interface and I/O to JStick SIMM Interface

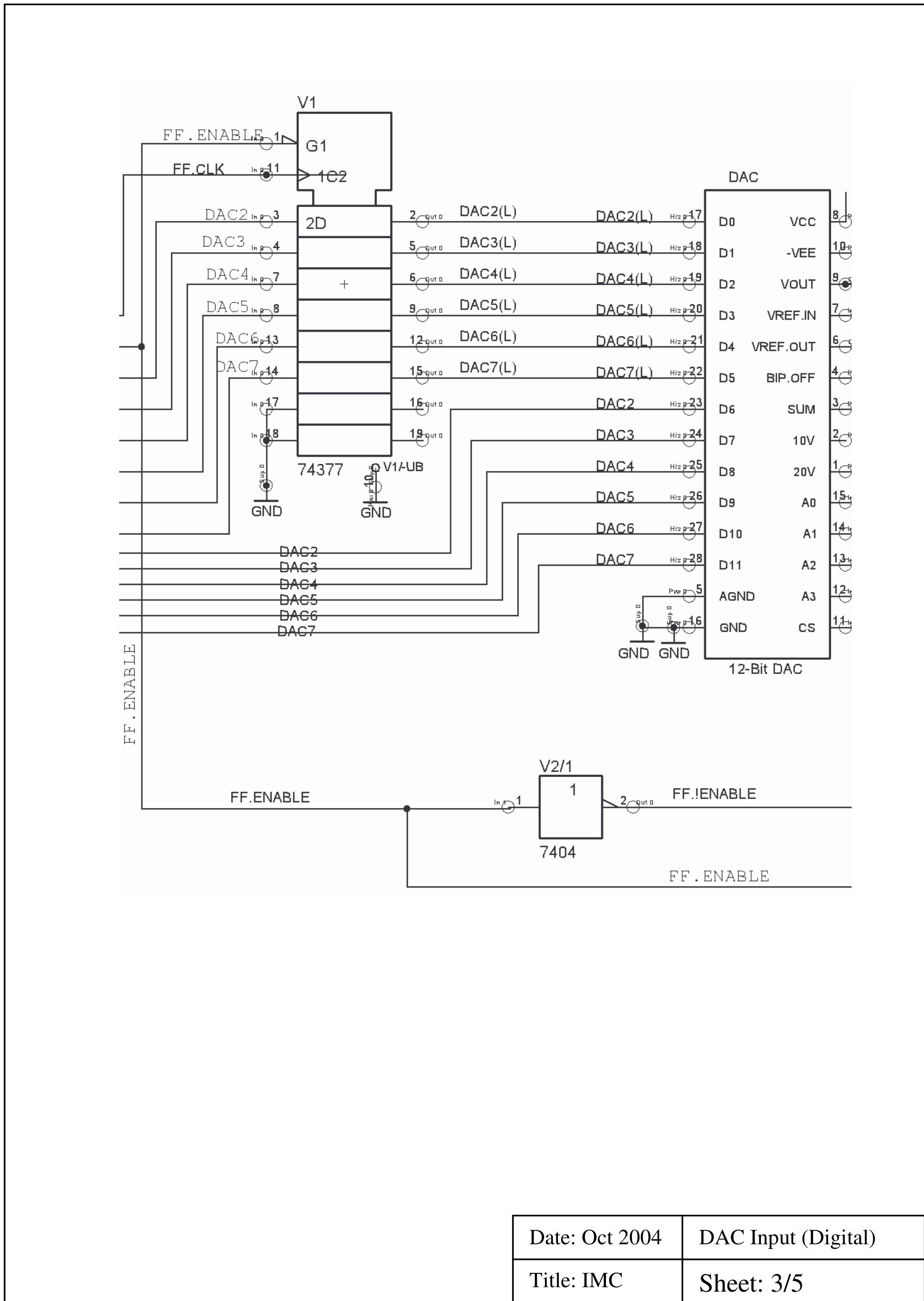
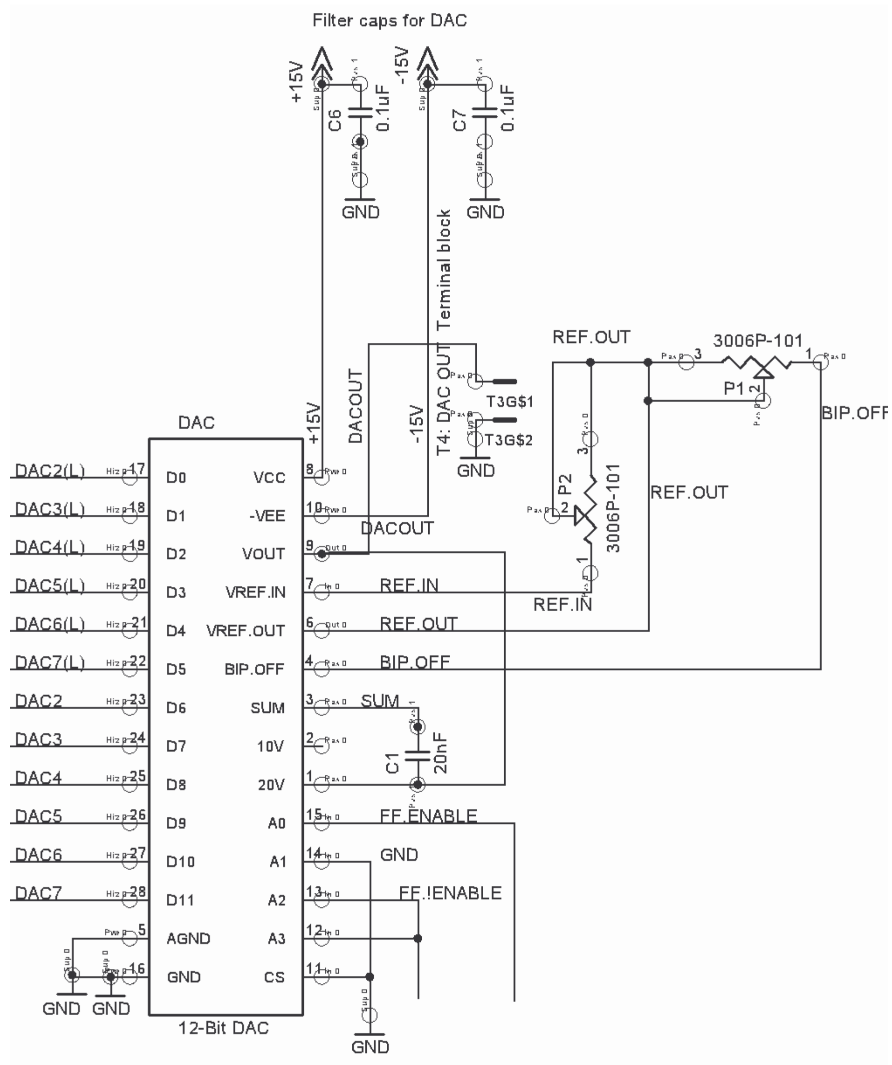


Figure A4: AD667 DAC Interface with Logic Devices



Date: Oct 2004	DAC Output (Analog)
Title: IMC	Sheet: 4/5

Figure A5: AD667 DAC Output Circuit

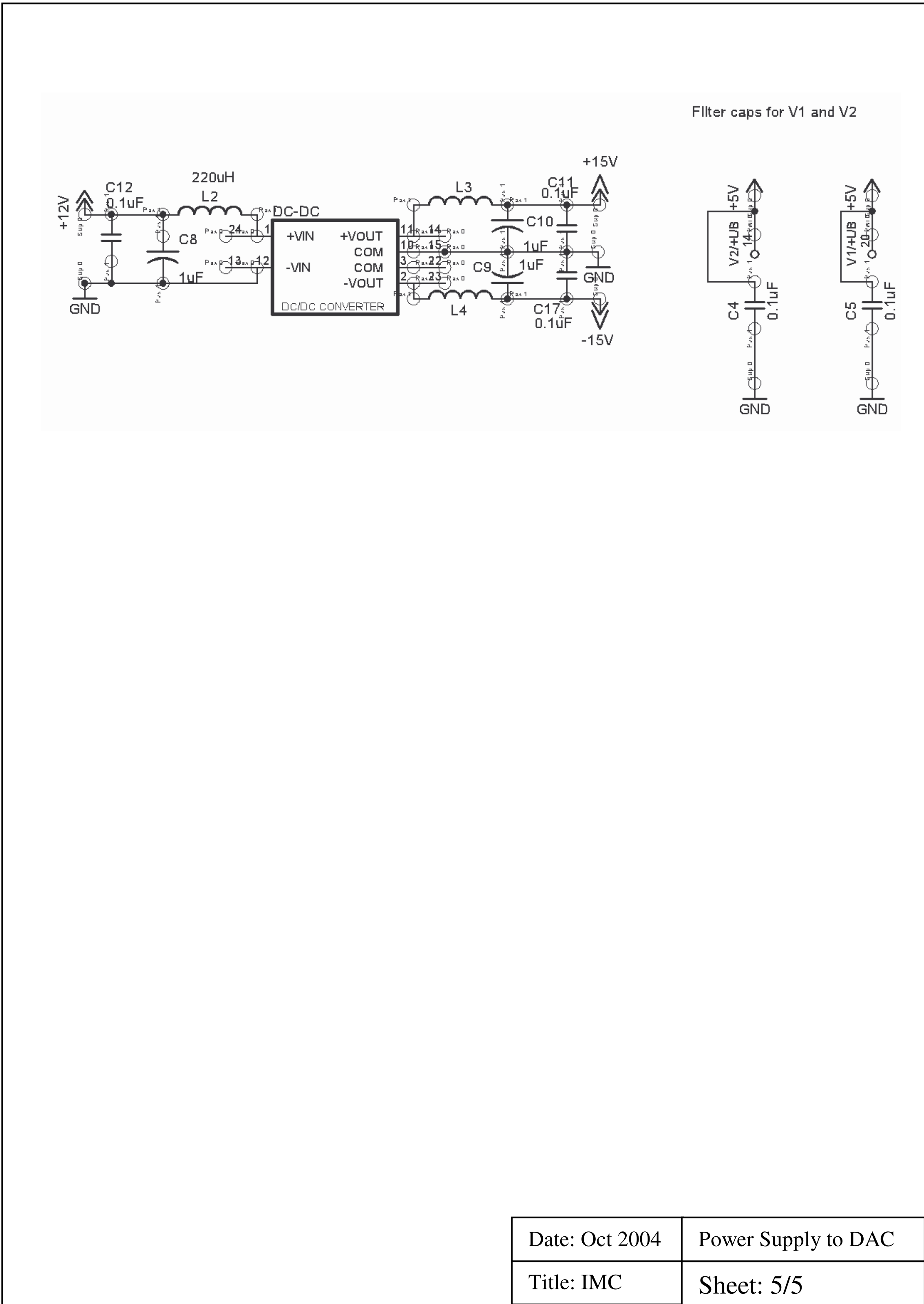


Figure A6: Filters and Power Supply

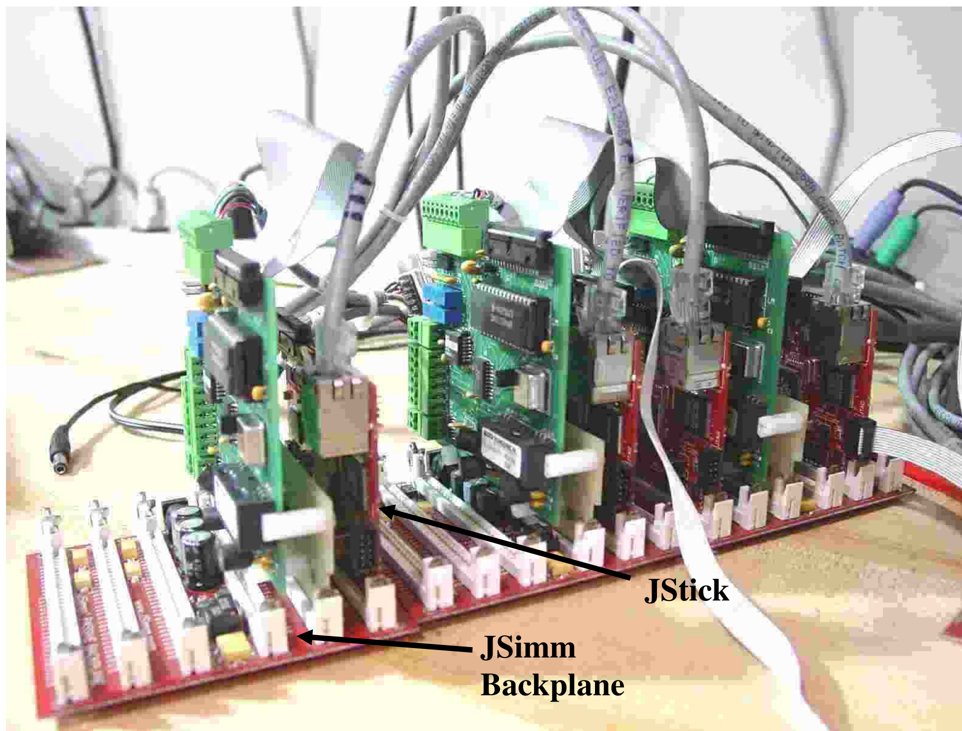


Figure A7: IMC Ensemble

A2 IMC Controller BOM

This section shows the Bill of Materials (BOM).

Table A1: Bill of Materials (2004)

Part	Value	Device	Package	Price (CND$)
C1	20pF	Capacitor	C1206	0.30
C2	0.1uF	Capacitor	C – 2.5	0.20
C3	0.1uF	Capacitor	C – 2.5	0.20
C4	0.1uF	Capacitor	C – 2.5	0.20
C5	0.1uF	Capacitor	C – 2.5	0.20
C6	0.1uF	Capacitor	C – 2.5	0.20
C7	0.1uF	Capacitor	C – 2.5	0.20
C8	1uF	Capacitor	C – 2.5	0.55
C9	1uF	Capacitor	C – 2.5	0.55
C10	1uF	Capacitor	C – 2.5	0.55
C11	0.1uF	Capacitor	C – 2.5	0.20
C12	0.1uF	Capacitor	C – 2.5	0.20
C12	0.1uF	Capacitor	C – 2.5	0.20

Part	Value	Device	Package	Price (CND$)
C14	10uF	Capacitor	C – 2.5	1.26
C15	0.1uF	Capacitor	C – 2.5	0.20
C16	0.1uF	Capacitor	C – 2.5	0.20
C17	0.1uF	Capacitor	C – 2.5	0.20
CLK	6 MHz	Oscillator	DIL08S	3.96
DAC	AD667	12-Bit DAC converter	PLCC28	11.87
DC-DC	1 Watt; 12VDC +/ -15VDC	DC-DC Converter	DIP	21.40
F1	0.1A hold	Resettable fuse		0.68
F2	0.5A hold	Resettable fuse		0.76
H1	2 x 15; 2mm	Header	DIL 2 mm	5.24
H2	2 x 15; 2.54mm	Header	DIL	5.00
JP1	3-pin	Jumper		0.60
JP1	2 x 4 pin	Jumper	Dual straight	0.79
L1	143 Ohm	Ferrite Bead	Axial 3.51	2.62
L2	220 uH	Inductor	Radial	0.50
L3	220 Ohm	Inductor	Radial	0.50
L3	220 Ohm	Inductor	Radial	0.50
LM	LM628	Precision Motion Controller	DIL28	35.20
PI	100 Ohm	Potentiometer	SIL	2.13
P2	100 Ohm	Potentiometer	SIL	2.13
RN	220/330 Ohm	Resistor Network	SIL6	0.74
T1	2 position	Terminal block	3.81 mm	2.90
T2	8 position	Terminal block	3.81 mm	9.40
T3	2 position	Terminal block	3.81 mm	2.90
T4	6 position	Terminal block	3.81 mm	7.10
V1	74377	D-Type FF	SO20W	0.35
V2	7404	Hex Inverter	SO14	0.22
V3	26LS32	Receiver	S016	0.36
Total				**163.46**
PCB (with silk screen and masking) Onetime Setup Fee				121.00
Fee per PCB				$7.49
Price for One Motion Control Board				**291.94**
Price for 6 Motion Control Boards				**1146.70**

Table A2: IMC Microcontroller, Cables & Router (2003)

Product	Approx. Unit Cost ($US)
JStick microcontroller	330.00
JStick backplane	97.90
Ribbon Cable and connectors	6.60
Ethernet Cable (CAT5)	7.70
Router (8 port)	99.00

A2 Motion Control Chips

Traditionally, two chips have dominated the embedded motion control and custom built applications arena. They are the Agilent[4] HCTL-1100, and the National Semiconductor LM628 listed in Table A.1. The HCTL-1100 costs about US$ 49. The two chips are very similar in their features and operations. The HCTL-1100 is 5V CMOS, which makes it incompatible with JStick's 5V TTL levels. Currently, many other motion control chips are being introduced on the market to replace these aging ones. Table A3[5] gives a comparison overview of some popular ones.

[4] The semiconductor branch of Agilent (formerly HP) is currently owned by Avago Technologies

[5] Randy Frank, *Design News*, Sept. 13, 2004.

Table A3: Comparison of Motion Control Chips

Company	Description	Motor Types[1]	Features	Package	Price (US)
Freescale Semiconductor 56F8357	16-bit DSP/MCU Hybrid (60MIPS @ 60MHz), 76 I/O	ACIM, BDC, BLDC, SRM, VRM, and Stepper Motors. 12 PWM outputs (6 individually programmable)	- 12 PWM outputs (6 individually programmable) - 256K Flash - Four 4-channel, 12-bit ADCs	160-pin LQFP	$17.92 in quantities of 1,000 Evaluation Module ($299)
International Rectifier IRMCK201	AC Servo Motor Control IC. Uses external MCU or host. 54 I/O	ACIM or ACPM Servo Drive System	- Space Vector PWM with 12-Bit resolution - 128 x 8 EEPROM - 4-channel, 12-bit ADC	100-pin QFP	$8.75 in qnties of 10,000 Development System ($1,999)
Performance Motion Devices Inc. MC58420	Motion Processor with over 130 commands. Uses external MCU or host. 256 I/O	DC servo, BLDC Microstepping, and Pulse and Direction Motors	- 10-bit 20 kHz PWM or 160bit DAC motor control output	144-pin TQFP	$14.75 per axis in qnties of 10,000 Developer's Kit ($995)
Microchip Technology PIC18F4431	8-bit MCU (10 MIPS @ 40 MHz) 36 I/O	ACIM and BLDC	- 8 Channels 14-bit Power Control PWM - 16 KB Flash 256B EEPROM - 9 channels of 10-bit ADC	44-pin TQFP	Price: $5.20 in qnties of 10,000 Development Board ($299)
ST Micro-electronics ST7MC1	8-bit MCU (8MHz) 60 I/O	ACIM and BLDC	- 6 high-sink PWM output channels for sinewave or trapezoidal control - 24KB Flash - 10-bit ADC with 16 input pins	64-pin TQFP	Price: $2.86 in qnties of 10,000 Starter Kit ($695)
Texas Instruments TMS320R2811	32-bit DSC (150MIPS @ 150MHz) 56 I/O	Servo Control	- 2 Event Managers each with 16-bit Compare/PWM - 20K x 16-bit SARAM - 12-bit, 16-channel ADC	128-pin PBK	$9.11 in qnties of 1,000.Kit ($495)

[1]Abbreviations: ACIM (AC Induction Motors); BDC (Brush DC motors); BLDC (Brushless DC motors); SRM (Switched Reluctance Motors); PM (Permanent Magnet); SARAM (Single Access RAM)

APPENDIX B: IMC SOFTWARE INTERFACE

The IMC domain is responsible for axis-specific activities such as joint control. In addition, the domain contains many components for intelligent interaction with the rest of the system. Table B1 summarizes the software packages and their corresponding classes. The class interfaces are described in this appendix.

Table B1: IMC Software

Package	Description	Classes
com.IMC.database	Network, configuration data and data access methods	*Data* *PushPullData* *FIFO* *JmDNS_Coordinator_Data* *FileServer_ConfigFiles*
com.IMC.coordination	Protocols for negotiating with system coordinator	*StateCoordinator* *MultiCasted_States* *Device* *StateBuffer* *Interpolation_Server* *Monitor* *EncoderReader*
com.IMC.drivers	Low level drivers for the communicating with the LM628, interrupt services, and digital I/O drivers	*HSIO_Driver* *Board_Clock* *JStickTimer_tc2* *LM628_Interrupt* *LimitSwitch* *Reference_Switch_Driver* *GPIOPinA*
com.IMC.network	Ethernet protocols	*DatagramServer* *MulticastServer* *TCPServer*
com.IMC.servlets	Web interface for viewing/editing configuration and PnP mechanisms	*JmDNS_Coordinator* *ConfigureDevice* *EditJmDNS* *PositionDump* *ControllerInfo* *ShutdownServlet*

B1 com.IMC.database

Class Data

public abstract class **Data**

Contains default and user defined network and motion device settings

Field Detail

public static final int **DEFAULT_MULTICAST_PORT_NUMBER**

---

public static final java.lang.String **DEFAULT_MULTICAST_IP_ADDRESS**

---

public static final java.lang.String **PRIORITY_MULTICAST_IP_ADDRESS**

---

public static final int **DEFAULT_DATAGRAM_PORT_NUMBER**

---

public static final int **PRIORITY_MULTICAST_PORT_NUMBER**

---

public static final int **PRIORITY_DATAGRAM_PORT_NUMBER**

---

public static final int **DEFAULT_MONITOR_SOCKET_PORT**

---

public static final java.lang.String **DEFAULT_SUPERVISOR_IP**

---

public static final int **DEFAULT_ENCODER_LINES**

---

public static int **USER_ENCODER_LINES**

---

public static final int **DEFAULT_LIMIT_SWITCHES**

---

public static int **USER_LIMIT_SWITCHES**

---

public static final java.lang.String[] **DEFAULT_DEVICE_NAMES**

---

public static java.lang.String **DEVICE_NAME**

---

public static final double **conversionFactor**

doubles are converted to integers by this parameter by the PC host before transmission over the network to the IMC.

---

public static double **mmToRevFactor**

conversion from travel distance in mm to rev (pitch-screw factor)

---

public static final java.lang.String **DEFAULTmmToRevFactor**

default conversion from travel distance in mm to rev = 1

---

public static int **PID_PROPORTIONAL**

---

public static int **PID_INTEGRAL**

---

public static int **PID_DERIVATIVE**

---

public static int **PID_INTEGRAL_LIMIT**

---

public static int **PID_DERIVATIVE_SAMPLING_SIZE**

---

public static final int **DEF_PID_PROPORTIONAL**

---

public static final int **DEF_PID_INTEGRAL**

---

public static final int **DEF_PID_DERIVATIVE**

---

public static final int **DEF_PID_INTEGRAL_LIMIT**

---

public static final int **DEF_PID_DERIVATIVE_SAMPLING_SIZE**

---

public static final int **DEF_EXCESSIVE_POS_ERROR**

---

public static int **encoderRepInt**

DEFAULT Encoder reporting interval = 100

---

public static double **speedOverRide**

---

public static int **EXCESSIVE_POS_ERROR**

---

public static final java.lang.String **CLOCKSOURCE**

DEFAULT clock source = BOARD_CLOCK

---

public static int **clock**

---

public static boolean **initializeController**

Controller initialized flag

---

public static int **initializeControllerFlag**

Controller initialized flag saved to file

---

public static boolean **isInterpolationDone**

Interpolation complete flag - encoder read dump follows if true

---

public static boolean **togglePIN3**

flag for toggling pin 3 so it can be monitored on a logic analyzer

---

public static double **home_pos**

Default home position = 0

---

public static int **logGranularity**

Frequency of logging position data

---

public static boolean **logPositions**

Flag for position logging; default is false i.e. no logging

---

public static final java.lang.String[] **configinfo**

Configuration information containing services available on platform

---

public static final java.lang.String[] **config**

default configuration information in matrix form

com.IMC.database
Class PushPullData

public class **PushPullData**

This Class manages data produced and consumed such as position set-points

Field Detail

private static int **supervisorPort**

---

private static boolean **isAbsolutePos**

---

private static boolean **isAbsoluteVel**

---

public static com.IMC.database.FIFO **posFIFOBuffer**

---

public static com.IMC.database.FIFO **velFIFOBuffer**

Method Detail

public synchronized double **get_pos**()

Returns:

position

---

public synchronized void **put_pos**(double position)

Used for single point move

Parameters:

position - double - position

---

public synchronized double **get_vel**()

if velocity is 0 thread calling this method will be put in a wait state until notified. This is used to synchronize start of motion.

Returns:

velocity

---

public synchronized void **put_vel**(double velocity)

For single position move

Parameters:

velocity - double - velocity

---

public synchronized void **put_accl**(double acceleration)

For single position move

Parameters:

acceleration - double

---

public synchronized void **put_isAbsolute**(boolean isAbsPos,
boolean isAbsVel)

For position and velocity

Parameters:

isAbsPos - true if position is absolute and false if relative
isAbsVel - true if velocity is absolute

---

public synchronized boolean **isAbsolutePos**()

Returns:
true for absolute position

---

public synchronized boolean **isAbsoluteVel**()
Returns:
true for abs velocity

---

public synchronized double **get_accl**()
Returns:
acceleration requested by supervisor

---

public synchronized int **get_Profile_type**()
Returns:
int indicating motion profile mode type requested by supervisor

---

public synchronized void **put_Profile_type**(int type)
Parameters:
type - profile mode

---

public synchronized void **put_speedOverRide**(int factor)
Parameters:
factor - speed over-ride percentage; 100 => 1, ie no overider; 120 => 1.2

---

public synchronized double **get_speedOverRide**()
Returns:
double speed Over-ride

---

public static void **createPosVelFIFO**(int size)
Creates one FIFO each for position & velocity data to be received from supervisor
Parameters:
size - size of each FIFO

---

public synchronized int **get_Filter**(int i)
Parameters:
i - index for filter
Returns:
specific filter (eg proportional) value

---

public synchronized void **put_Filter**(int data,
int i)
sets PID filter values with user-defined data array
Parameters:
data - int; specific filter value
i - position of parameter eg 1 for proportional, 2 for integral

---

public static synchronized java.lang.String **get_SupervisorIP**()
Returns:
String - Supervisor's IP Address

---

public synchronized void **put_SupervisorIP**(java.lang.String address)
Stores supervisors IP address
Parameters:
address - String; IP address

public static synchronized int **get_SupervisorPort**()
Returns:
int Supervisor's IP port number

---

public synchronized void **put_SupervisorPort**(int port)
Stores supervisors IP port number
Parameters:
port - int port number

---

com.IMC.database
Class FIFO

public class **FIFO**
Contains various methods for managing a FIFO Buffer

Constructor Detail

public **FIFO**(int i)
Parameters:
i - buffer size

Method Detail

public synchronized void **synchput**(double i)
puts a double i at the tail end of the FIFO. If FIFO is full wait until there is space, ie hold lock until notified

---

public boolean **isFull**()
Returns:
true if FIFO is full

---

public int **getFree**()
Returns:
amount of free space

---

public synchronized double **synchronized_Take**()
take double from head of FIFO. If it's empty hold lock until notified
Returns:
double

---

public boolean **isEmpty**()
Returns:
true if FIFO is empty

---

com.IMC.database
Class FileServer_ConfigFiles

public class **FileServer_ConfigFiles**
extends com.IMC.database.JmDNS_Coordinator_Data
CLDC storage connection. Utility class for storing configuration files. It contains methods to configure files

Field Detail

private static final java.lang.String **FILE_SYSTEM_MOUNT_POINT**
The name of the Flash File System mount point;

---

private static final java.lang.String **FILE_SYSTEM_ROOT**
top level of the flash file system.

---

private static java.util.Vector **fileContent**
contents of file

Constructor Detail

public **FileServer_ConfigFiles**()
implicit call to the com.IMC.servlets.JmDNS_Coordinator

Method Detail

public void **run**()
searches directories for configurations files and creates them if with default values if they don't exist
Throws:
IOException -

---

public java.lang.String **getConfig**(java.lang.String[] value)
gets default configuration parameters from com.IMC.database.Data and converts them to a String
Parameters:
value - String[] contains configuration parameter
Returns:
String i.e, configuration parameter

---

public static void **getParameters**(java.lang.String filename)
retrieves stored controller configuration data from file and updates static variable in Constant.class
Parameters:
filename - String name of file

---

public void **createDir**(java.lang.String path,
java.lang.String filename)
Creates a directory.
Parameters:
path - is the subdirectory to create the file in
filename - is the name of the file to create
Throws:
IOException - if unable to access the file system

---

public static void **writeFile**(java.lang.String filename,
java.lang.String data)
Writes data to a file.
Parameters:
filename - is the location where the data is to be written
data - is the data to be stored in the file
Throws:
IOException - if unable to access the file system

---

public void **readFile**(java.lang.String fileName)

Reads and displays the contents of a file.

Parameters:

fileName - specifies the file to be read.

Throws:

IOException - if unable to access the file system

---

public void **directory**(java.lang.String path)

Displays directory structure starting at a root directory.

Parameters:

path - specifies the root directory to display

Throws:

IOException - if unable to access the file system

---

public void **deleteFile**(java.lang.String filename)

Deletes a single file from the file system.

Parameters:

filename - specifies the file to be removed

---

com.IMC.database
Class FileService

public class **FileService**

This class contains all methods needed by servlets etc for different file I/O operations

Method Detail

public static synchronized java.util.Vector **readFile**(java.lang.String filename)

Reads and displays the contents of a file into a Vector.

Parameters:

filename - String; specifies the file to be read.

Returns:

Vector holds contents of file

Throws:

IOException -

---

public static synchronized void **writeFile**(java.lang.String filename,
java.util.Vector data)

Writes data to file from a Vector. Elements are line separated

Parameters:

filename - String; name of file
data - Vector; Vector containing data to be written

Throws:

IOException -

---

public static void **deleteDir**(java.lang.String parentDir,
java.lang.String dir)

Deletes a single directory from the file system.

Parameters:

parentDir - directory that contains the directory to be deleted
dir - name of the directory to be deleted

Throws:

IOException - if unable to delete file

---

public static synchronized java.lang.String[] **split**(java.lang.String data, int n, char c)

method takes a string, splits it where there are commas and puts them in a String array of size n. This method is used to organize String data obtained from file for registering rendezvous services. Eg "_http._tcp.local., foo, 80, 0, 0, path=index.html" will be split into 6 strings; "_http._tcp.local." "foo" "80" "0" "0", "path=index.html"

Parameters:

data - String
n - number of strings
c - splitting is done wherever there is this character eg comma

Returns:

String[]; array containing data

Throws:

IndexOutOfBoundsException -

---

public static boolean **createRegister**(java.lang.String type,
java.lang.String name,
int port,
java.lang.String text,
java.util.Vector register)

Method used to create a JmDNS 'register service' string

Parameters:

type - String; eg "_http._tcp.local."
name - String; eg "X TABLE"
port - int; eg 6000
text - String; eg "Controller for X Table"
register - Vector; holds a concatenation of the above

Returns:

boolean

---

public static java.lang.String **removeAll**(java.lang.String string,
char charc)

method removes all characters (charc)such as white spaces from a string

Parameters:

string - String
charc - char; character to remove from string

Returns:

String; string without charc above

---

public static double **stringToDouble**(java.lang.String number)

this method converts a number in String primitive to a double value. Method not available in Java CLDC API.

Parameters:

number - String; number in String form

Returns:

double; number cast as double

---

com.IMC.database
Class JmDNS_Coordinator_Data

public class **JmDNS_Coordinator_Data**
extends com.IMC.database.Data

Class holds registers containing initial JmDNS data.

Field Detail

static java.lang.String **discoverServicesFile**

contains initial JmDNS services to be discovered by JmDNS_Coordinator.class

public static java.lang.String **registerServicesFile**

contains initial JmDNS services to be registered by JmDNS_Coordinator.class

Method Detail

public static java.lang.String[] **constructRegisterString**(boolean isDeviceNamed)

constructs String[] buffer to hold JmDNS register parameters

Parameters:

if isDeviceNamed is true the name given to device is used, else the default name, X TABLE, is used

Returns:

String[] matrix containing all register values

public static java.lang.String[] **constructDiscoverString**()

constructs String[] buffer to hold JmDNS discover parameters

Returns:

String[] matrix containing all discover values

public static java.lang.String **registerSingle**(java.lang.String type,
java.lang.String name,
int port,
int weight,
int priority,
java.lang.String text)

constructs a buffer to hold one JmDNS service construct

Parameters:

type - JmDNS service type eg. _datagram._udp.local., _device._pid.local.,_http._tcp.local.,_mcast._udp.local.,_mcaststream._udp.local.
name - name of service eg., DatagramServer, deviceName, MulticastReceiver MulticastStream.
port - port number of service
text - text information about service

Returns:

String comma separated concatenation of JmDNS service

public static void **discoverServices**(java.lang.String[] data)

returns line separated discover values as discoverServicesFile

Parameters:

data - String[]

public static void **registerAll**(java.lang.String[] data)

returns line separated register values as registerServicesFile

Parameters:

data - String[]

public static void **initJmDNS**(boolean isDeviceNamed)

This function executes registerAll() and discoverServices() if device has not been initialized or has been reconfigured registerAll() and discoverServices()

Parameters:

if - isDeviceNamed is true execute

B2 com.IMC.coordination
Class StateCoordinator

public class **StateCoordinator**
extends com.IMC.network.DatagramServer
implements java.lang.Runnable

This class inherits datagram connection from DatagramServer for cooperating with supervisor through a FSM. See com.IMC.network.DatagramServer

Field Detail

private com.IMC.coordination.StateBuffer **statebuffer**

---

public static int **trajectory_data**

---

public static int **trajectoryMode**

---

public static boolean **isInterpolatorSocketOpen**

---

public static int **max_trajectory_data_Size**

---

boolean **receive_trajectory**

---

int **state_Flag**

---

public byte[] **in_buffer**

---

public final int **inbuffer_length**

---

public javax.microedition.io.Datagram **in_packet**

---

public javax.microedition.io.Datagram **out_packet**

Constructor Detail

public **StateCoordinator**(com.IMC.coordination.StateBuffer sbuffer)

calls super class to open datagram connection to receive packets on port com.IMC.database.data.DEFAULT_DATAGRAM_PORT_NUMBER

Parameters:

sbuffer - = serverbuffer

Method Detail

public synchronized void **run**()

Thread implementation for servicing datagrams received. State flags are received from supervisor and serviced; eg flag 0; receive single set of trajectory data (pos, vel accl) and flag device FSM to run trajectory right away;

---

private void **serviceTrajProfileDatagram**(int profileType)

Method services trajectory profile data. It uses a FIFO buffer and coordinates with supervisor as data stream in and is consumed. Thus large storage space for trajectory data is not needed

Parameters:

profileType - int; indicates which state or mode of trajectory is desired, ie synchronized, interpolation data, unsynchronized, break.

---

com.IMC.coordination
Class Counter

public class **Counter**

Class implements methods to read and log encoder positions

Field Detail

public static int[] **position**

array to store positions

---

public static int **messageCount**

index used to set granularity of logs; eg log every 3rd point

---

public static int **index**

position index

Method Detail

public static void **posCounter**()

reads position into position[]

---

public static void **posCounter2**()

Logs according to granularity provided. This is used by a logging thread which runs continuously

---

com.IMC.coordination
Class Device

public class **Device**

extends java.lang.Thread

This class makes several calls on drive the com.IMC.drivers.LM628 to command the LM628 for various moves. It also starts DatagramServer and Monitor thread objects and runs a thread with a Finite State Machine to service supervising coordinator's commands

Field Detail

private static com.IMC.coordination.StateBuffer **statebuffer**

---

private static com.IMC.drivers.LM628 **Model**

---

private com.IMC.coordination.StateCoordinator **statecoordinator**

---

public static com.IMC.coordination.Monitor **monitor**

Constructor Detail

public **Device**(com.IMC.coordination.StateBuffer sbuffer)

Instantiates lm628 with serverbuffer as its argument; Starts the dServer with serverbuffer as its argument; Starts the limit switch drivers

Parameters:

sbuffer -

Method Detail

public static void **setPIDfilter**()

Method to set the LM628 PID filter. Filter parameters are stored in sbuffer

---

public synchronized void **run**()

Attempts to initialize lm628. If successful the following are executed; Instantiates monitor with serverbuffer as argument; Starts FSM to service supervising coordinator commands

---

com.IMC.coordination
Class EncoderReader

public class **EncoderReader**
extends java.lang.Thread

implements a low priority thread to read encoder position

Field Detail

private com.IMC.coordination.Monitor **monitor**

---

private static com.IMC.coordination.StateBuffer **serverbuffer**

Constructor Detail

public **EncoderReader**(com.IMC.coordination.Monitor mon,
com.IMC.coordination.StateBuffer sbuffer)

Parameters:

mon -
sbuffer -

Method Detail

public synchronized void **run**()

Method continuously reads encoder position and send it across network to supervisor

---

com.IMC.coordination
Class Interpolation_Server

public class **Interpolation_Server**
extends com.IMC.network.DatagramServer
implements java.lang.Runnable

This class implements a high priority thread to receive streams of multicasted setpoints from an interpolator for coordinated axes moves.

Field Detail

private static int **mode**

mode = 0 => set LM628 to position mode ; mode=1 set LM628 in velocity mode

---

private static int **controlMode**

used in velocity mode to set direction of travel

---

public static int **axisNum**

unique ID of controller supplied by supervisor

---

private static int **bufSize**

datagram byte[] buffer size

---

private com.IMC.drivers.LM628 **lm628**

---

private com.IMC.coordination.StateBuffer **sbuffer**

Constructor Detail

public **Interpolation_Server**(int modes,
boolean flag,
com.IMC.coordination.StateBuffer statebuffer)

calls super class to open datagram connection to receive multicasts on com.IMC.database.PRIORITY_MULTICAST_PORT_NUMBER.; bufSize: 4 bytes for interpolation period, 4 bytes for each coordinated axis. , ie, bufSize = axisNum * 4 + 4;

Parameters:

modes - 0 -> position mode; 1-> velocity mode
flag - true if datagramconnection has already been created
statebuffer - StateBuffer

Method Detail

public void **run**()

Thread's run method creates datagram packet with bufSize Field, creates high priority LM628 object and runs the appropriate profile mode, ie position or velocity

---

void **positionMode**(javax.microedition.io.Datagram packet)

position Mode execution. Datagram containing velocities and positions are received. The ones meant for receiving controller are extracted based on its axisNum ID. Method loads the LM628 controller with trajectory data each time data arrive

Parameters:

packet - Datagram; passed to it by run() method

---

void **velocityMode**(javax.microedition.io.Datagram packet)

velocity Mode execution. Datagram containing velocities and positions are received. The ones meant for receiving controller are extracted based on its axisNum ID. Method loads the LM628 controller with trajectory data each time data arrive. When position direction changes, motor is commanded to change direction of rotation

Parameters:

packet - Datagram; passed to it by run() method

---

com.IMC.coordination
Class Interpolation_Server_Starter

public class **Interpolation_Server_Starter**

Class executes method to initialize and start the Interpolation_Server

Method Detail

public void **startDatagramInterpolator**(int mode,
com.IMC.coordination.StateBuffer statebuffer)

Starts DatagramServerInterpolator

Parameters:

mode - int; 0 ->position mode; 1 -> velocity mode
statebuffer - StateBuffer

---

com.IMC.coordination
Class MainClass

public class **MainClass**
extends com.IMC.coordination.StateBuffer
implements java.lang.Runnable

The main thread starts the controller.

Constructor Detail

public **MainClass**()

Method Detail

public void **run**()

thread's run method creates a ServerBuffer object, serverbuffer and runs com.IMC.coordination.StateBuffer.startController synchronized method. This method waits for lock to be released by JmDNS_Coordinator.class, i.e., for supervising coordinator to logon. When lock is released, MultiCastServer.class and Device.class thread objects are started with serverbuffer as their arguments

---

public static void **main**(java.lang.String[] arg)

main method starts FileServer_ConfigFiles.run(), initiates clock for motion controller (if JStick clock is selected), and calls a NTP timer server with sntpReceiver() to set its base time

Parameters:

arg - String[]

---

public static void **sntpReceiver**()

Logs on to a time server and sets aJile's wall date and clock

---

com.IMC.coordination
Class Monitor

public class **Monitor**
extends com.IMC.network.TCPServer

Creates TCP server socket to send data, synch flags and messages to supervisor

Field Detail

private static java.io.DataOutputStream **dataout**

Constructor Detail

public **Monitor**(com.IMC.coordination.StateBuffer serverbuffer)

Calls super class to create server socket connection on com.IMC.database.DEFAULT_MONITOR_SOCKET_PORT. Starts encoder reader which streams encoder positions

Parameters:

serverbuffer - ServerBuffer

Method Detail

public void **send_data**(int data)

Parameters:

data - int

public static void **send_flag**()

Method sends synchronization flag to supervisor coordinator during synchronized motion

---

public static void **send_Text**(java.lang.String text)

Parameters:

text - String

---

public static void **send_Stop**()

---

public static void **sendEmergencyStop**()

---

public static void **send_initialInfo**()

---

com.IMC.coordination
Class MultiCasted_States

public class **MultiCasted_States**
extends com.IMC.network.MulticastServer
implements java.lang.Runnable

Receives motor STOP, RUN flags from supervisor and puts them in a StateBuffer object

Field Detail

private static com.IMC.coordination.StateBuffer **statebuffer**

---

private static int **mcastport**

Port number to listen on for multicast messages

---

private static java.lang.String **mcastaddress**

IP address to listen on for multicast messages

---

public static int **mcastFlag**

Flag value

Constructor Detail

public **MultiCasted_States**(com.IMC.coordination.StateBuffer s)

Calls super class to join multicast group on com.IMC.database.data.Data.DEFAULT_MULTICAST_IP_ADDRESS and com.IMC.database.data.Data.DEFAULT_MULTICAST_PORT_NUMBER

Parameters:

s - StateBuffer

Method Detail

public void **run**()

Uses the receiver to listen forever for state flags such as trajectory mode, start, stop, from supervisor. Flags are put in StateBuffer object

---

com.IMC.coordination
Class StateBuffer

public class **StateBuffer**
extends com.IMC.database.PushPullData

This Class contains registers which report states eg stop, start. It extends PushPullData @see com.com.IMC.data.PushPullData

Field Detail

private int **stopState**

---

private static boolean **deviceRunFlag**

---

public static boolean **break_flag**

---

private static boolean **check_BusyFlag**

Used in a synchronized method to put EncoderReader in wait

Constructor Detail

public **StateBuffer**()

initializes flags

Method Detail

public synchronized int **get_flag**()

get state of stopState flag. wait and notified() methods used by this method to put calling methods in wait or release them

Returns:

int stopState

---

public synchronized void **put_flag**(int sbyte)

Sets stopState

Parameters:

sbyte - int - if 0, FSM continues (or starts) trajectory; if 1 stops trajectory and puts objects calling on this method to wait for go cmd; if 2 hold and wait for lock to be released. This is used in cases where trajectory data is sent to two or more controllers. Method causes each controller to hold after receiving this data until it receives a multicast 'go' cmd. Multicast is used to effect synchronized motion

---

public synchronized void **get_monitorBusyFlag**()

Used in synchronized move and all high priority communications with supervisor to put objects calling on this method into a wait state

---

public synchronized void **put_monitorBusyFlag**(boolean value)

Used in synchronized move and all high priority communications with supervisor. Methods sets check_BusyFlag

Parameters:

value - boolean

---

public synchronized void **put_deviceRunFlag**(boolean value)

Sets deviceRunFlag.

Parameters:

value - true to release FSM to run, false otherwise

---

public synchronized void **get_deviceRunFlag**()

Method puts Device in wait until lock is released - to save CPU time

---

public synchronized void **startController**()

Method to control start of controller. If controller needs to be restarted because some vital parameters have been changed, method will put calling object in wait state

---

B3 com.IMC.servlets

Class PositionDump

public class **PositionDump**

extends javax.servlet.http.HttpServlet

This servlet is for displaying encoder positions logged by the com.IMC.coordination.Counter in HTML format

Method Detail

public void **init**(javax.servlet.ServletConfig config)

Server calls this method when servlet's URL is requested

Parameters:

config - ServletConfig

Throws:

ServletException -

---

protected void **doGet**(javax.servlet.http.HttpServletRequest req,
javax.servlet.http.HttpServletResponse res)

Queries com.IMC.coordination.Counter for position logs and displays them in HTML format

Parameters:

req - HttpServletRequest
res - HttpServletResponse

Throws:

ServletException -
IOException -

---

com.IMC.servlets

Class ConfigureDevice

public class **ConfigureDevice**

extends com.qindesign.servlet.AuthenticatedHttpServlet

This class executes servlet methods to create html GUI for user to configure the motion controller. It runs on a Tynamo server built for aJile's embedded Java devices

Field Detail

private java.lang.String **filename**

configuration parameters in file /configuration/config.txt
parameters are the field names below

---

private static java.lang.String **device**

---

private static java.lang.String **encoder**

---

private static java.lang.String **switches**

---

private static java.lang.String **mmToRevFactor**

---

private static java.lang.String **PROPORTIONAL**

private static java.lang.String **INTEGRAL**

---

private static java.lang.String **DERIVATIVE**

---

private static java.lang.String **INTEGRAL_LIMIT**

---

private static java.lang.String **DERIVATIVE_SAMPLE**

---

private static java.lang.String **encoderRepInt**

---

private static java.lang.String **speedOverRide**

---

private static java.lang.String **EXCESSIVE_POS**

---

private static java.lang.String **clockSource**

---

private static java.lang.String **restartFlag**

---

private static java.lang.String **home**

---

private static boolean **restart**

restart controller flag

Method Detail

public void **init**(javax.servlet.ServletConfig config)

The Server initiates this servlet with this method when its URL is requested by client. Method reads configuration file into configuration vector

Parameters:

config - ServletConfig

Throws:

ServletException -

---

public java.lang.String **getRealm**(javax.servlet.http.HttpServletRequest req)

Required method. Gets the realm based on the request.

---

public boolean **isAuthorized**(java.lang.String realm,
java.lang.String user,
java.lang.String pass)

Required method. Checks if the given user/password is authorized in the given realm.

---

protected void **doUnauthorizedGet**(javax.servlet.http.HttpServletRequest req,
javax.servlet.http.HttpServletResponse resp)

Unauthorized GET request.

---

protected void **doUnauthorizedPost**(javax.servlet.http.HttpServletRequest req,
javax.servlet.http.HttpServletResponse resp)

Unauthorized POST request.

---

protected void **doGet**(javax.servlet.http.HttpServletRequest req,
javax.servlet.http.HttpServletResponse res)

generates html of controller configuration and handles GET requests from users (clients), ie generates HTML forms and text for users to change values; has control buttons for users to call doPost and also set controller up for desired configuration; Modifies configuration file and saves it in nonvolatile memory; alerts user to restart controller if configuration changes eg clock speed are critical.

Parameters:

req - HttpServletRequest; used to read incoming HTTP headers and HTML form data
res - HttpServletResponse; used to specify the HTTP response line and headers

Throws:

ServletException -
IOException -

---

protected void **doPost**(javax.servlet.http.HttpServletRequest req,
javax.servlet.http.HttpServletResponse res)

doPost method generates html form for client to change controller configuration such as PID filter values, clock speed, etc.

Parameters:

req - HttpServletRequest
res - HttpServletResponse

Throws:

ServletException -
IOException -

---

private void **save**()

Method to save configuration file

Throws:

IOException -

---

private void **reset**()

Method to reset configuration file to default

Throws:

IOException -

---

com.IMC.servlets
Class ControllerInfo

public class **ControllerInfo**
extends javax.servlet.http.HttpServlet

This Servlet displays information about controller services and configuration

Field Detail

private java.util.Vector **applications**

---

private java.lang.String **filename**

---

private com.IMC.database.FileService **file**

---

private static java.util.Vector **configuration**

Method Detail

public void **init**(javax.servlet.ServletConfig config)

The Server initiates this servlet with this method when its URL is requested by client. Method reads configuration files into Vectors

Parameters:

config - ServletConfig

Throws:

ServletException -

---

protected void **doGet**(javax.servlet.http.HttpServletRequest req,
javax.servlet.http.HttpServletResponse res)

Parameters:

req - HttpServletRequest

res - HttpServletResponse

Throws:

ServletException -

IOException -

---

com.IMC.servlets
Class EditJmDNS

public class **EditJmDNS**

extends javax.servlet.http.HttpServlet

This class executes servlet methods to create html GUI for user to configure JmDNS 'discover' and 'register' services for this device It runs on a Tynamo server built for aJile's embedded Java devices

Field Detail

private static java.util.Vector **viewListener**

Vector to hold JmDNS 'discover' data

---

private static java.util.Vector **viewRegister**

Vector to hold JmDNS 'register' data

Method Detail

public void **init**(javax.servlet.ServletConfig config)

The Server initiates this servlet with this method when its URL is requested by client; Method reads contents of JmDNS 'discover' and 'register' files into viewListener and viewRegister Vectors

Parameters:

config - ServletConfig

Throws:

ServletException -

---

protected void **doGet**(javax.servlet.http.HttpServletRequest req,
javax.servlet.http.HttpServletResponse res)

generates html of JmDNS services on network and also services registered by controller; processes data posted by doPost(); has control buttons for users to call doPost

Parameters:

req - used to read incoming HTTP headers and HTML form data

res - used to specify the HTTP response line and headers

Throws:

ServletException -

IOException -

protected void **doPost**(javax.servlet.http.HttpServletRequest req,
javax.servlet.http.HttpServletResponse res)

Generates HTML forms for clients to change JmDNS settings, register services and also listen for specified services on the network

Parameters:

req - HttpServletRequest

res - HttpServletResponse

Throws:

ServletException -

IOException -

---

com.IMC.servlets
Class JmDNS_Coordinator

public class **JmDNS_Coordinator**

extends javax.servlet.http.HttpServlet

The Server initiates this servlet automatically when the server is started. This is the main tool for the controller's PnP reconfigurability. When the controller starts the Server, this servlet listens for services on the network, ie, the Supervisor and other nodes on the network. Service info received contains all parameters to enable this controller to communicate appropriately. When services are removed the servlet notifies the controller. The servlet also registers or publishes services it provides eg motion control. Method reads configuration file into configuration vector

Parameters:

config - ServletConfig

Throws:

ServletException -

Field Detail

private static java.util.Vector **addBuffer**

holds services discovered

---

private static java.util.Vector **removeBuffer**

services removed from network

---

private static java.util.Vector **resolveBuffer**

services resolved

---

private static java.util.Vector **viewListener**

controller listens for services in this Vector

---

private static java.util.Vector **viewRegister**

controller registers services in this Vector

---

public static javax.jmdns.JmDNS **jmdns**

---

private static java.lang.String **supervisorIP**

Supervisor's IP is stored by this string

---

private static int **supervisorPort**

---

public static java.util.Vector **deviceName**

array for device names discovered on network

public static boolean **isServiceRegistered**

---

private static java.util.Vector **listenerVector**

holds listeners for viewListener Field

Method Detail

public void **init**(javax.servlet.ServletConfig config)

This is called immediately when Server starts. Method starts JmDNS, executes methods to listen and discover services. If controller has been rebooted after a reconfiguration procedure which requires a reboot, the servlet registers its services. The servlet checks Constants.initializeController flag for this.

Parameters:

config - ServletConfig

Throws:

ServletException -

---

public static void **discoverJmDNS**()

Method executes a JmDNS service listener for each service to be discovered

---

public static void **registerJmDNS**()

Method calls registerJmDNS() for each service in viewRegister

---

protected void **doGet**(javax.servlet.http.HttpServletRequest req,
javax.servlet.http.HttpServletResponse res)

Displays HTML showing services registered and discovered

Parameters:

req - HttpServletRequest

res - HttpServletResponse

Throws:

ServletException -

IOException -

---

public static void **registerAService**(java.lang.String type,
java.lang.String name,
int port,
int weight,
int priority,
java.lang.String text)

Method is called by registerJmDNS() for each service to be registered

---

public static void **unregisterAll**()

Method used to unregister all services

B4 com.IMC.drivers

Class Reference_Switch_Driver

public class **Reference_Switch_Driver**

Reference switch driver.

Constructor Detail

public **Reference_Switch_Driver**(java.lang.String device)

Moves motion device to its hardware reference switch. Implements a TriggerEventListener to service interrupt on the reference position pin. Also used to move device to its home position

Parameters:

device - String represents motion device. Home position for device is stored in its configuration file

---

Method Detail

void **GotoHomePosition**()

Finds home position relative to reference switch as defined in configuration file

---

com.IMC.drivers
Class GPIOPinA3

public class **GPIOPinA3**

Class to assert JSimm pin A3 .

Field Detail

public static com.ajile.drivers.gpio.GpioPin **pin3**

Creates General Purpose I/O pin A3

Method Detail

public static boolean **state**()

Returns:

true if pin is high; false if pin is low

---

public static boolean **enable**()

Configures pin as output pin and drives it low

Returns:

true

---

public static boolean **disable**()

Configures pin as output pin and drives it high

Returns:

true

---

com.IMC.drivers
Class HSIO_Driver

public class **HSIO_Driver**

Sets up HSIO pins and configures the HSIO for read/write operations

Field Detail

private static final int **HSIO_CS0_ADDRESS**

Chips Select CS0 address = 0x01400000;

---

private int **HSIOPortAddress**

HSIOPortAddress = HSIO_CS0_ADDRESS

Constructor Detail

public **HSIO_Driver**(int PinAddress,

byte A19_A16,
int A20)

creates a new HsioPort with port address PinAddress and a clock divider of A19_A16 on chip select 0. Wait bit is A20

Parameters:

PinAddress - The new HSIO address of the port.
A19_A16 - clock divider bits of the HSIO address + 1.
A20 - wait bit

---

public **HSIO_Driver**(int PinAddress,
byte A19_A16,
int A20,
boolean SelectCS1)

creates a new HsioPort with port address PinAddress and a clock divider of A19_A16 on chip select 0 or 1. Wait bit is A20

Parameters:

PinAddress - The new HSIO address of the port.
A19_A16 - clock divider bits of the HSIO address + 1.
A20 wait bit
SelectCS1 - true if CS1 false if CS0 should be selected

Method Detail

public void **setHsioAddress**(int PinAddress)

Sets HSIO address in HSIO address space. It verifies that the address is only 12 bits and throws an IllegalArgumentException if it is not.

Parameters:

PinAddress - int

Throws:

IllegalArgumentException -

---

public void **setHsioTiming**(byte A19_A16)

sets the HSIO timing bits [19:16] in the JStik address to the low order bits of A19_A16.

Parameters:

A19_A16 - byte

Throws:

IllegalArgumentException - if there are more than 4 bits

---

public void **setHsioA20**(int Tas)

sets the HSIO wait state, A20. Can be 0 or 1. Adds one CLKO to the setup time from address asserted to RD or WR asserted. Also adds one CLKO period to the write hold time (WR negated to WAIT negated).

Parameters:

Tas - int

Throws:

IllegalArgumentException - if more than 1 bit ie decimal 1

---

public void **setChipSelect**(boolean SelectCS1)

Sets the chip select bit in the port address. *

Parameters:

SelectCS1 - if true, chip select 1 is selected. If false, chip select 0 is selected

---

public int **getRawJStikAddress**()

returns HSIOPortAddress for use with RawJem reads and writes without overhead.

Returns:

int address of HSIOPortAddress

---

public void **setRawJStikAddress**(int JStikAddress)

Verifies that the address is a valid HSIO address in the JStik address space. A bad address can write to arbitrary places in memory and cause object and heap corruption.

---

public int **read**()

reads a byte from the HSIO port.

Returns:

The return value is an integer for performance reasons.

---

public void **write**(int data)

writes a byte to the HSIO port. data is an int to improve performance over the read, modify, write implementation of the JVM. The HSIO data bus is only 8 bits wide anyway, so the high order 24 bits are not transmitted.

Parameters:

data - is the byte to be transmitted.

---

com.IMC.drivers
Class JStickTimer_tc2

public class **JStickTimer_tc2**

Usage of the 3rd aJ100 General Purpose Timer/Counter hardware with output on pin 15. Low Level Device Driver classes. This is used as an alternate clock source for the LM628. The neat thing is flexibility of clock control. Different clock speeds can be configured on this timer and diff sorts of interrupts may be received and serviced accordingly. LM628 requires clock speeds of between 1 and 6MHz inclusive. Jumper JP1 on the controller board has to be inserted on pins CLK2 and OUT!!

Field Detail

private com.ajile.drivers.gptc.TimerCounter **tc2**

---

private static int **prescalerReloadRegisterValue**

---

private static int **reloadRegisterValue**

Constructor Detail

public **JStickTimer_tc2**(double freqMHz)

Configures the hardware timer 2 to generate frequency of 6MHz or less to clock the LM628.

Method Detail

public static void **main**(java.lang.String[] args)

Create an Example instance then perform some operations on that instance.

---

public static double **freqCalculator**(int PrescalerReloadRegisterValue,
int ReloadRegisterValue)

method to calculate freq. given PrescalerReloadRegisterValue and setReloadRegisterValue

Parameters:

PrescalerReloadRegisterValue - int

ReloadRegisterValue - int

Returns:

double frequency in MHz

public static void **prescalerReloadValues**(double freqMHz)

This function calculates PrescalerReloadRegisterValue and setReloadRegisterValue for symmetrical square waves.

Parameters:

freqMHz - is the desired frequency

Throws:

java.lang.IllegalArgumentException - if frequency>6MHz

---

com.IMC.drivers
Class LimitSwitch_Left

public class **LimitSwitch_Left**

Contains methods to service interrupts received on left limit switch

Field Detail

final com.ajile.drivers.gpio.GpioPin **pinA3**

set up GPIO pin connected to left limit switch

---

com.IMC.coordination.Monitor **monitor**

Monitor object sends interrupt messages to supervising coordinator

---

Constructor Detail

public **LimitSwitch_Left**(com.IMC.coordination.Monitor mon)

Funtion sets up a TriggerEventListener to receive and service interrupts Also debounces pin to avoid sporadic responses

Parameters:

mon - Monitor

Method Detail

void **send**()

Used by Constructor to send emergency message if limit switch is triggered

---

com.IMC.drivers
Class LimitSwitch_Right

public class **LimitSwitch_Right**

Contains methods to service interrupts received on right limit switch

Field Detail

final com.ajile.drivers.gpio.GpioPin **pinA4**

set up GPIO pin connected to right limit switch

---

com.IMC.coordination.Monitor **monitor**

Monitor object sends interrupt messages to supervising coordinator

---

Constructor Detail

public **LimitSwitch_Right**(com.IMC.coordination.Monitor mon)

Funtion sets up a TriggerEventListener to receive and service interrupts Also debounces pin to avoid sporadic responses

Parameters:

mon - Monitor

Method Detail

void **send**()

Used by Constructor to send emergency message if limit switch is triggered

---

com.IMC.drivers
Class LM628

public class **LM628**

This class contains functions for commanding the LM628 and sending/receiving data from/to it. Field values are commands. Refer to the LM628 manual for definitions

Field Detail

public static final int **RESET**

---

public static final int **RSTI**

---

public static final int **DFH**

---

public static final int **SIP**

---

public static final int **LPEI**

---

public static final int **LPES**

---

public static final int **SBPA**

---

public static final int **SBPR**

---

public static final int **MSKI**

---

public static final int **LFIL**

---

public static final int **UDF**

---

public static final int **LTRJ**

---

public static final int **STT**

---

public static final int **RDSIGS**

---

public static final int **RDIP**

---

public static final int **RDDP**

---

public static final int **RDRP**

---

public static final int **RDDV**

---

public static final int **RDRV**

---

public static final int **RDSUM**

---

public static final int **PORT12**

---

private static final int **addr**
    address A0 of JStick HSIO

---

private static final byte **div**
    HSIO Timing value; div = 1

---

private static final int **tas**
    HSIO Wait value; tas =1;

---

private static final int **addrBase**
    base address of JStick HSIO

---

public static double **NLINE**
    number of encoder lines

---

private static double **CLK**
    LM628 clock value

---

private static int **SMP_RT**
    Sampling rate

---

public static int **rdataLM628**
    variable for storing data read from LM628

---

public static int **rdstatusLM628**
    variable for storing data LM628 status info

---

private static final com.IMC.drivers.HSIO_Driver **HSIO_ADDRESS_A0**
    HSIO_Driver object for address A0

---

private static com.IMC.drivers.HSIO_Driver **HSIO_ADDRESS_BASE**
    HSIO_Driver object for base address

---

private static com.IMC.coordination.StateBuffer **statebuffer**

---

private static double **pos_Constant**
    variable to store constant for calculating position

---

private static double **vel_Constant**
    variable to store constant for calculating velocity

---

private static double **accl_Constant**
    variable to store constant for calculating acceleration

Constructor Detail

public **LM628**()
    default constructor

---

public **LM628**(com.IMC.coordination.StateBuffer sbuffer)

constructor initializes clock value, number of encoder lines and position, velocity and acceleration constants

Parameters:

sbuffer=statebuffer -

Method Detail

public static void **check_busy_bit**()

Polls status byte until busy bit is cleared by the chip ie, = 0

---

public static void **write_command**(int CMD)

function writes command to the LM628 base address. On LM628 PS->low, CS->low, WR->low On JStick A0->low, CS0->low, WR->low. Each address pin (A0-A11) holds low if there's no RD/WR to it.

Parameters:

CMD - command value as in Fields

---

public static void **write_data**(int data)

Function writes data to the LM628 to address space A0. On LM628 PS->high, CS->low, WR->low On JStick A0->high, CS0->low, WR->low.

Parameters:

data - in integer form

---

public static int **read_data**()

Function reads data from the LM628 address space A0. On LM628 PS->high, CS->low, RD->low On JStick A0->high, CS0->low, RD->low.

Returns:

value read

---

public static int **read_status**()

Function reads the LM628 status register from base address. On LM628 PS->low, CS->low, RD->low. On JStick A0->low, CS0->low, RD->low.

Returns:

value read

---

public static int **read_signals_register**()

reads the LM628 16 bit signals register

Returns:

signals_reg

---

public static void **chk_motoroff**()

polls status register until motor off bit is set (1)

---

public static void **wait_traj_bit**()

polls status register bit until trajectory end bit is set (1) or a stop signal is received from a state machine

---

public static void **chk_breakpt**()

polls status register bit until breakpoint bit is set (1) or a stop signal is received from a state machine

---

public static void **waitTrajBit_log**()

polls status register bit until trajectory end bit is set (1) or a stop signal is received from a state machine. In the meantime it logs positions

---

public static void **chkBreakpt_log**()

polls status register bit until breakpoint bit is set (1) or a stop signal is received from a state machine. In the meantime it logs positions

---

public static void **breakpoint**(double FIN_POS,
boolean isAbsolute)

This method loads a breakpoint position as relative or absolute

Parameters:

FIN_POS - double ; final position. This is scaled up (see LM628 documentation) and type cast as an integer. Since Java integer is 32-bits long, method splits FIN_POS into 4 bytes and writes them a byte at a time starting with the MSB
isAbsolute - boolean; true if FIN_POS is absolute

---

public static synchronized void **traj_sel**(int high_byte,
int low_byte,
double FIN_POS,
double VEL_FIN,
double ACCL)

The Trajectory command is followed by a 2-byte data containing motion configuration parameters, eg load relative position, position or velocity mode, etc. This is followed by the trajectory data in the order acceleration position velocity. Each is 32 bits (integer) so method splits each into 4 bytes and writes them starting with the MSB.

Parameters:

high_byte - high byte of trajectory command data
low_byte - low byte of trajectory command configuration data
FIN_POS - Final position
VEL_FIN - Final velocity
ACCL - Final acceleration see traj_sel method above

---

public static synchronized void **traj_sel_2**(int control_bytes,
double FIN_POS,
double VEL_FIN,
double ACCL)

The Trajectory command is followed by a 2-byte data containing motion configuration parameters, eg load relative position, position or velocity mode, etc. This is followed by the trajectory data in the order acceleration position velocity. Each is 32 bits (integer) so method splits each into 4 bytes and writes them starting with the MSB.

Parameters:

control_bytes - int; trajectory command configuration data. This is 2 bytes so it is split into 2 bytes
FIN_POS - Final position
VEL_FIN - Final velocity
ACCL - Final acceleration see traj_sel method above

---

public static synchronized void **traj_sel_3**(double FIN_POS,
double VEL_FIN)

This loads only position (in revs) and velocity (rev/s): trajectory control byte configures for absolute position and velocity

Parameters:

FIN_POS - Final position
VEL_FIN - Final velocity see traj_sel method

---

public static synchronized void **traj_sel_abs_Rel**(double FIN_POS,
double VEL_FIN,
boolean isAbsolutePos,
boolean isAbsoluteVel)

Method loads only position (in revs) and velocity (rev/s):

Parameters:

FIN_POS - Final position
VEL_FIN - Final velocity
isAbsolutePos - true for absolute position or false for relative position
isAbsoluteVel - true for absolute velocity or false for relative velocity see traj_sel method

---

public static synchronized void **trajSel_Vel**(double VEL_FIN,
boolean isAbsolute)

Method loads only velocity (mm/s) in velocity mode.

Parameters:

FIN_POS - Final position
VEL_FIN - Final velocity
isAbsolute - true for absolute velocity or false for relative velocity see traj_sel method

---

public static synchronized void **traj_sel_4**(int mode,
double VEL_FIN)

Method loads only velocity (mm/s) in velocity mode.

Parameters:

mode - used by trajectory configuration to determine motor direction
VEL_FIN - Final velocity see traj_sel method

---

public static synchronized void **traj_sel_home**(double VEL_FIN,
double ACCL)

Method loads trajectory in velocity mode for moving motion device to its home position

Parameters:

VEL_FIN - Final velocity
ACCL - acceleration see traj_sel method

---

public static void **filter_sel**(int Kp,
int Ki,
int Kd,
int Il,
int CLK_SC)

Programs the PID Filter

Parameters:

Kp - Proportional term
Ki - Integral
Kd - Derivative
Il - Integral limit
CLK_SC - programs derivative sampling rate

---

public static synchronized void **run_motor**()

Function to run motor

---

public static void **define_home**()

Defines home (position 0)

---

public static void **excessivePosError**(int EncoderCounts)
method sets the condition for detecting excessive position error
Parameters:
EncoderCounts - int

---

public static int **readIndex**()
method for reading position recorded in the index register
Returns:
int; index position

---

public static int **readPosition**()
reads real position
Returns:
int; position

---

public static int **readVel**()
read real velocity
Returns:
int; velocity

---

public static int **readDesiredPosition**()
method to read desired position
Returns:
int; desired position

---

public static void **reset_interrupt_register**(int value)
Method to set interrupt register
Parameters:
value - register value desired. Zero in an interrupt register bit position resets the corresponding interrupt

---

public static boolean **initialize**()
Method for doing hardware and interrupt resets
Returns:
boolean true if successful

---

public static void **mask_reg**(int mask_bits)
method for masking interrupt bits in interrupt register
Parameters:
mask_bits - int

---

public static void **stop_Anywhere**()
method to halt motor abruptly

---

public static void **set_DAC**()
Method for calibrating the DAC [AD667]. The motor driver should be off. 1. set all bits to 0 by issuing a reverse velocity command and adjust the offset trimmer until the output is -10.00V. 2.

set all bits to 1 by issuing fwd velocity command and adjust the gain trimmer until the output is 9.9976V.

---

public static void **reset**()

Method to do a soft reset of the LM628

---

com.IMC.drivers
Class Board_Clock

public class **Board_Clock**

Class to enable/disable the onboard clock. Its enable pin is hooked to JStick SIMM pin A6.

Method Detail

public static boolean **disable**()

writes a zero to pin A6 to disable the onboard clock

Returns:

boolean; true

---

public static boolean **enable**()

writes a 1 to pin A6 to enable the onboard clock

Returns:

boolean; true

---

com.IMC.drivers
Class LM628_Interrupt

public class **LM628_Interrupt**

Contains method to receive and service hardware interrupts from the LM628

Field Detail

final com.ajile.drivers.gpio.GpioPin **pinA1**

---

com.IMC.coordination.Monitor **monitor**

Monitor.class object transmits interrupt condition to supervisor

---

Constructor Detail

public **LM628_Interrupt**(com.IMC.coordination.Monitor mon)

Implements TriggerEventListener to receive and service interrupts

Parameters:

mon - Monitor = monitor as in field

Method Detail

void **send**()

Sends interrupt info to supervisor

---

com.IMC.drivers
Class MotorAmp

public class **MotorAmp**

Class to enable/disable motor drivers. Motor amp's enable pin is hooked to JSimm pin A5 through an inverter since pins are logic high when Jstick is powered.

Method Detail

public static boolean **enable**()
    writes 0 to pin A5 to drive it low
Returns:
    boolean true

---

public static boolean **disable**()
    writes 1 to pin A5 to drive it high
Returns:
    boolean; true

B5 com.IMC.network
Class TCPServer

public class **TCPServer**
    Class creates a TCP server socket stream connection

Field Detail

public javax.microedition.io.StreamConnectionNotifier **scn**

---

public javax.microedition.io.StreamConnection **connection**

Constructor Detail

public **TCPServer**(int server_port)
Parameters:
    server_port - int

---

com.IMC.network
Class DatagramServer

public class **DatagramServer**
    This class provides a CLDC DatagramConnection for datagram transmissions

Field Detail

public javax.microedition.io.DatagramConnection **dgconn**

Constructor Detail

public **DatagramServer**(int datagram_port)
Parameters:
    datagram_port - int

---

com.IMC.network
Class MulticastServer

public class **MulticastServer**
    Creates a Multicast socket

Field Detail

public javax.microedition.io.MulticastConnection **mSocket**
    Multicast Socket

---

public int **mGroup**

Internet address group

Constructor Detail

public **MulticastServer**(java.lang.String multicast_address,
int multicast_port)

Parameters:

multicast_address - String
multicast_port - int

APPENDIX C: COORDINATOR SOFTWARE INTERFACE

The System Coordinator domain handles all supervisory activities to guarantee the appropriate execution of tasks on the IMC controllers. All high-level tasks and commands are generated in this domain. These include human-machine interactions, system configuration, and Meta tasks such as "complex" inverse kinematics, which cannot be handled by the IMC nodes or the real-time coordinator. Table C1 shows the packages in this domain and their corresponding classes. The class interfaces are described in this appendix.

Table C1: System Coordinator Software

Package	**Description**	**Classes**
com.coordinator.GUI	Abstracts the Human-Machine Interface	*MainApplication* *MainGUIFrame* *TrajDataFrame* *TrajTable* *PIDTable*
com.coordinator.coordination	Protocols for commanding and coordinating activities	*JmDNS_Coordinator* *JmDNS_Event_Server* *ControllerIO* *Monitor* *SynchFlag* *Trajectory_Server*
com.coordinator.database	Temporary and permanent global repository	*Data* *JmDNS_DATA* *Traj_Configuration_Data* *GCodeParser* *GCodeSende*
com. coordinator.interpolation	Abstracts interpolation and kinematics algorithms	*Interpolator* *Transmission* *Transmission_ACK* *Transmission_Flag*
com. coordinator.network	Abstracts protocols for communication	DatagramSender McastDirect MulticastSender TCP_Client UDP_Client

C1 com.coordinator.database

Class Traj_Configuration_Data

public class **Traj_Configuration_Data**

This class contains methods for initializing class for constructing trajectory set-points in TrajDataFrame.class

Field Detail

public static java.io.File **getFileDirectory**

trajectory file "getFileDirectory" is used to find the parent directory

---

public static java.lang.String **AXES**

---

public static java.lang.String **ACCELERATION**

---

public static java.lang.String **INTERPOLATION_PERIOD**

---

public static java.lang.String **INTERPOLATION_TYPE**

---

public static java.lang.String **INTERPOLATION_PLATFORM**

---

public static java.lang.String **COMMUNICATION_MODE**

---

public static java.lang.String **CONTROLLER_MODE**

---

public static java.lang.String **isAbsolute**

---

public static java.lang.String **isAbsoluteVel**

---

public static java.lang.String[] **AXIS_STRING**

"1","2","3","4","5","6","TRIPOD (5 AXIS)","PUMA" show up in combo box in TrajDataFrame class *

---

public static final java.lang.String[] **INTERPOL_TYPE_FIXED_STRING**

"breakpoint","nonbreakpoint", "nonbreakpoint (synchronized)","G CODE" show up in combo box in TrajDataFrame class *

---

public static java.lang.String[] **INTERPOL_TYPE_STRING**

---

public static java.lang.String[] **INTERPOLATION_PLATFORM_STRING**

"Local Host", "JStick Host" show up in combo box in TrajDataFrame class

---

public static java.lang.String[] **COMMUNICATION_MODE_STRING**

"via RT JStick Host", "Multicast Direct" show up in combo box in TrajDataFrame class

---

public static java.lang.String[] **CONTROLLER_MODE_STRING**

"via RT JStick Host", "Multicast Direct" "Position Mode", "Velocity Mode"

---

public static int **counterSize**

controllers set their log buffers to this size

---

public static int **counterGranul**

granularity or how often controllers log positions. This doesn't apply to GCode interpolation mode

---

Constructor Detail

public **Traj_Configuration_Data**(java.io.File getFileDirectory)

Parameters:

getFileDirectory - File; configuration file directory obtained from TrajDataFrame

---

Method Detail

public static void **writeFile**()

this methods saves configuration parameters to trajConfig.txt which is in the same directory as user defined or selected trajectory data file

---

public static void **Configuration_Data**()

trajectory file "getFileDirectory" is used to find the parent directory this method attempts to fetch configuration parameters from trajConfig.txt. The parameters are used to update the TrajDataFrame GUI. If the file does not exists, default parameters are used and the trajConfig.txt is created with these values

---

com.coordinator.database

Class Data

public class **Data**

Class contains all network connection fields and control mode keys and values for hashtable it creates.

Field Detail

public static final int **MONITOR_PORT**

---

public static final int **MONITOR_PORT_TIMEOUT**

---

public static final int **UDPdirectPort**

---

public static final int **TCPdirectPort**

---

private static java.lang.String **MULTICAST_IP_ADDRESS**

---

private static int **MULTICAST_PORT_NUMBER**

---

public static boolean **isJmDNSAlive**

---

public static int **interpolatorTCPPort**

---

public static int **interpolatorUDPPort**

---

public static int **interpolatorUDPStreamPort**

---

public static java.lang.String **interpolatorIP**

---

public static int **McastDirectPort**

---

public static java.lang.String **McastDirectIP**

---

public static final java.lang.String[] **DEVICES**

---

public static java.lang.String **AXIS_1_IP**

---

public static java.lang.String **AXIS_2_IP**

---

public static java.lang.String **AXIS_3_IP**

---

public static java.lang.String **AXIS_4_IP**

---

public static java.lang.String **AXIS_5_IP**

---

public static java.lang.String **AXIS_6_IP**

---

public static int **AXIS_1_PORT**

---

public static int **AXIS_2_PORT**

---

public static int **AXIS_3_PORT**

---

public static int **AXIS_4_PORT**

---

public static int **AXIS_5_PORT**

---

public static int **AXIS_6_PORT**

---

public static int **AXIS_1_WEB_PORT**

---

public static int **AXIS_2_WEB_PORT**

---

public static int **AXIS_3_WEB_PORT**

---

public static int **AXIS_4_WEB_PORT**

---

public static int **AXIS_6_WEB_PORT**

---

public static java.lang.String[] **CONTROL_MODES**

control modes; hastbale keys

---

public static int[] **CONTROL_FLAGS**

holds values for hashtable keys

Method Detail

public static int **getControlFlag**(java.lang.String flag)

Stores and retrieves control flags from hashtable

Parameters:

flag - String

Returns:
int; flag

---

public static int **getDeviceID**(java.lang.String device)
Stores ID of device detected on network to hashtable
Parameters:
device - String
Returns:
int; device ID
Throws:
Exception -

---

public static int **getDevicePort**(java.lang.String device)
Stores port nos. of devices detected on network in a hashtable
Parameters:
device - String
Returns:
int; port number
Throws:
Exception -

---

public static java.lang.String **getDeviceIP**(java.lang.String device)
Stores IP addresses of devices detected on network in a hashtable
Parameters:
device - String
Returns:
String; device
Throws:
Exception -

---

public static java.lang.String **putDeviceIPInfo**(java.lang.String device
java.lang.String IP_Address,
int port)
Stores IP address of device in hashIP and port number in hashPort
Parameters:
device - Device name; IP_Address - IP address; port – port number
Throws:
Exception -

---

public static int **getMulticastPort**()
Returns:
int; main multicaster port number

---

public static void **putMulticastPort**(int port)
Parameters:
port - int; main multicaster port number

---

public static java.lang.String **getMulticastIP**()
Returns:
String; main multicaster IP address

---

public static void **putMulticastIP**(java.lang.String ip)

Parameters:

ip - String; main multicaster IP address

---

public static java.lang.String **getMcastDirectIP**()

Returns:

String; IP address for multicaster streaming setpoints

---

public static void **putMcastDirectIP**(java.lang.String ip)

Parameters:

ip - String; IP address for multicaster streaming setpoints

---

public static int **getMcastDirectPort**()

Returns:

int; port number for multicaster streaming setpoints

---

public static void **putMcastDirectPort**(int port)

Parameters:

port - int; port number for multicaster streaming setpoints

---

public static int[] **getInterpolatorPorts**()

Returns:

int[]; holds interpolator's port numbers

---

public static void **putInterpolatorPorts**(int tcpport,
int udpport,
int udpstreamport)

Parameters:

tcpport - int; interpolator's TCP port
udpport - int; interpolator's UDP port
udpstreamport - int; interpolator's UDP port for setpoint streaming

---

public static java.lang.String **getInterpolatorIP**()

Returns:

String; interpolator's IP address

---

public static void **putInterpolatorIP**(java.lang.String ip)

Parameters:

ip - String; puts interpolator's IP address in interpolatorIP

---

com.coordinator.database
Class DataTranspose

public class **DataTranspose**

Class has a method for transposing data in a file

Constructor Detail

public **DataTranspose**(java.io.File file,
java.lang.String outFileName)

Parameters:

file - File; input file
outFileName - String

Method Detail

public static void **main**(java.lang.String[] args)
main method executes DataTranspose

Parameters:

args - String[]

---

com.coordinator.database

Class FileDataToArrayConverter

public class **FileDataToArrayConverter**

Implements method to read trajectory data from file and store them in arrays according to the number of controllers

Method Detail

public static void **dataServer**(int type,
java.io.File trajFile,
int[] dataDim)

Method called by TrajDataFrame.class to decompose trajectory data into arrays.

Parameters:

type - int; type=0->breakpoint, nonbreakpoint or synchronized trajectory mode type
trajFile - File; trajectory file
dataDim - int[]; array containing row and column sizes

---

com.coordinator.database

Class GCodeParser

public class **GCodeParser**

This class parses NC G Code for interpolation.

Field Detail

public static java.io.BufferedReader **in**

---

public static java.io.DataOutputStream **dout**

---

public static java.io.File **file**

Constructor Detail

public **GCodeParser**(java.io.File gfile)

Parameters:

gfile - File; G Code file

Method Detail

public void **gCodeParser**()

Method to parse G Code into another file with the same name (and directory as the G Code file but with .bin extension ; In brief procedure is as ff; File ignores all comments prefixed to G Code; G Code commands are in 3 categories; motion parameters (eg G01, G02, G03, G00), state

(eg G90); coordinates (eg X90). Motion parameters are prefixed to coordinates by method, eg G01X90. If another motion parameter is read the previous is overwritten, eg G02X10.

---

com.coordinator.database
Class GCodeSender
public class **GCodeSender**

Sends parsed NC G code by TCP to JStick module designated for realtime interpolation This method tends to cause the module to draw excess current, likely due to the Ethernet controller set to TCP mode for this transaction. This causes the control module sharing the backplane to reset several times. Hence this method should probably be used only when the interpolator module has its own backplane

Method Detail

void **gCodeParser**()

See Also:

GCodeParser for implementation details. Coordinates in this method are converted to integers by scaling them by 1000. This is because it is computationally efficient for JSticks to receive datagrams containing integers than doubles.

---

com.coordinator.database
Class JmDNS_DATA
public class **JmDNS_DATA**
extends com.coordinator.database.Data

Class holds JmDNS data

Field Detail

public static java.lang.String[] **register**

array for services to be registered

---

public static java.lang.String[] **discover**

Constructor Detail

public **JmDNS_DATA**()

Fills register and discover arrays with data . Example register[0] = "_supervisor._tcp.local.,SUPERVISOR,MONITOR_DEFAULT_PORT,0,0"; discover[0] = "_http._tcp.local."; contains address and port of controller's server; connection info is used by GUI to create links discover[1] = "_mcast._udp.local."; contains address and port of controller's multicast receiver to connect with com.coordinator.network.MulticastSender discover[2] = "_device._pid.local.";contains address and port of controller's datagram server to connect with com.coordinator.network.DatagramSender. discover[3] = "_mcaststream._udp.local."; connection information for McastDirect

See Also:

com.coordinator.network.MulticastSender, com.coordinator.GUI.MainGUIFrame, com.coordinator.database.Data, com.coordinator.network.McastDirect, com.coordinator.network.DatagramSender

---

C2 com.coordinator.coordination
Class ControllerIO
public class **ControllerIO**

extends com.coordinator.network.DatagramSender

Regulator for setting up values and parameters to be sent to Controllers via datagram

See Also:

com.coordinator.network.DatagramSender.

Constructor Detail

public **ControllerIO**()

creates trajectory array for holding pos, vel and accl for each device. This is used for single point (jog) mode

---

Method Detail

---

public void **jog**(java.lang.String device,
 java.lang.String flag)

Method for jogging axis or axes.

Parameters:

device - String
flag - String;

---

public void **shutdownController**(java.lang.String device,
 java.lang.String flag)

method to send shutdown message to individual or all controllers

---

public static void **setMulticast**(java.lang.String flag)

method to send multicast run or stop flag to devices on the network

---

public static void **setDatagram**(java.lang.String flag,
 int numaxes,
 double period,
 double accl)

sends control flag and data to real-time interpolator. This interpolator may be put in a state to wait for G code or receive setpoints from local interpolator

Parameters:

flag - String; eg interpolator_viaRT_flag
numaxes - int; number of coordinated axes
period - double; period of interpolation
accl - double; acceleration

---

public static void **setDatagram**(java.lang.String flag,
 int[][] pid)

For sending user-defined PID filter values

Parameters:

flag - String; if flag ==pidfilter send filter values to registered devices
pid - array pid[m][n] for devices

---

public void **setDatagram**(java.lang.String flag,
 double accl,
 int logSize)

Parameters:

flag - String; trajectory mode eg interpolator_gcode_velocitymode
accl - double; acceleration
logSize - int; size of log file

---

public void **setDatagram**(java.lang.String flag,
int speedOverRide)

sends speed Over-Ride value

---

public void **setDatagram**(java.lang.String flag)

for sending various signal (modes) to set the state of controllers. Signals are derived from com.coordinator.network.Database hastable keys

Parameters:

flag - String; key eg "drives_off"

---

com.coordinator.coordination
Class Trajectory_Server

public class **Trajectory_Server**
implements java.lang.Runnable

When host PC or JStick is not used for explicit online interpolation, as in non-breakpoint, breakpoint and synchronized_non-break point modes, this thread class coordinates streaming of coarse trajectory data by datagram to controllers. When controller's receive buffers are full it informs its coordinator thread to send next batch. This enables us to use minimum resources on the controllers

Field Detail

public static final int **maxDataRow_per_pkt**

Two columns of data (position, velocity) are packed into datagram. maxDataRow_per_pkt is the maximum n of the n x 2 matrix.

---

java.lang.String **address**

---

int **port**

---

byte[] **trajectory_data**

---

com.coordinator.network.UDP_Client **udp**

Constructor Detail

public **Trajectory_Server**(java.lang.String address,
int port,
int dataColumn)

Parameters:

ipAddress - String; IP Address of controller
port - int; port number
dataColumn - int; column position of data in trajectory file

Method Detail

public void **run**()

Thread run method. Sends datagram packet containing a portion of data in trajectory file to controller and waits for acknowledgement before sending next packet - until all data is sent or a stop command is issued

---

com.coordinator.coordination
Class JmDNS_Coordinator

public class **JmDNS_Coordinator**
extends com.coordinator.database.JmDNS_DATA

Implements methods for providing a PnP interface with controllers. It uses JmDNS protocol to register and discover services. JmDNS_Event_Coordinator listener method is called by its discover method to listen for events.

Constructor Detail

public **JmDNS_Coordinator**()

Initiates JmDNS

Method Detail

public java.lang.String[] **split**(java.lang.String data,
int n,
char c)

method takes a string, splits it where there are characters (c) and puts them in a String array of size n

Parameters:

data - String
n - int
c - char

Returns:

String[]

Throws:

IndexOutOfBoundsException -

---

public void **registerAService**(java.lang.String type,
java.lang.String name,
int port,
int weight,
int priority,
java.lang.String text)

Method registers one JmDNS service

Parameters:

type - String eg _supervisor._tcp.local.
name - String eg SUPERVISOR
port - int IP port number
weight - int
priority - int
text - String

---

public void **registerServices**()

method registers all services in JmDNS_DATA register array

---

public void **discoverServices**()

used to discover services listed in JmDNS_DATA discover array

---

public static void **writeFile**(java.io.File filename,
java.util.Vector data)

Utility method to implements ObjectOutputStream to write data to filename

Parameters:

filename - File

data - Vector

---

public static java.util.Vector **readFile**(java.io.File filename)

Utility program to read data from filename

Parameters:

filename - File

Returns:

Vector

---

com.coordinator.coordination
Class JmDNS_Event_Server

class **JmDNS_Event_Server**

extends com.coordinator.database.JmDNS_DATA

implements javax.jmdns.ServiceListener, javax.jmdns.ServiceTypeListener

Class for servicing events discovered by JmDNS on network. JmDNS_Coordinator calls this class to discover specified services. This class over writes serviceAdded, serviceRemoved and serviceResolved methods in javax.jmdns.ServiceListener and also serviceTypeAdded method in javax.jmdns.ServiceTypeListener.

Field Detail

private javax.jmdns.ServiceInfo **serviceInfoDevice**

---

static final com.coordinator.coordination.SynchFlag **SYNCH**

Method Detail

public void **serviceAdded**(javax.jmdns.ServiceEvent event)

Method used by JmDNS ServiceListener to listen for events; these incoming events are serviced; _mcast._udp.local.; _mcaststream._udp.local.; _device._pid; _http._tcp.local.; When connection information is received for each service, eg IP address for a controller's multicaster, the connection is launched. When _device._pid event is received, a Monitor thread is created with connection parameters and thread is executed to requests connection with controller's TCP server

Parameters:

event - javax.jmdns.ServiceEvent

See Also:

JmDNS_DATA, Monitor

---

public void **serviceRemoved**(javax.jmdns.ServiceEvent event)

This method services event removal from its host, ie users of services are informed; eg URL address is removed from GUI.

Parameters:

event – ServiceEvent

---

com.coordinator.coordination
Class Monitor

public class **Monitor**

extends com.coordinator.database.Data

implements java.lang.Runnable

The Monitor.class receives encoder positions from controllers and also trajectory end flags for synchronization via TCP client socket

See Also:

com.coordinator.network.TCP_Client

Constructor Detail

public **Monitor**(int device,
javax.jmdns.ServiceInfo deviceInfo,
com.coordinator.coordination.SynchFlag synchflag,
java.lang.String add,
int portNo)

Parameters:

device - int
deviceInfo - ServiceInfo
synchflag - SynchFlag
add - String; IP address
portNo - int; port

Method Detail

public void **run**()

Thread's run method receives encoder positions, synchronization flags, urgent and other info from controller

---

com.coordinator.coordination
Class SynchFlag

public class **SynchFlag**

This class stores synch flags received by the Monitor. In synchronized trajectory mode, each controller sends a flag to indicate end of trajectory and waits. When all flags have been received, i.e, synch==threads, synch is reset to 0 and MulticastSender is alerted by Monitor thread to multicast flag to alert them to run next trajectory.

Field Detail

public static int **synch**

stores synch flags received

---

public static int **threads**

stores number of thread Monitors, ie number of controllers

Method Detail

public synchronized void **put_flag**(int i)

increments synch by i, ie 1

Parameters:

i – int

public synchronized void **put_threads**(int i)

Each Monitor thread created calls this method to increment threads by i, ie 1

Parameters:

i - int

---

public synchronized int **get_flag**()

Returns:

int; synch

---

public synchronized void **reset_flag**()

sets synch to 0

---

public synchronized int **get_threads**()

Returns:

int; number of threads.

C3 com.coordinator.interpolation

Class Transmission_Flag

public class **Transmission_Flag**

InterpolationData_Trans uses this class to get and set a boolean flag in this class

Field Detail

static boolean **flag**

Method Detail

public synchronized void **getFlag**()

sets flag. If flag is false, method calling this (UDP.transaction) will be put in wait state until flag is true.

---

public synchronized void **setFlag**(boolean sflag)

sets flag and notifies UDP waiting on lock

Parameters:

sflag - boolean; flag

---

com.coordinator.interpolation

Class Interpolator

public class **Interpolator**

extends com.coordinator.interpolation.Transmission

implements java.lang.Runnable

NC code interpretation and interpolation is done in this class. The code was originally received from NRC-IMTI in C format and was designed for DSP's. Rodney converted it to Java and added protocols for transmitting data over network

Constructor Detail

public **Interpolator**(int numAxes,
java.lang.String platformType,
int period,
boolean ACDEC,
java.io.File file,
java.lang.String transactionType,
com.coordinator.network.McastFlag flag)

Parameters:

numAxes - int; number of axis
platformType - String; JStick or local PC host
period - int;interpolation period
ACDEC - boolean; if true use acceleration and deceleration profiling in interpolation
file - File; parsed Gcode file
transactionType - String; viaRT,UDP,Multicast, ie communication mode

flag - McastFlag; use for interrupting interpolation

Method Detail

public void **run**()

Thread's run method reads NC data, interpolates, calls inverse kinematics if needed and calls methods to send data to controllers either directly or through real-time JStick coordinator

---

void **readCoord_Circular**()

run() calls this method to read circular interpolation data

---

void **readCoord_linear**()

run() calls this method to read linear interpolation data

---

private int **InterpolationCoordinator**(int axes)

run() calls this method to coordinate interpolation, ie call axisInit1 followed by velocityInit1(), axisInit2(axes) and velocityInit2(). Number of interpolation steps are returned and used to calculate setpoints. After ach computation setpoints are sent on the network

---

void **tripodInvKin**()

calls tripod's inverse kinematics with translation and orientation derived from interpolation

---

void **axisOutput**()

method for passing setpoints to communication protocol

---

com.coordinator.interpolation
Class Transmission

public class **Transmission**
extends com.coordinator.database.Data

Protocol for sending setpoints from interpolator to JStick coordinator

Constructor Detail

public **Transmission**(java.lang.String type)

creates UDP socket

Parameters:

type - String; trajectory type, eg "via RT JStick Host"

Method Detail

public synchronized void **transaction**()

creates and sends datagram packets containing setpoints. To avoid flooding the JStick coordinator, the size is limited to 576 bytes. The method works together with Transmission_ACK and Transmission_Flag to wait for response from the JStick coordinator before next UDP is sent. Method is interrupted by user stop command

---

com.coordinator.interpolation
Class Transmission_ACK

public class **Transmission_ACK**
extends com.coordinator.interpolation.Transmission

implements java.lang.Runnable

UDP class calls this method to receive flag from JStick coordinator that it is ready for next batch of setpoints.

Method Detail

public void **run**()

UDPExt thread's run method waits for datagram from coordinator; it uses UDP.class socket connection to receive, blocking until data arrives

---

C4 com.coordinator.GUI

Class TrajTable

public class **TrajTable**

extends javax.swing.JFrame

This class shows a table for inputting trajectory data (position velocity) for all axes

Constructor Detail

public **TrajTable**()

sets table (columns and rows) with headers

Method Detail

private void **jbInit**()

sets graphical display parameters

Throws:

Exception -

---

com.coordinator.GUI

Class MainApplication

public class **MainApplication**

This is class is the application program for starting the Graphical User Interfaces

Field Detail

public static com.coordinator.GUI.MainGUIFrame **frame**

---

private static com.coordinator.network.McastFlag **mcastflag**

Constructor Detail

public **MainApplication**()

Sets graphical parameters for MainGUIFrame

Method Detail

public static void **main**(java.lang.String[] args)

Starts Application and com.coordinator.coordination.JmDNS_Coordinator;

Parameters:

args - String[]

---

com.coordinator.GUI

Class MainGUIFrame

public class **MainGUIFrame**

extends javax.swing.JFrame

Constructor Detail

public **MainGUIFrame**(com.coordinator.network.McastFlag flag)

Parameters:

flag - McastFlag

Method Detail

private void **jbInit**()

Called by constructor to display graphics

Throws:

Exception -

---

protected void **processWindowEvent**(java.awt.event.WindowEvent e)

Overridden so we can exit when window is closed

Parameters:

e - WindowEvent

---

void **axis_1Go_actionPerformed**(java.awt.event.ActionEvent e)

axis 1 Go button

Parameters:

e - ActionEvent

---

void **axis_2Go_actionPerformed**(java.awt.event.ActionEvent e)

axis 2 Go button

---

void **axis_3Go_actionPerformed**(java.awt.event.ActionEvent e)

axis 3 Go button

---

void **axis_4_Go_actionPerformed**(java.awt.event.ActionEvent e)

axis 4 Go button

---

void **axis_5_Go_actionPerformed**(java.awt.event.ActionEvent e)

axis 5 Go button

---

void **axis_6_Go_actionPerformed**(java.awt.event.ActionEvent e)

axis 6 Go button

---

void **go_actionPerformed**(java.awt.event.ActionEvent e)

Go button for synchronized moves

---

void **setPoints**()

method to pass (position, velocity, acceleration in textfield boxes to ControllerIO

---

void **runProfile_actionPerformed**(java.awt.event.ActionEvent e)

method below is for running different trajectory profile modes

---

void **stop_actionPerformed**(java.awt.event.ActionEvent e)

Method to call multicaster to stop controllers

Parameters:

e - ActionEvent

---

void **reference_actionPerformed**(java.awt.event.ActionEvent e)

When 'reference position' button is clicked method alerts ControllerIO

---

void **Load_actionPerformed**(java.awt.event.ActionEvent e)
When 'Load trajectory' button is clicked method calls TrajDataApplication with file data

Parameters:
e - ActionEvent

---

void **driverson_actionPerformed**(java.awt.event.ActionEvent e)
When 'drives_on' button is clicked method alerts ControllerIO

---

void **driversoff_actionPerformed**(java.awt.event.ActionEvent e)
When 'drives_off' button is clicked method alerts ControllerIO

---

void **PIDFilter_actionPerformed**(java.awt.event.ActionEvent e)
method to execute PIDTable to show PID Filter table

---

void **dreference_actionPerformed**(java.awt.event.ActionEvent e)
When 'define_home' button is clicked method alerts ControllerIO

---

public void **setWebButton**(java.lang.String device,
java.lang.String flag,
java.lang.String name)
Method for setting and enabling web buttons

Parameters:
device - String; device
flag - String; on button is enable; off it is disabled
name - String

---

void **webButton1_actionPerformed**(java.awt.event.ActionEvent e)
opens web browser for axis 1

---

void **webButton2_actionPerformed**(java.awt.event.ActionEvent e)
opens web browser for axis 2

---

void **webButton3_actionPerformed**(java.awt.event.ActionEvent e)
opens web browser for axis 3

---

void **webButton4_actionPerformed**(java.awt.event.ActionEvent e)
opens web browser for axis 4

---

void **webButton5_actionPerformed**(java.awt.event.ActionEvent e)
opens web browser for axis 5

---

void **webButton6_actionPerformed**(java.awt.event.ActionEvent e)
opens web browser for axis 6

---

void **New_actionPerformed**(java.awt.event.ActionEvent e)
method executes TrajTable for displaying and editing trajectory data

---

void **shutdown_actionPerformed**(java.awt.event.ActionEvent e)
method alerts controllerIO to shutdown controllers

---

public static void **setPowerButtons**(java.lang.String device,

java.lang.String flag)
this method shows status of controllers (on or off)

---

void **axis_1_power_actionPerformed**(java.awt.event.ActionEvent e)
alerts controllerIO to shut down axis_1 controller

---

void **axis_2_power_actionPerformed**(java.awt.event.ActionEvent e)
alerts controllerIO to shut down axis_2 controller

---

void **axis_3_power_actionPerformed**(java.awt.event.ActionEvent e)
alerts controllerIO to shut down axis_3 controller

---

void **axis_4_power_actionPerformed**(java.awt.event.ActionEvent e)
alerts controllerIO to shut down axis_4 controller

---

void **axis_5_power_actionPerformed**(java.awt.event.ActionEvent e)
alerts controllerIO to shut down axis_5 controller

---

void **axis_6_power_actionPerformed**(java.awt.event.ActionEvent e)
alerts controllerIO to shut down axis_6 controller

---

void **speedReady_actionPerformed**(java.awt.event.ActionEvent e)
handles speed over-ride GUI controls and alerts controllerIO

---

void **jCheckBox1_itemStateChanged**(java.awt.event.ItemEvent e)
all controllers may be shut down by this logic if a fault occurs on one of them

---

void **logButton_actionPerformed**(java.awt.event.ActionEvent e)
this method is for logging encoder positions of individual controllers

---

public void **updateProgressBar**()
progress bar for encoder logger

---

public void **zero_actionPerformed**(java.awt.event.ActionEvent e)
alerts controllerIO when 'go to zero position' button is clicked

Parameters:
e - ActionEvent

---

public void **Home_actionPerformed**(java.awt.event.ActionEvent e)
alerts controllerIO when 'home' button is clicked

Parameters:
e - ActionEvent

---

com.coordinator.GUI
Class PIDTable

public class **PIDTable**
extends javax.swing.JFrame

This class implements the GUI for setting, editing and saving PID values

Constructor Detail

public **PIDTable**()

Method Detail

private void **jbInit**()

Called by Constructor to display graphics

Throws:

Exception -

---

com.coordinator.GUI
Class TrajDataApplication

public class **TrajDataApplication**

Field Detail

boolean **packFrame**

Application to execute TrajDataFrame.class GUI

Constructor Detail

public **TrajDataApplication**(java.io.File fileName)

Called by MainGUIFrame with file as argument

Parameters:

fileName - File; file with trajectory data

---

com.coordinator.GUI
Class TrajDataFrame

public class **TrajDataFrame**
extends javax.swing.JFrame

This class displays a GUI for choosing trajectory type, platform, communication type and modifying trajectory data

Constructor Detail

public **TrajDataFrame**(java.io.File file)

Constructs the frame

Method Detail

private void **jbInit**()

Component initialization called by constructor

---

public void **jComboBox1ListStart**(java.lang.String firstItem)

method to populate interpolation data type list when GUI shows up

---

public void **jComboBox1List**(java.lang.String firstItem)

method to populate interpolation data type list when GUI resets

---

public void **jComboBox2List**(java.lang.String firstItem)

method to populate interpolation platform list

---

public void **jComboBox3List**(java.lang.String firstItem)

populate number of axis list

---

public void **jComboBox4List**(java.lang.String firstItem)

populate motion controller mode list

---

public void **jComboBox5List**(java.lang.String firstItem)
populate motion communication mode list

---

public void **jComboBox5List2**(java.lang.String firstItem)
Populate motion communication mode list. This is used for interpolation modes other than G Code {"TCP Direct", "UDP Direct"}

---

public int[] **fileArrayDim**()
measures the row and column sizes; used for trajectory types other than G Code

---

public void **fileTypeTest**()
tests for the type of data file

---

boolean **saveFile**()
method to save current file configuration and trajectory data files

---

boolean **saveAsFile**()
Save current file, asking user for new destination name.

---

protected void **processWindowEvent**(java.awt.event.WindowEvent e)
Overridden so we can exit when window is closed

---

void **trajTypeCombo_actionPerformed**(java.awt.event.ActionEvent e)
Method to handle trajectory type selected

Parameters:
e - ActionEvent

---

void **platformCombo_actionPerformed**(java.awt.event.ActionEvent e)
HANDLES INTERPOLATION PLATFORM selected

Parameters:
e - ActionEvent

---

void **AxesCombo_actionPerformed**(java.awt.event.ActionEvent e)
Handles AXES (e.g. 2, 4) selected

---

void **contModeCombo_actionPerformed**(java.awt.event.ActionEvent e)
handles CONTROLLER MODE selected

---

void **comModeCombo_actionPerformed**(java.awt.event.ActionEvent e)
handles COMMUNICATION MODE selected; e.g. "via RT JStick Host", "Multicast Direct"

Parameters:
e - ActionEvent

---

void **OK_actionPerformed**(java.awt.event.ActionEvent e)
handles OK button as ff; set acceleration, period and abs/relative strings in Traj_Configuration_Data handles selection of non-break point trajectory type handles condition for break point trajectory type and alerts controllerIO handles condition for "nonbreakpoint (synchronized)" trajectory type and alerts controllerIO handles condition for G Code; interpolation on real time JStick and alerts controllerIO handles condition, local interpolation, setpoints transmission, which may be option 1 (viaRT) transmits setpoints to a real time module

which in turn distributes to controllers option 2 (UDPdirect) transmits directly to controllers by UDP option 3 (McastDirect) transmits to controllers by multicast and alerts controllerIO

Parameters:

e - ActionEvent

C5 com.coordinator.network

Class UDP_Client

public class **UDP_Client**

Class for creating and sending datagrams via a datagram client socket

Field Detail

java.lang.String **address**

---

int **port**

---

byte[] **data**

---

java.net.DatagramPacket **outpacket**

---

java.net.DatagramPacket **inpacket**

---

java.net.DatagramSocket **ds**

---

java.net.InetAddress **inetaddr**

Constructor Detail

public **UDP_Client**(java.lang.String address,
int port,
byte[] data)

prepares a DatagramPacket dp and DatagramSocket ds

Parameters:

address - String
port - int
data - byte[]

Method Detail

public void **send**()

sends Datagrampacket out_packet by DatagramSocket ds

---

public void **send**(byte[] outdata)

sends outdata in Datagrampacket out_packet by DatagramSocket ds

Parameters:

outdata -

---

public void **receive**(byte[] in_buffer)

receives datagram into in_buffer

Parameters:

in_buffer -

---

com.coordinator.network
Class DatagramSender

public class **DatagramSender**
extends com.coordinator.database.Data

Transmits control flags and data to JStick servers

Method Detail

```
public static void trajectory(double pos,
                              double vel,
                              double accl,
                              java.lang.String dgramipaddr,
                              int dgramport,
                              int mode)
```

transmits datagram to controllers for jogging axes

Parameters:

pos - double; position
vel - double; velocity
accl - double; acceleration
dgramipaddr - String; IP address
dgramport - int; port number
mode - int; informs controllers of this mode, i.e. jogging

```
public static void sendFilter(java.lang.String dgramipaddr,
                              int dgramport,
                              int index,
                              int[][] PID,
                              int flag)
```

Sends PID filter parameters to controllers

Parameters:

dgramipaddr - String; IP address of controller
dgramport - int; datagram port
index - index for device
PID - Proportional, Integral, Derivative
flag - shows data type, i.e. PID data

```
public static void control(java.lang.String dgramipaddr,
                           int dgramport,
                           int mode,
                           int value)
```

Sends datagram containing speed over-ride

Parameters:

dgramipaddr - String; ip address
dgramport - int; ip port
mode - int; flags motion controllers
value - int; speed over-ride value

```
public static void control(java.lang.String dgramipaddr,
                           int dgramport,
                           int mode)
```

Sends various command flags to motion controllers;

Parameters:

dgramipaddr - String; IP address
dgramport - int; IP port number
mode - shows data type e.g. 6 => turn on motor drive;

---

public static void **control**(java.lang.String dgramipaddr,
int dgramport,
int mode,
double accl,
boolean isAbsolute,
boolean isAbsoluteVel,
int pktSize,
int logSize,
int logGran)

This method is used in break-pt, non break pt and synchronized trajectory modes to set the states of controllers for this mode

Parameters:

dgramipaddr - String; IP Address
dgramport - int; Port number
mode - int; flag to set controller state
accl - double; acceleration
isAbsolute - boolean; true if positions are absolute
isAbsoluteVel - boolean; true if velocities are absolute
pktSize - int; used to set buffer size for incoming packets
logSize - int; used to set number of position values to log
logGran - int; used to set log granularity, ie frequency of log

---

public static void **control**(java.lang.String dgramipaddr,
int dgramport,
int mode,
double accel,
int axisNum,
int logSize)

For G-code interpolation mode:

Parameters:

dgramipaddr - String; IP Address
dgramport - int; Port number
mode - int; flag to set controller state
accel - double; acceleration
axisNum - int; axis ID
logSize - int; size of position data to log

---

public static void **control**(java.lang.String dgramipaddr,
int dgramport,
int mode,
int numaxes,
double interpTime,
double acceln)

sends data to JStick interpolator

Parameters:

dgramipaddr - String; IP Address
dgramport - int; Port number
mode - indicates the type of data
numaxes - int; number of axes
interpTime - double; period of interpolation
acceln - double; acceleration

---

com.coordinator.network

Class McastDirect

public class **McastDirect**

This class is used by com.coordinator.interpolation.Interpolator to multicast setpoints directly to controllers

Constructor Detail

public **McastDirect**(java.lang.String ipaddr,
int port)

Parameters:

ipaddr - String; IP address of controller
port - int; port

Method Detail

public static void **multicastDirect**(int[] data)

Method for multicasting setpoints to controllers

Parameters:

data - int[]; setpoints

---

com.coordinator.network

Class McastFlag

public class **McastFlag**

contains methods for flagging controllers when a user issues a stop command

Field Detail

public static boolean **flag**

Method Detail

public synchronized void **put_runStatusFlag**(boolean pflag)

sets run pflag

Parameters:

pflag - boolean

---

public synchronized boolean **get_runStatusFlag**()

Returns:

flag boolean

---

com.coordinator.network

Class MulticastSender

public class **MulticastSender**
extends com.coordinator.database.Data

Class for sending state signals by multicast to JStick servers

Method Detail

public static void **run**(byte[] flag)

Parameters:

flag - signal to be sent to controllers

---

com.coordinator.network

Class TCP_Client

public class **TCP_Client**

This class creates a TCP client socket and has methods to create DataInput and DataOutput streams on this socket

Field Detail

java.io.DataInputStream **data_in**

---

java.io.DataOutputStream **data_out**

---

java.net.Socket **client_socket**

---

int **timeout**

---

java.lang.String **address**

---

int **port**

Constructor Detail

public **TCP_Client**(java.lang.String address,
int port,
int timeout)

Creates a client socket and waits forever until server is found

Parameters:

address - IP Address
port -
timeout - Connection timeout

Method Detail

public java.io.DataInputStream **dataInStream**()

Creates an input TCP stream for the client socket

Returns:

DataInputStream

---

public java.io.DataOutputStream **dataOutStream**()

Creates an output TCP stream for the client socket

Returns:

DataOutputStream

---

public void **close_Stream**()