A computing architecture for a multiple robot controller

Anthony Topper
B. Eng.

Department of Electrical Engineering
McGill University, Montréal

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree
Master of Engineering

June 1991

Abstract

This thesis describes the architecture of the *Kali* multi-arm robot control system, including discussions on design trade-offs and performance, as well as a detailed implementation using off-the-shelf hardware and software. *Kali's* main objectives are flexibility, integration, and hardware and software modularity so as to facilitate programming, experimentation and portability. It is used primarily to prototype concepts in multi-arm coordination, teleoperation, force control, and sensor fusion etc. To implement *Kali* requires only a minimal real-time kernel, a sufficiently powerful computer and a robot interface. It is based on the principle that, today, computational hardware, real-time operating systems and robot interfaces no longer impede robot controller design. Rather it is control software and system integration which are now the main problems. To that end, this work also discusses in depth the fundamental problems in the design and engineering of robot controllers from an implementational point of view using *Kali* as a primary design example.

Résumé

On décrit dans cette thèse l'architecture de *Kali*, un système de commande de robots multi-bras. On discute aussi des différentes options de sa conception, de la performance, ainsi que de la mise en oeuvre à l'aide de composants matériels et logiciels disponibles dans le commerce Les objectifs du système *Kali* sont la flexibilité, l'intégration et la modularité matérielle et logicielle pour faciliter la programmation, l'expérimentation et la portabilité. On l'utilise principalement pour construire des systèmes prototypes de coordination multi-bras, de téléopération, de commande a retour d'efforts et de fusion multi-sensorielle. La mise en oeuvre de *Kali* se fait a l'aide d'un noyau temps-réel de fonctionalité minimale, d'un système informatique suffisament puissant et d'un interface manipulateur. Ce développement est basé sur le principe que de nos jours, le matériel de calcul, les systèmes temps-réels et les interfaces manipulateurs ne sont plus ce qui retarde la conception des systèmes de commande de manipulateurs Le problème est maintenant celui du développement du logiciel et de l'intégration du système. Dans cette optique, ce travail examine en détail les problèmes fondamentaux de la conception des controleurs de manipulateurs du point de vue de la mise en oeuvre en utilisant *Kali* comme exemple principal.

Acknowledgements

First and foremost I thank project leader Professor Vincent Hayward without whose vision and guidance *Kali* never would have become a reality. Secondly, I thank Professor Laeeque Daneshmend for his invaluable contributions, numerous suggestions and for the design of the *Kali* servo code. I also thank Ajit Nilakantan who along with Professor Hayward created the *Kali* trajectory generator software. I extend my gratitude to Dr. Samad Hayati of the Jet Propulsion Laboratory for pushing through red tape and funding the project. I also confer my appreciation to John Lloyd for his thoughts on robot software design, to Mike Parker for his help in UNIX networking intricacies and to Cem Eskenazi for honoring many special requests. Finally, I would like to thank my family for their encouragement and unfailing support.

Contents

List of Tables

List of Figures

1. Introduction

Castles in the air—they're so easy to take refuge in. So easy to build, too.
—Henrik Ibsen (1828-1906)

1.1 The Robot Controller Problem

Robotics is by its very nature an experimental science, yet rarely has the literature discussed one of the major practical difficulties in robotics, namely, the process of actually building a robot control system—the robot controller problem It encompasses the process of engineering a solution to a particular application including all its practical and idiomatic aspects such as computing hardware, software, actuator interfaces, etc. It is not in itself any theory or rationale of the application at hand; nor is it about any particular design philosophy; it is, however, about communication mechanisms, servo rates, processor speed, kernel calls, and programming tools, etc. The former are prerequisites to the problem, the latter are the problem.

In the past, the major obstacles to building robot controllers were the poor floating point performance of commercial processors and the meager capabilities and often unbearable development environments of commercial real-time kernels. These limitations drove researchers to waste most of their effort just to build an adequate computing platform with sufficient software tools upon which to test their theories about robotics. In addition, interfacing a digital control system directly to a robot's motors and sensor feedback was a non-trivial task, and the lack of ready-to-use hardware interfaces made it a difficult undertaking for all but experienced robotics engineers. Thus, robot controllers often depended on unique hardware and software architectures and operated only with specific robots and host computers. Such systems were often too cumbersome to use and certainly too difficult, if not impossible, for others to adopt for their own research Despite assertions of their powerful capabilities, most research controllers performed even simple tasks less well than the primitive commercial systems they were designed to surpass.

Today, however, thanks to vendor independent open architectures, and the advent of powerful yet inexpensive microprocessors with integrated real-time programming environments, it is now easy to engineer an impressive, yet easy-to-use, robot controller in a matter of weeks using off-the-shelf hardware and software. All that is required: a sufficiently powerful computer, a minimal real-time kernel, a robot interface, and robot control software. The first three can be purchased simply from commercial sources, leaving the last to be obtained either from other researchers or created as needed. McGill University's *Kali*[1] robot control system is an example of this new trend. It is a software architecture for the control of multiple coordinated manipulators founded on the principle that the robot controller problem is no longer hindered by inadequate floating-point hardware, real-time operating systems, or robot interfaces. Rather it is control software and system integration which are now the main problems—precisely those areas of interest to researchers.

1.2 Software is Paramount

In seeking a plausible system architecture, the robot controller designer must, in effect, have a good knowledge of the empirical and idiomatic aspects and the various computational, programming, and I/O demands of a basic robot controller including: computational hardware fast enough to perform trajectory generation and joint level servoing, user programming interface and development environment, and a real-time kernel to tie the system together. Once these are laid down, the problem becomes one of software architecture to meet all the specifications while 'mapping' the entire system onto an appropriate hardware implementation [Ostroff87]. In practice, this is far easier said than done, not only because of the myriad problems in designing a complex real-time system, but also because determining specifications 'a priori' during the design phase is a difficult task, since much information about the environment is not known until run-time [Ramamritham88]. Thus rapid prototyping, both in building a functional skeleton of the end-result and in simulating algorithms, plays an important role in the creation of any modern robot controller.

[1] In Hindu mythology, Kali the Divine Mother, is often represented as a creature with many arms.

Without adequate ease in creating and altering code, or a fast enough edit-compile-test cycle, development bogs down and the flexibility necessary for research and experimentation quickly decays. In fact, it is the programming environment and the development tools which determine the usefulness of a robot control system, in the end.

1.3 Thesis Overview

It is to assist the reader in this endeavor—the building of functional, ease-to-use robot controllers—that this thesis is dedicated. It examines the problems in the design and implementation of such systems while providing *Kali* as the principal design example including all the critical nuts and bolts information, otherwise so often forgotten:

- *Software structure.* The blueprint for a robot controller are the processes required to make it work, these include application code (task planning), trajectory generation, and servoing. A detailed examination of their organization, data structures, algorithms, and communication mechanisms is made.
- *Computational requirements.* What performance features does each process need from the hardware? How much computational power is needed execute a given algorithm at the required sampling rate? How many MIPS[1] or MFLOPS[2] are needed?
- *Computing architectures.* What architecture best suits the robot controller problem: vectored, pipelined, dataflow, or connection machine? Since a robot controller is built on a foundation of computing hardware, a fundamental understanding of it is necessary in order to select/design computational components for a robot control system. Perhaps nothing has stirred so much confusion as the advantages/disadvantages among the different hardware architectures: CISC, RISC, DSP, VLIW, systolic arrays,

[1] Million Floating Point Operations Per Second
[2] Million Instructions Per Second usually based on the benchmark that a Digital Equipment Corp VAX 11/780 is 1 MIPS

superscalar, superpipelining, etc. The costs/benefits of each as applied to robotics is explained.

- *The communication bottle neck.* Today, a robot controller is likely to be a tightly coupled multiprocessor system with complex interprocessor communication. The major architectural issue is whether to use either shared memory or message passing. What are the advantages/disadvantages of each? Which is most often used and Why? What mechanisms for interCPU messaging are necessary such as gang scheduled test-and-set, sleep-and-wake-up, or uncoordinated test-and-set? How are these implemented together with communication protocols and algorithms? In either case, synchronization of data exchange is always the crux of the problem. This, in turn, leads to the other major problem—bus traffic. As sampling rates and computation loads increase, it becomes the major bottleneck on performance. How does it increase and what can be done about it?
- *Commercial hardware.* No doubt this subject matter changes as fast as technology progresses, thus making any discussion of it quickly outdated, however, at some point decisions must be taken and real world hardware selected. For the reader's convenience an evaluation of current CPUs such as SPARC, MC68040, Mips[1] R3000, i860, etc., as well as computer buses such as VMEbus, EISAbus, FutureBus, etc. is also made. A discussion on other miscellaneous, though equally important, implementation issues like memory access time, power consumption, bus termination, and arbitration is also offered.
- *Real-time Software requirements.* A robot controller is very much a 'hard' real-time device for which scheduling deadlines such as servoing must be met unfailing. Thus what features are needed from a real-time kernel? How

[1] Corporate name for Mips Computer Inc , not to be confused with Million Instructions Per Second

long should it take to process an interrupt, a kernel call, or a interprocess message? Which commercial kernels provide these features and how well do they perform them?

- *Control Issues.* Fundamental to any robot control system are the trajectory generator rates and servo rates. How fast should these be generated to guarantee a smooth trajectory? Stable control of the manipulator? What are the upper and lower bounds? What are typical numbers? What about controller delay, quantization problems, and dynamic range of calculations?
- *Robot Interfaces.* What is the basic model for a robot joint? How are encoders used to determine position and velocity. How they are interfaced? How is a robot calibrated to absolute position? Some valuable tricks and techniques are explained as well as the complete design for a robot interface board is given. In addition, an examination of the advantages/disadvantages between brushed and brushless electric motors (the most common type of robot actuator), and between linear and switching amplifiers is made, although no comprehensive discussion of actuators is offered.
- *Safety and robustness.* Finally, how is safety and robustness (ability to withstand and recover from error or disaster) ensured in an experimental system? To prevent disaster it is essential to have a means of detecting faults, software failures and dangerous conditions.

1.4 Scope

In short, this work examines all the empirical and idiomatic aspects of robot controller design and implementation. However, it does not discuss the theory of robot control, trajectory generation, robot language design, or programming. For such grounding, the reader is directed to these works: [Paul72] one of the original works in digital robot control; [Paul81] the inveterate and founding handbook of modern robot control and trajectory generation ([Luh83] also provides a useful synopsis on the subject); [Craig89] an updated and more comprehensive version of Paul's

classic; [Craig88] and [Asada86] provide useful explanations and examples about advanced manipulator control; lastly, [Brooks75] and [Allworth87] contribute many principles, techniques, and wise counsel in software and real-time systems design.

In this thesis, the *Kali* robot control system is used as the principal design example which deals with the issues of multi-processor, multi-robot performance and control that represent a significant step in complexity and requirements over that for a single robot. Furthermore, this work deals with the diverse interrelated design issues concerning robot controllers and contains a wealth of information useful and necessary to the realization of any such systems.

Finally, it must also be pointed out that this thesis is not an attempt to proselytize the reader to some new, supposedly superior, scheme for a robot control system. Rather it is to demonstrate that the 'yet another robot controller' cliché is no longer relevant by showing the reader that any competent research team can quickly put together a controller tailored to the needs of a particular robotic system.

2 The Elements of Robot Controller Design

When we mean to build,
We first survey the plot, then draw the model;
And when we see the figure of the house,
Then we must rate the cost of the erection;
Which if we find outweighs ability,
What do we then but draw anew the model
—William Shakespeare (1564-1616)

2.1 The Robot Control Hierarchy

As with any engineering endeavor, the challenge is to decompose the problem into an as architecturally simple and economic solution as possible. Always it boils down to the design decisions and the engineering trade-offs. Of primary importance to robot controllers are the design choices imposed by that archetype of system design—the hierarchy of tasks. A scheme which takes maximum advantage of the natural data abstractions inherent in robotic control systems [Volz84]. Precise specifications for each layer or task in the hierarchy can be independently defined, thus simplifying development and removing, as much as possible, the burden of explicitly programming the details of the underlying mechanical system, *ie.*, manipulator kinematics, dynamics, motor parameters, etc. Changes in either hardware or software do not propagate throughout the system, but remain confined to the relevant levels in the hierarchy.

The fundamental system trade-off for robot controllers is between the computational complexity of their component tasks and the sampling rate of those tasks. That is, given the application: how complex are the strategies for planning, modeling, and control versus the sampling rate at which they are computed? The sampling frequency of control algorithms and, of course, the algorithms themselves define the performance of the system (aside from the inherent electro-mechanical limitations), but since the designer is always limited by the computational power available, he must trade-off between the two. For instance, a simple servo algorithm may provide fast position control at a high sampling rate, essential for pick-and-place applications, but a more

sophisticated algorithm using laser guiding sensors may provide better tracking control at slower sampling rate, necessary for seam welding. There is always the nagging possibility, though, that the poor performance of a sophisticated control scheme is due to a limitation in sampling frequency and not—as is sometimes the case with a well conceived algorithm, justified in simulation— failing miserably because of some fundamental, ill perceived flaw.

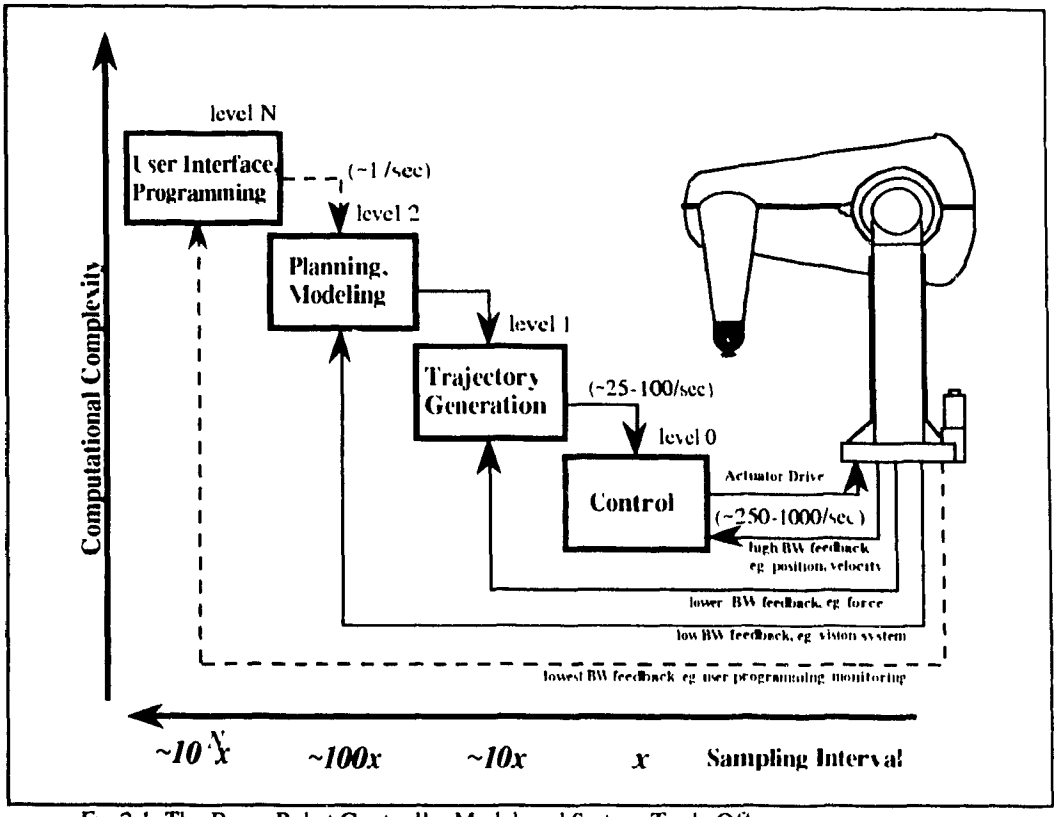


Fig 2.1 The Basic Robot Controller Model and System Trade-Off
A hierarchy of control tasks with increasing complexity and sampling interval as one advances up the hierarchy
level 0: actuator I/O and joint level control such as position or tracking control, force compliance, dynamics compensation, etc Some sensor I/O such as torque feedback for gripping, grinding, better tracking
level 1 temporal and geometric motion control Some simple multi-sensor input is also common, *ie.*, teleoperation with force feedback, peg-in-hole insertions, sensors to avoid collisions, etc Depending on the system design, integration between levels 0 and 1 varies considerably
level 2: world modeling, geometric reasoning, path planning, decisions based on symbolic sensor information, *ie.*, pick up object recognized by vision system
level N user interface (graphic/iconic protocols), programming environment, operating system, and off-line programming

Figure 2.1 about illustrates this basic trade-off in terms of the generic architecture for robot controllers: the user's application process is at the highest level and issues task commands (about 1 per sec) to a trajectory planning process which creates motion requests (more commonly

the user code does the trajectory planning itself), these motions requests (Cartesian space motions, joint space motions, force profiles, almost anything) are then, in turn, processed by a trajectory generator which translates these into a continuous stream of joint set-points quickly enough to ensure smooth motion (~25-100 per sec) while satisfying all the specified spatial, temporal, and dynamic constraints (*ie.*, move in a straight line from A to B at a speed of 1 m/s while not exceeding 10 m/s^2 acceleration); in the final stage, these points are sent to servo loops (these could be individual joint-based PID servos or a Cartesian space control loop with force feedback) which stabilize the manipulator at these set-points (~250-1000 per sec). All motions are typically permitted by the trajectory generator provided that they are within the manipulator's kinematic and dynamic capability When the trajectories are generated fast enough (usually ten times faster than the manipulator's natural structural resonant frequency [Paul81]), and the servos stabilize quickly enough (*ie.*, in one sample interval), so that tracking error dynamics are approximately linear over successive servo samples, then the generated set-points are perceived as continuous and smooth motions [Ahmad88]!

Despite the fact that manipulators are highly non-linear control plants, this scheme works because it exploits a successive small signal paradigm in which motions are seen as points along nominal trajectories in Cartesian velocity/force subspaces (level 1) and feed-forward dynamics are used to linearize the control (level 0). In this way each layer in the hierarchy locally linearizes its functionality, so as to provide to its higher, 'parent', layer a linear system, in effect. Thus, coupling is highly reduced and the problem is divided into smaller and better defined pieces—the basic principle behind robot controllers. The prevailing assumption, however, is that the nonlinearities are not sufficiently severe to prevent effective local linearizations in the neighborhood of the operating point.

2.2 Anatomy of a Robot Controller

In general, robot control requires an implementation of each level in the hierarchy including hardware interfaces. However, the trajectory generation and servoing tasks are the minimum required for the most basic, often sufficient, function of a robot controller, namely,

position and/or tracking control along some specified nominal trajectory. These two tasks, in essence, form the heart of every robot control system.

2.2.1 Trajectory Generation

The user application issues motion/position requests to the system in an asynchronous manner using either a blocking (wait for move to be completed) or non-blocking (motion is queued, user code continues) mode. The trajectory generator then interpolates between the last motion/position to the newly desired one. Initially, it is necessary to get smooth motion of the individual joints or joint space control—known as 'joint mode'—because 1) joint trajectory generation is the easiest type of motion to get up and running, and 2) it is often necessary to move the arm into a given position free from singularities, especially for the debugging of other system software. Next, the task is to get smooth motion along arbitrary curves in space, known as 'Cartesian mode,' which simplifies programming, *ie.*, easy workspace geometry with jigs, etc.

Generally, trajectory generation is split into three sections: 1) the generation of the set points along a straight line segment where constant linear velocity is variable of control (the user specifying the time for the movement), 2) the merging or blending of motions between line segments to maintain smooth motion (*ie.*, continuity of velocity and possibly acceleration) when rapid changes of direction lead to increased forces on the joints (a fourth or higher order polynomial is usually fitted as the transition path between the line segments [Paul81, Hayward88(2), Lloyd91]), and 3) inverse kinematics which converts the Cartesian space line geometry into the joint space used by the servos. The final, blended Cartesian trajectory can be sent either directly to a Cartesian-based controller, or, more commonly, through inverse kinematics to joint mode controllers. The rate of trajectory generation depends on the structural resonance of the arm [Paul81]—too slow a rate and the arm may begin to shake (resonate). Accurate set-point generation and effective segment transition algorithms in response to sensor feedback are still much of a research topic [Paul85, Hayward84], especially for multi-robot control systems [Ahmad88(2), Hayward88, Hsu89].

Checks for singularities and/or configuration changes are critical at the trajectory generation level, lest the servos are given discontinuous joint demands (*eg.*, equally valid kinematic

solutions 180° apart). Avoiding the problem by picking the solution with the joints closest to the current position is not useful either, because near a singularity arithmetic round off error can cause unpredicted configuration 'flips'. It is standard practice, today, for robot controllers simply to leave it up to the programmer who decides on the configuration—the system simply 'advises' him, *ie.*, checks and issues warning of singularities or imminent configuration changes before the motion or does nothing and crashes during the motion. Many higher level path planning systems, though, have attempted to automate the process

2.2.2 Intertask Interface

A queue is used to interface the asynchronous motion requests issued by the user code to the trajectory generator. A linearly interpolated FIFO (First-In First-Out buffer, similar to a queue, except each 'end' is tied to a synchronous task of a different frequency) is used to interface the trajectory generation with the much faster servo control level. In this way large motion displacements are 'smoothed' out into smaller ones that the servo controller can better handle. If a position vector, **A** , represents the current position and, **B**, the desired new position, and r the ratio of the servo rate to the trajectory generator rate, then the incremental amount **(B-A)**/r can be added to **A** for $\lfloor r \rfloor$ interpolated cycles (*ie.*, at each cycle of the servo), so that $\mathbf{(B-A)}\lfloor r \rfloor/r$ is the position at the final cycle In the event $\lfloor r \rfloor < r$, the trajectory generator rate is not an integer multiple of the servo rate, the $\lfloor r \rfloor$-1st cycle adds the remaining tail-end amount. It could also be spread-out among more cycles to avoid a 'jolt' at the $\lfloor r \rfloor$-1st cycle.

2.2.3 Servo Control

Because of the decomposition into what amounts to an approximately continuous linear time invariant system, the large body of theory developed for linear systems is usually applied to robot controllers. For tracking control, a dual rate computed torque control strategy is generally employed: pole-placement via state feedback in the main loop with a forward loop compensator

intended to invert the highly non-linear dynamics of the feedback compensated plant [Craig89]. Expressed as an equation:

$$\tau = \mathbf{H}(\mathbf{q})\,\mathbf{u} + \mathbf{h}(\mathbf{q}, \dot{\mathbf{q}})$$

where τ is the vector of torque demands on the joint actuators; $\mathbf{H}$ the manipulator inertia matrix, $\mathbf{h}$, a vector function of joint position and velocity, representing all the dynamic forces acting on the manipulator, *ie.*, centrifugal, Coriolis, gravitational, joint friction, etc (these parameters easily vary as much as three orders of magnitude over different configurations, speed, acceleration [Bejczy74]), finally, $\mathbf{u}$ represents the vector of decoupled linear joint controllers, *ie.* a PID controller where k_{D_j} and k_{P_j} are the proportional and derivative feedback respectively (for the j^{th} entry)

$$u_j = \ddot{q}_{d_j} - k_{D_j}(\dot{q}_{d_j} - \dot{q}_j) - k_{P_j}(q_{d_j} - q_j)$$

with q_j and its time derivative representing the joint position and velocity respectively, and q_{d_j} with its time derivatives, the position demand, velocity demand, and acceleration demand respectively The 'core' loop, *ie.*, the computation of $\mathbf{u}$, must be calculated quickly enough to ensure stable control. A reasonable rule of thumb is that this should be sampled approximately 10 times higher than the joint's natural frequency [Paul81]. For most DC servo motors mechanically coupled to links of a representative mass, time constants are between 20 and 100 msec [Craig88] Thus, from the point of view of linear theory it should suffice to sample at about 100—500 Hz However, sampling rates as high as 5 kHz may be required for direct drive arms [Kanade84, Shalom88]. On the other hand, the parameters $\mathbf{H}$ and $\mathbf{h}$ depend on the configuration $\mathbf{q}$ and change much more slowly than the sampling rate for $\mathbf{u}$. Thus, if it is estimated that these functions vary with a significant amount, say for every 5° of joint displacement, and rate of change of the configuration of the robot is slow, say with a slew rate of less than 180° a sec (fast by todays' direct drive robot standards), then these 'inverse dynamics' need only be calculated at a mere 36 Hz (180°/sec – 5° = 36 Hz) [Kircanski86, Zhang88]! Figure 2.2 below shows the basic tasks of a robot controller and their how they interact, along with details on sampling rates.

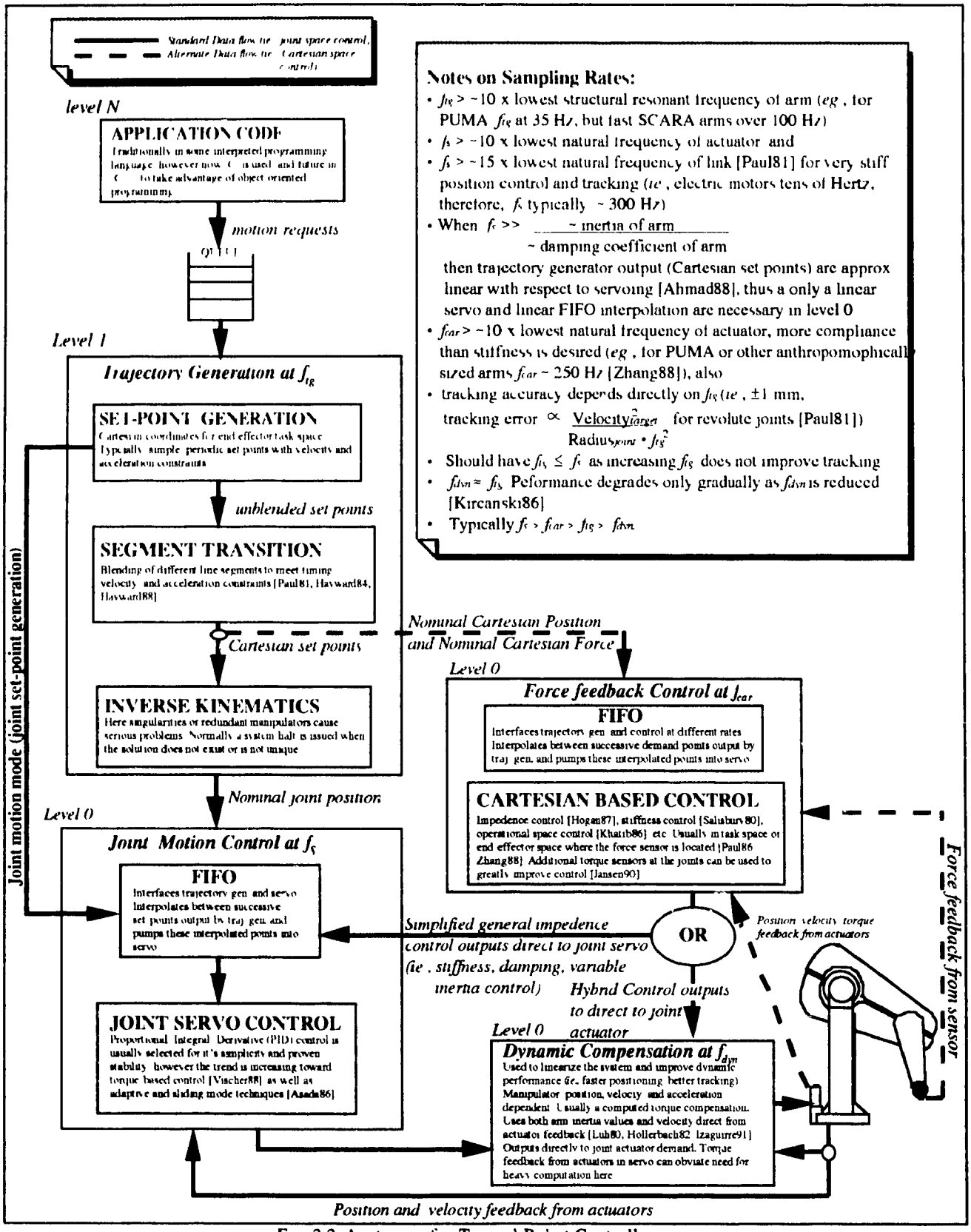


Fig. 2.2 Anatomy of a Typical Robot Controller

As it turns out, though, both the gravity and joint friction actually help to dampen out the system and reduce the control bandwidth, so that moderate position/tracking control can be achieved using just plain individual joint PIDs without any dynamics compensation—initially a godsend for slow computers in the past, now a curse, impeding higher performance and one of the compelling motivations behind 'cleaner' direct drive arms [Kanade84, Craig88, Kazerooni88] with simpler models.

In research robotics, however, the typical application involves not only position/tracking control, but also force control in Cartesian space, *ie.*, insert peg in hole by 'feel', or more often a combination of the two. A force sensor at the wrist and/or in the joints provides the appropriate feedback. Other sensor modalities like tactile and vision are being employed in an integrated fashion, so as to achieve greater control than would be possible with any one sensor alone—known as multi-sensor fusion [Hackett90]. However, these algorithms are of a different nature than those used for traditional robotics motion/force control in that they lend themselves more easily to parallel decomposition (*ie.*, image processing algorithms) The discussion here, however, concentrates on the traditional modalities for robot control.

2.2.4 Computational Requirements

In general, the computational cost for position/tracking control—by far the most computationally intensive task in the robot control hierarchy—for a six degree of freedom arm is about 50 kFLOPS using ordinary PID servos at a 1 kHz sampling rate More sophisticated algorithms, such as adaptive PID, require around 2000 kFLOPS @ 1 kHz sampling Assuming a worst case scenario using, say a 6 DOF direct drive arm requiring a 5 kHz sampling frequency, and using a computed torque method with inverse kinematics in the core loop (about 1500 FLOPS per control cycle), then only 7.5 MFLOPS of computational power are needed—well within the reach of many current microprocessors (see section 2.3.5).

For generalized hybrid force/position control, on the other hand, the requirements are considerable greater. Typically about 750 kFLOPS @ 250 Hz sampling [Zhang88] or about 3 MFLOPS per kHz are needed (15 MFLOPS for the 5 kHz, direct drive manipulator example).

The traditional preoccupation among researchers, however, has been with the inverse dynamic computation—thought to be the critical component in achieving better tracking performance From the theoretical perspective, the computational complexity has been reduced over the years from about $O(n^2)$ (10^5 FLOPS for n= 6 DOF) using Lagrangian formulation, to about $O(n)$ (10^3 FLOPS for n=6 DOF) [Luh80, Hollerbach82] using various Newton-Euler formulation It has even be reduced further using a completely empirical approach employing sensitivity analysis techniques to estimate, within a given percentage accuracy, the inverse dynamics, as one would of any function, given enough experimental data points [Izaguirre91]. On other hand, parallel algorithms executed on specialized hardware can achieve *O(log n)* or better [Lathrop85, Fijani91, McMillan91] (discussed further in sections 2.3.2 and 2.3.3). Table 2.1 summarizes the computational requirements for a typical set of robotics algorithms.

Table 2.1 Computational Requirements for Robot Control

Algorithm	FLOPS (per 6 DOF manipulator)
Tracking/Position Control @ 1 kHz (level 0):	
PID control	50 k
Adaptive PID control	2000 k
Force control @ 1 kHz (level 0):	
Stiffness control [Salisbury80]	1600 k
Impedance control [Hogan87]	2200 k
Hybrid control [Raibert81]	2200 k
Operational Space control [Khatib86]	3000 k
Feed-forward dynamic compensation @ 50 Hz (level 0):	
[Luh80]	80 k
[Hollerbach82]	60 k
[Izaguirre91]	40 k
(level 1):	
Trajectory generation & inverse kinematics (@ 50 Hz)	25 k

For each algorithm given (as applied to a six DOF robot), the approximate computation cost in floating point operations per second per Hertz of sampling frequency is shown That is to control a robot, such as a PUMA 560, at 250 Hz sampling frequency using a PID requires 12 5 kFLOPS = 50 kFLOPS x (250 Hz/1 kHz) The force control computational requirements are examined in detail in [Zhang88]

2.3 Computing Architecture

Once the designer decides on basic strategies, he or she must precisely determine the functions every level in the hierarchy should perform, being careful to ensure each executes efficiently enough to meet the associated real-time constraints Next, he or she establishes a computing architecture and then a suitable communication mechanism among the processes Depending on the resource sharing required by the tasks and the design of applicable parallel algorithms, he or she may choose a loosely coupled architecture based on networks, a tightly coupled one based around a common bus or on some variation between the two

The cardinal rule of digital control systems is to minimize delay, since it most profoundly effects stability and performance [Korein78, Franklin86] Great care must be taken to effectively reduce system bottlenecks without increasing latency. For instance, adding pipelining may improve system throughput, but it leads to larger delay times [Stone87]. There is nothing to be gained, for example, by using one processor to perform Cartesian trajectory generation while another processor, performing inverse kinematic transformations, waits for input from the first (a possible exception can be made, though, when an algorithm is implemented as an application specific integrated circuit, ASIC, where sheer speed of the pipeline results in a small overall delay [Lee86, Javaheri87]). Similarly, the effective computation time in a control cycle (a level 0 control loop) is always diminished by the I/O latency, because the task must wait for input feedback before beginning computation, only the time elapsed from the input of operands to the time when the results are output is actually free for computation.

Since it is generally recognized that certain computations within a sample period can be performed independently of the others, and by allocating them in different ways on a number of concurrently running processors or processing elements (performing select operations), the realm of parallel computing has come to be inexorably tied to robot controller design There are two major forms of parallel computing: coarse-grain and fine-grain Coarse grained parallelism, popularly referred to as parallel processing, refers to multiple processes running in cooperative fashion to perform a single program, examples of which are dataflow machines, coarse pipelining, and not

incidentally, multiprocessing systems defined by their communication paradigm like shared memory or message passing. In contrast fine-grain parallelism exists within a process at the the level of the individual operations (*ie.*, additions, subtractions, etc.), examples of which are vector machines, pipelines, systolic arrays, and host of CPU paradigms like CISC, RISC, VLIW, and DSP (terms explained later on). Fig 2.3 below shows basic advantages and disadvantages of typical parallel computing architectures used in robotics.

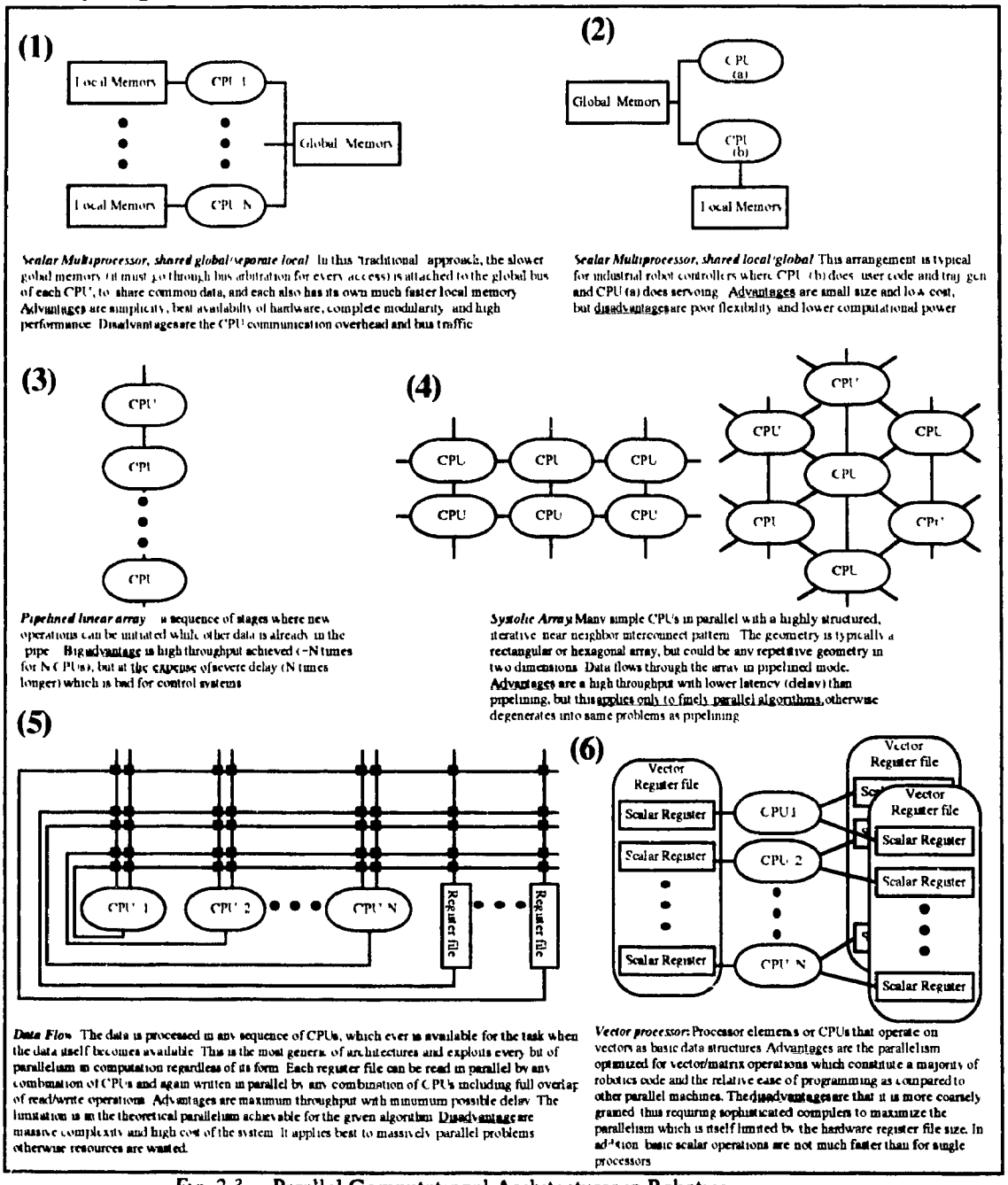


Fig. 2.3 Parallel Computational Architectures in Robotics

The two, coarse and fine grain parallelism, are, in fact, complementary and may be used concurrently. However, coarse grained parallelism is not user transparent, because it is still too difficult for compilers to translate generic sequential programs into multiple parallel processes Although it has been observed that a great deal of fine-grain parallelism can be achieved at the cost of increased hardware complexity, the use of large grain parallelism tends to favor overall system simplicity [Stone87].

At first glance, it would seem that the most appropriate solution to the robot controller problem is some complex concoction that exists in the lore of massively parallel computing—a data flow approach (where computations are performed as soon as data becomes available, *ie.*, computation 'flows' with the data), or maybe a pipeline, or systolic array, or maybe something else?

To help in making this determination, a popular metric, first coined by Amdahl [Amdahl67], is used as a comparative measure for different parallel architectures: defined (as one might expect) as the ratio between execution time using a single processor versus that using multiple processors—the 'speedup factor'. Amdahl pointed out that this measure, far from being linear, is, in fact, inherently limited by the amount of parallelism in the algorithm. This parallelism can be characterized by a parameter f, the fraction of computation that must be done serially. Note that this is not the granularity of a parallel algorithm, but rather a more fundamental measurement The granularity indicates the percentage communication overhead regardless of the parallelism in the algorithm. The two are often confused since highly parallel, small f, architectures are often finely grained and loosely parallel ones, big f, tend to be coarsely grained. The effective speedup, S, according to Amdahl becomes:

$$S = \frac{P}{fP + 1 - f}$$

when $f = 1$ all computation must done serially, so hardware parallelism is wasted and no speedup is possible (S=1). The same or worse holds true for the granularity (it is quite conceivable that a multiprocessing system requires so much communication, that it actually runs slower than the single processor case). On the other hand, when $f = 0$ then all computation is in parallel and performance increases more directly with the number of CPUs (S=P). Another important indicator is the efficiency, E, of a parallel architecture in matching an algorithm. It is defined as $E = S/P$ ($0 < E \leq 1$,

when E=1 then 100 percent parallelism). E is essentially limited by the inherent parallelism of the algorithm. This measure of efficiency, interestingly, leads to a fundamental conclusion about the practical parallelism possible in level 0 robotics algorithms.

2.3.1 Multiprocessing

The general hope lying behind distributed processing systems is that if one conventional processor isn't fast enough then maybe more will be. However communication bottlenecks and the inherent limits in the parallelism of robotics code restrict performance gains. Because real-time control tasks are as much I/O driven as computationally bound, communication schemes become a large part of the problem. Li and Malek [Li88] as well as Stone [Stone87] give detail analyses on various communication models. In the best case, assuming fully overlapping communications (rarely achieved), the system speedup is bounded by the communication overhead, for a uniform communications distribution, S_{max} = 1/(percent communication overhead or granularity). So a 10 percent overhead (a typical number) results in maximum speedup factor of less than ten, no matter how parallel the algorithm is (*ie.*, f=0) or how many CPUs are employed! (in general, there are an optimal number of CPUs resulting in maximum speedup for a given algorithm). Thus, as a rule, multiprocessor architectures should consist of inexpensive and simple processing elements with interprocess communication being as fast and efficient as possible, ideally easily reconfigurable and expandable with minimal deterioration of bandwidth. However, for practical purposes, a simple common bus architecture approach is often chosen (see appendix A).

For communication design there are basically two techniques: message passing and shared memory. Message passing makes the software easier to design and debug (provides for better data hiding more like object oriented systems [Schwan85, Bihari89, Clark89, Gentleman89]), but at a severe cost due to the excess overhead required for the message protocol. The performance bottleneck comprises not only the effective bus data rate (especially where the system is loosely coupled and shared memory is not used to hold messages) and memory contention (in more tightly coupled systems). Typical latency times on the order of milliseconds (see figure 2.4 below).

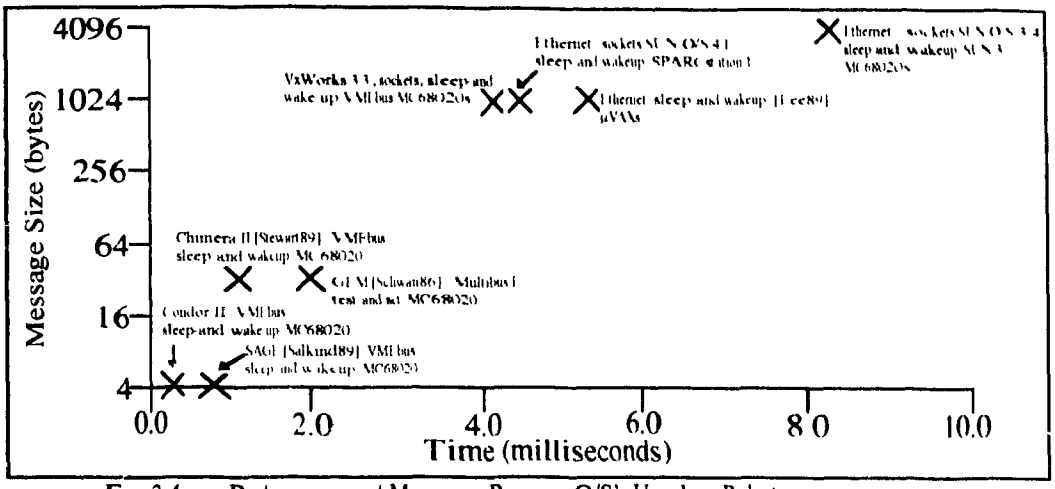


Fig. 2.4 Performance of Message Passing O/S's Used in Robotics
Based on multiprocessor intercommunication across a common bus or network

Shared memory, on the other hand, offers an approach nearing the maximum speed of the bus. However, only when processors are designated fixed priority access to the bus and contention is kept to a minimum. Most researchers prefer a shared memory architecture, because the robot control hierarchy defines tasks with large granularities, *ie.* highly decoupled and serial in nature. So memory accesses are largely confined to the local processor, thus making bus contention infrequent and predictable—the case where shared memory works best. A major disadvantage, however, is that such algorithms result in poor hardware utilization for highly parallel architectures (*ie.*, many CPUs with communication overhead, figure 2.5 below). [Leung88].

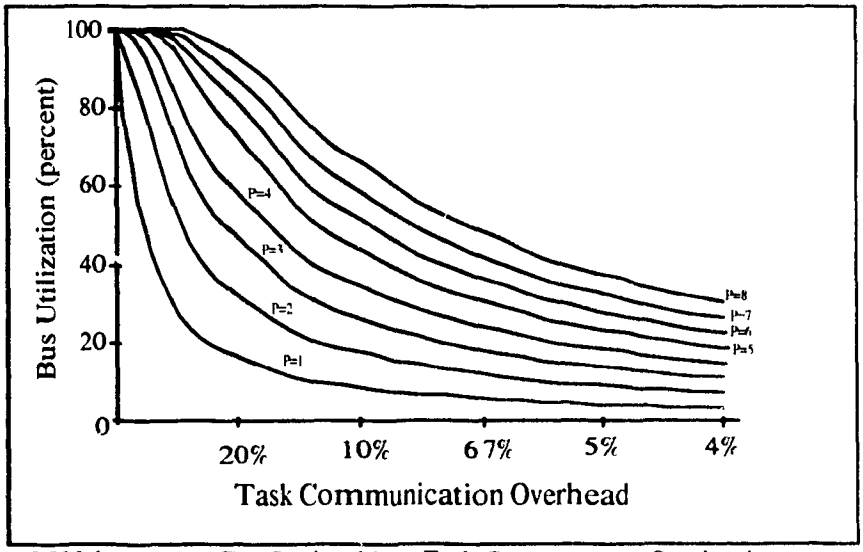


Fig. 2.5 Multiprocessor Bus Bandwidth vs Task Communication Overhead
Where P is the number of processors Based on a stochastic Petri Net model with a Markovian queueing theory (*ie.*, CPUs randomly accessing the bus and waiting if it is busy)[Marsan83, Han89] Break point is 20%

Bus contention depends on the two factors: a) bus hardware performance features such as arbitration and how fast memory is accessed (see Appendix A), and b) the amount of interaction between the tasks executing on the various processors which is basically synchronization problem: any shared data structure updated by more than one processor must be protected from conflicting concurrent updates by some sort of semaphore mechanism. The two common methods are the test-and-set spin-lock (on which many variations on exist such as delayed-retry, tournament scheduling, etc. [Dining89, Graunke90]) and the sleep-and-wakeup interrupt method (see figure 2.6 below)

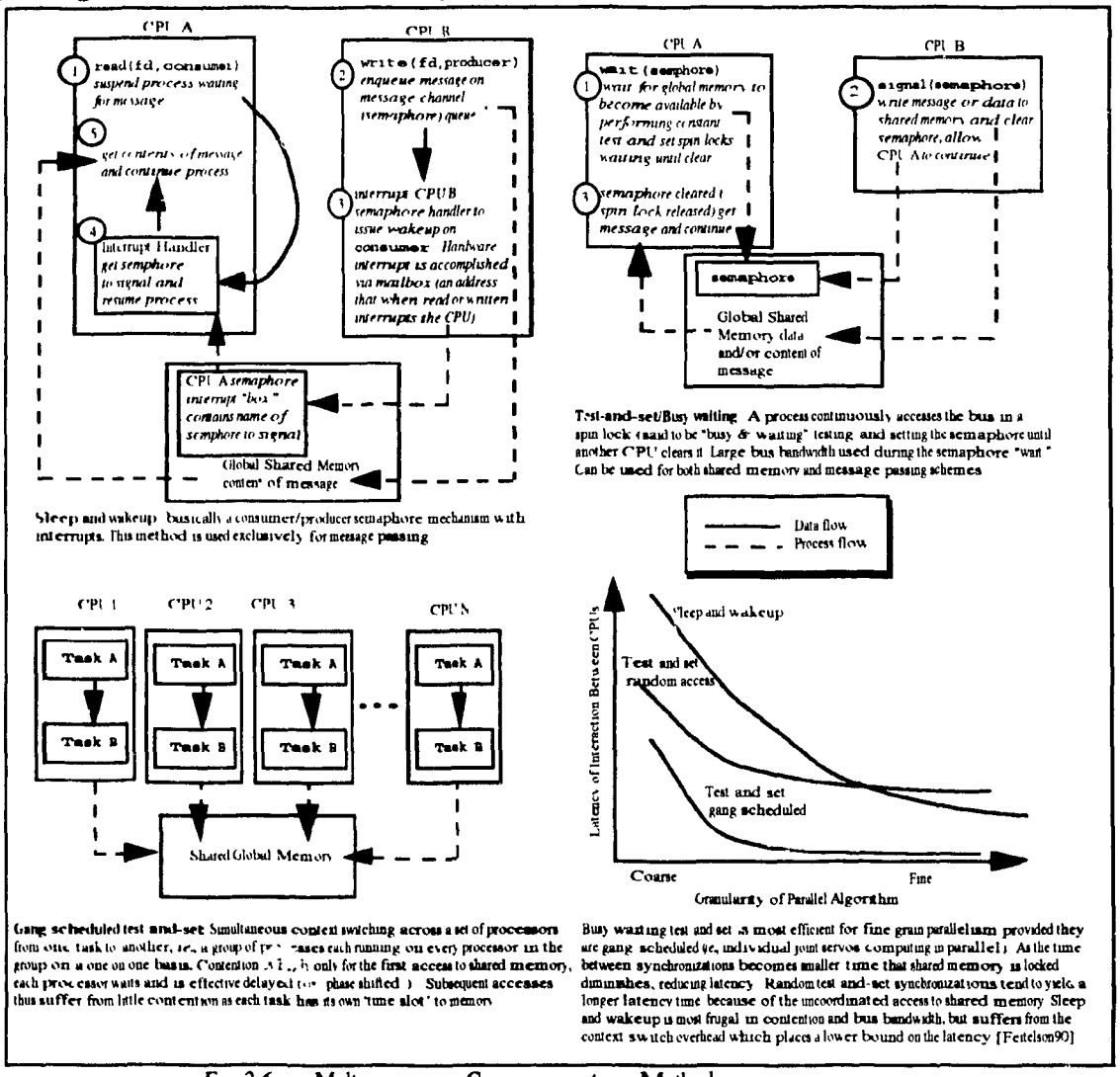


Fig. 2.6 Multiprocessor Communication Methods

Nevertheless partitioning the robot control problem for a multiprocessor is straightforward along the traditional coarse-grain lines in the hierarchy (*ie.*, trajectory generation in one processor and servo control in another) and a slew of systems have been designed strictly on

this paradigm (see section 2.7). Generally, the parallelism offered was the splitting of individual joint servo loops among different processors (S≈P, but S=1 for everything else including Cartesian based control). Another customarily adopted technique is to perform computations redundantly by all processors to achieve more partitioning for a given algorithm, sacrificing efficiency. On the other hand, for very coarsely grained cases, some researchers have even offered a network-based solution [Lee89]. More finely grained approaches targeted at specific algorithms (*ie.*, inverse kinematics), however, require substantial engineering. Nigam and Lee [Nigam85] proposed the use of commercial microprocessors interconnected to suit the particular algorithm with S=2.5. On the other hand, Kasahara and Narita utilized a special depth first/initial heuristic search scheduling algorithm [Kasahara85] on a multiprocessor system connected by a common bus to perform inverse dynamics. Levin [Levin87] employed a Transputer (general purpose CPU with serial links for communication, about 0.65 MFLOPS with floating point coprocessor) array for servo control with dynamics compensation. Other radical hardware approaches have been proposed all dealing with the intensive number crunching level 0 algorithms such as inverse dynamics, kinematics, Jacobians, and control.

2.3.2 Systolic Arrays and Pipelines

Most robotics algorithms can be conveniently represented as an array or vector, so it seems plausible that systolic array processing has the potential to offer great performance. However, these type of processors rely on a high degree of regularity, processing possibly hundreds of identical elements, to achieve high speed—a high degree of fine grain parallelism. The problem arises that as the number of sequential elements drops (*ie.*, what is called a 'stall' when the data configurations or instructions change randomly which force a reloading of the array), the overhead in beginning or filling the array can far exceed the actual calculation time. Pipelines suffer from the exactly the same problem, being the one dimensional case of array processors.

Nash and Przytula [Nash85, Przytula88] were the first to successfully develop a systolic array (16x16 processing elements) to perform linear matrix operations useful for most robotics algorithms including kinematics and inverse dynamics. Orin *et. al.* [Orin85, Orin86, Ling88] designed a pipelined multiprocessor (32-bit floating point ASIC implementation) system for

Jacobian computations and inverse dynamics (S<1.5). Lee and Chang [Lee86, Chang88] employed the bit-serial CORDIC algorithm [Volder59, Harber88] in a 25 stage pipeline to solve the inverse Jacobian very quickly (40 μsec inverse kinematic solution), but with relative inefficiency (S=1.38, E=0.06), hence the large pipeline needed to 'squeeze-out' the last bit of computation. Javaheri [Javaheri87] designed a floating point ASIC with 4 communication ports intended as processing elements for more efficient systolic array (1<S<2.2, E<2.2/P). Kircanski *et al.* [Kircanski89] used array processor approach using a multi-stage pipeline which matches the number of hardware multipliers and adders in the system against the mix of additions and multiplications in the instruction stream. Coupled with a novel symbolic approach to robot kinematics and dynamics algorithms [Kircanski88], the control loop (for 6 DOF) was claimed to be computed in 100 μsec. Efficient use of processing elements, though, drops off rapidly as the number of multipliers and adders is increased (1<S<2.2, E<0.7).

The basic problem is that all such architectures is that provide only increased bandwidth or throughput and not smaller latency. The time required for computation is always the maximum array depth (*ie.*, the number of processing stages required to complete the algorithm) multiplied by the processing stage operation time.

Trying to avoid this, Wang and Butner [Wang87, Butner88] created a heterogeneous hierarchy of processors for level 0 control each specialized for a given operation: bit slice processor for inverse kinematics & dynamics, CORDICs for trigonometric functions, and TI32020 DSPs for servoing. Results are impressive with 300 μsec for a complete control loop (computed torque servo with dynamics in the core loop). However interprocessor communication consumes over 16% of the processing time (S<2.3).

On the other hand, when the problem is confined only to inverse dynamics (no kinematics, trajectory generation, or code branching), remarkable speedups are possible. Fijani [Fijani91] showed that $O(n^3)$ inverse dynamic algorithms are optimal for parallel computation and that systolic arrays can be built to achieve S>5 over the best serial algorithms [Fijani91(2)], including communication overhead! Similarly, early on, Lathrop [Lathrop85] presented results using a Newton-Euler formulation with a logarithmic recursion implementable as a systolic pipeline

having $O(\log n)$ overall execution and S≈4. A solution is available at each clock cycle; however, there is a 400+ pipeline stage latency (6 DOF) for the results.

2.3.3 Vector Machines

Alternatively, vector machines make use of the fact most robotics algorithms can be expressed as 3-element vector operations (*ie.*, optimally decomposed 4x4 homogeneous matrices). It seems straightforward that a three fold increase in speed is possible. Ling [Ling88] proposed a vector processor of this type with a 20 cycle vector-matrix multiple time (3x1 by 3x3) a 6 MFLOPS rating with 125 nsec cycle time (2.5 μsec for the vector-matrix operation). However even commercially available, non-vectored DSPs (Digital Signal Processors, optimized for vector and matrix operations) [Dyer88] such as the MC96000 (Motorola) can perform the same operation in 24 cycles [Sohie88], not mention the vastly superior implementation technology available to a major semiconductor houses like Motorola (the same vector-matrix operation using a 75 nsec cycle time executes in 0.89 μsec on the MC96000). Interestingly, this is accomplished using only a single overlapping multiplier/adder unit.

Again for the specialized inverse dynamics problem, new approaches have yielded remarkable results. McMillan [McMillan91] used a Parallel Block Predictor-Corrector numerical method to solve for inverse dynamics, treating it as a differential equation problem. Employing a CRAY Y-MP supercomputer (one of the traditional vector machines) speedups are as high as 5.2 were achieved (# CPUs = 8, E<0.67).

2.3.4 Scalar Machines

Quite clearly robotics algorithms are not very parallel in nature. In fact, it can be roughly concluded, by empirical evidence alone (*ie.*, all the previous robotics architectures S<3, except for special sub-problems, *eg.*, inverse dynamics) that only 2 CPUs really are needed for maximum speedup of level 0 robotics code, anything more seems to be wasted. So $f \approx 0.5$ as figure 2.7 below illustrates.

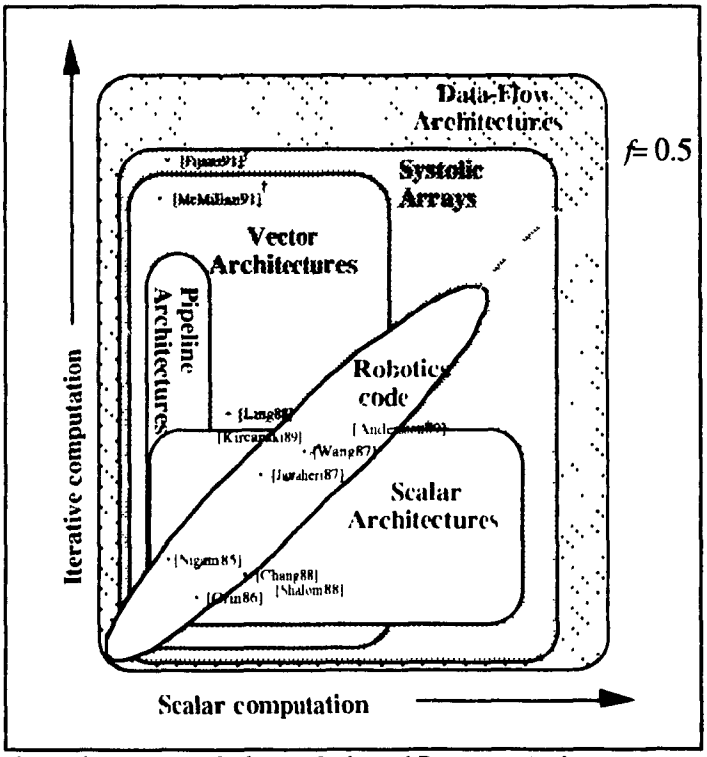


Fig. 2.7 Relationship Between Robotics Code and Processor Architectures
As previous system architectures show, robotics code is actually more scalar in nature (hence the limited performance pipelines and systolic arrays) It seams that VLIW machines with only two floating point units is sufficiently optimal for robotics [+]Inverse dynamics only

Realizing this Andersson [Andersson89] opted for the single 'big-iron' approach (a fast single scalar processor) , where one general purpose CPU (in this case with two floating point units using parallel buses to memory called 'JIFFE') performs all level 0 tasks. Efficiency is over 99%, and the processor was capable of performing the complete inverse dynamics and PID servoing in less than 50 μsec (about 20 MFLOPS, 40 MFLOPS on matrix addition and multiplication operations.

It appears inescapable that scalar processors constantly absorb any advantage claimed by parallel processing for traditional robotics force/motion applications (the same cannot be said, however, for other areas like image processing). Thus, it can be argued that at level 0 in the robot control hierarchy, parallel processing (as applied to a single manipulator) doesn't work. Robotics must be thought of as a coarsely grain operation along the traditional lines in the robot control hierarchy: the user code and trajectory generation together, since the user code, basically Cartesian motion statements, is closely coupled to motion requests which may vary considerably in

computation at run-time (*ie.*, error handling, collision avoidance, user code debugging, etc.), level 0 tasks, such as the servoing, can be placed in a separate CPU, since it is a critical process not to be burdened with unnecessary computation, or, worse, stopped because of a data error (*eg.*, a singularity). In general, the problem is to deal with the physical limitations of the object being controlled.

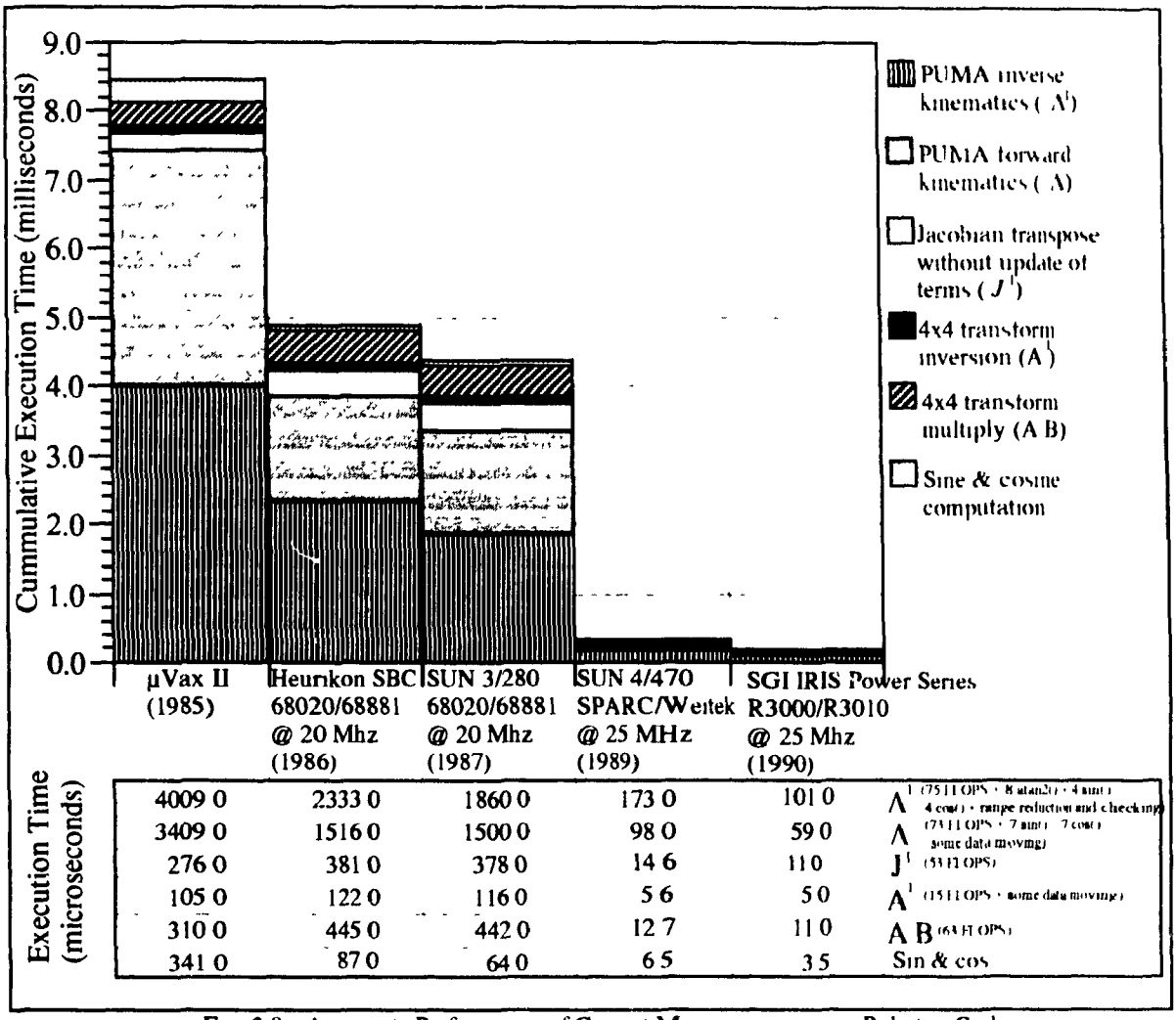


Fig. 2.8 Aggregate Performance of Current Microprocessors on Robotics Code
Using the RCCL speed test by John Lloyd. All arithmetic is performed using single precision floating point and the standard C library for trigonometric functions. A SPARCstation™ by SUN Microsystems is about 30 times faster than a µVAX and the Mips R3000 CPU with R3010 FPU (Floating Point Unit) in the Silicon Graphics IRIS workstation is up to 50 times faster than a µVAX in floating point performance on robotics code.

Interestingly enough, commercial microprocessors are taking a similar track to the one used by Andersson—one instruction stream with 2 floating point units (usually one adder and one

multiplier) with overlapping operation. The progress of microprocessor technology has improved so much that the numerous algorithmic improvements and specialized hardware created by researchers in the past is rapidly becoming obsolete in comparison. In addition, on the hardware side, the highest performing research systems are rapidly losing their performance edge over commercial processors. Consider figure 2.8 above showing some of the improvements in recent microprocessors (already considered 'old', the newer CPUs are faster still, like Advanced Micro Devices 29050 performing vector-matrix (3x1 by 3x3) multiply operations in 0.3 μsec and 4x4 matrix multiplies in 0.9 μsec approaching 40 MFLOPS for these operations [AMD90]). It is getting increasingly difficult to out perform the major semiconductor houses with discrete home-brew solution like JIFFE, whether in silicon, *ie.*, ASICs, or not.

2.3.5 Super Chips

The argument now is which commercial CPU is fastest? In order to make this decision, an understanding of minimizing the execution time is necessary: three factors contribute to this: N , the number of instructions that must be executed, C, the average number of processor cycles per instructions, and S, the number of seconds per processor cycle, so that execution time = N x C x S. Primarily these three depend on the compiler's optimization capability, the instruction set architecture, and the implementation technology. However the interrelationship between the factors is quite complicated, decreasing one factor may increase another by as much or more!

The traditional technique for general purpose processors is to decrease N at the expense of a smaller C. This approach, termed CISC (Complex Instruction Set Computing), attempts to better utilize the microparallelism present in horizontally microcoded machines by defining more complex instructions with more internal micro-parallelism in the hope that N would decrease more sharply than C. On the other hand, the RISC (Reduced Instruction Set Computing) approach employs the opposite philosophy: reduce C and S at the expense of N. In this case, hardwired instructions and heavy pipelining try to reduce C and S while powerful compiler optimization techniques keep N down [Gimarc87, Piepho89]. The result is a considerable overall reduction in the execution time. Current techniques to further improve performance involve increasing parallelism,

further decreasing C, by executing more than one instruction per cycle and using elaborate instruction sequencing techniques [Krick91]. Essentially, these are simplified dataflow designs made possible by 'look-ahead' techniques in the instruction stream, register scoreboarding (tagging those registers not being used by the current instruction as available for a concurrent instruction in another execution unit), and even more highly optimized compilers. One such new hardware technique is superpipelining where two or more pipelines are used in parallel an instruction is moved into each of execution pipelines every cycle. Essentially, it is a simple, fixed multiple dispatch architecture. Superscalar is another, where instructions are fed into more than one execution unit by a dispatch unit. It is more generalized than superpipelining, since dispatching is not on a fixed schedule, but depends more on the instruction stream ([Johnson91] provides an in-depth discussion on superscalar design). Still another, though simpler variation, on these themes, is also possible by embedding more than one instruction (at compile time) in each machine word. Known as VLIW (Very Long Instruction Word) processors, they seek to gain speed by increasing memory bandwidth to the CPU through shear word width, *eg.* 128 bits, so that many instructions can be fetched in one memory cycle (JIFFE, for example, uses a 200-bit instruction word length). Figure 2.9 below shows the comparison of single precision floating point capabilities of currently available microprocessors/computers (see Appendix A for detailed discussion on current and future commercial CPUs).

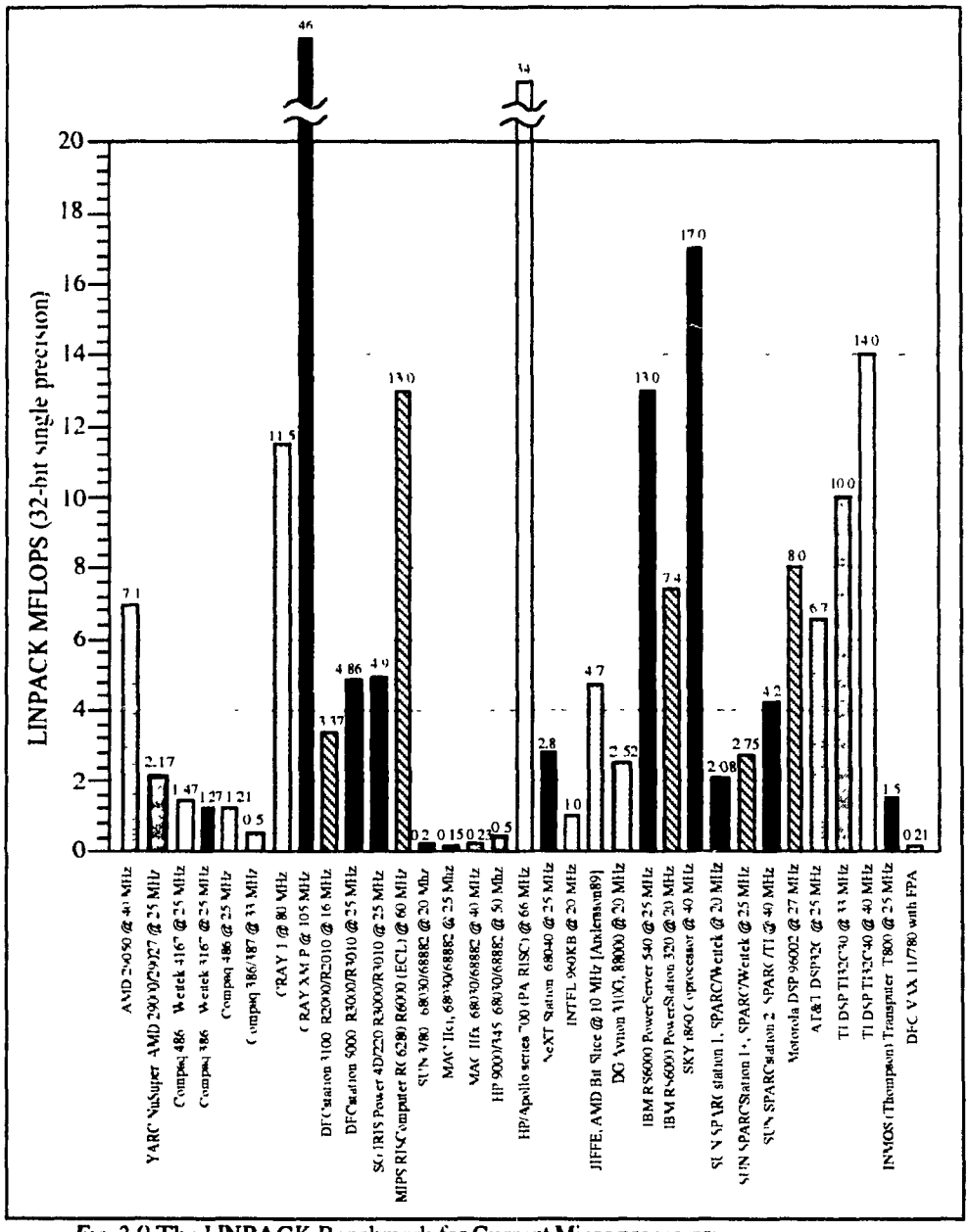


Fig. 2.9 The LINPACK Benchmark for Current Microprocessors

The LINPACK algorithms benchmark [Weicker90], developed at the Argon National Laboratory, measures aggregate floating point performance over a wide range or mathematical computations including matrix multiplication and trigonometric functions, etc It is normally used to characterize efficiency in vector processing, but since robotics contains much matrix/vector and trig functions, a high LINPACK rating is a reasonable, though by far not definitive, indication of the relative ability of that processor to execute robotics code

2.5 Real-Time Kernels

Traditionally, real-time kernels have been a great preoccupation of robot controller designers, most of whom have dwelt on creating 'better' real-time kernels with more features and facilties. However, these kernels, varying from the simple [Chen86] to the complex [Stewart89], typically haven't achieved any better performance than other real-time kernels on the same hardware. This is because the real problem is that robot controller performance is not generally limited by the real-time kernel, but by the robot control software itself, or the lack of it, its integration with the real-time kernel and the application at hand [Gopinath89]

As a rule, in robotics applications the essential performance factors for real-time kernels are: a fast context switch time and low interrupt latency, to provide for the fastest possible response to critical events and a minimized overhead for periodic functions which comprise bulk of robot controller computational requirements; an interprocess (or in the case of multi-CPU systems) interprocessor communication and synchronization mechanism (*eg.*, usually shared memory with semaphores), to provide for effective coordination among the required tasks, and debugging & development tools, though considered ancillary features, to provide the necessary ease in engineering a successful system. See Appendix B for an in-depth comparison of current commercial real-time kernels.

2.6 Robot Joints

A robot is essentially a collection of individually controlled joints each of which can be seen as a complete sub-system on its own (except in the case of Cartesian mode control in which all the joints are collectively controlled). Whereas many different types of actuation and feedback mechanisms exist and are being developed, the gear reduced DC electric motor with optical encoder position feedback is still the actuation/feedback most commonly employed today (though high torque output direct drive electric motors are being increasingly adopted [Kanade84, Kazerooni88, Shalom88]). Indeed, robot joints are becoming progressively more light weight, compact, easily integrated, efficient, and nearly maintenance free. [Dote90] provides a comprehensive examination

of both the practical and theoretical aspects in implementing a single joint or motor control systems. The typical joint control and feedback is shown in figure 2.10 below.

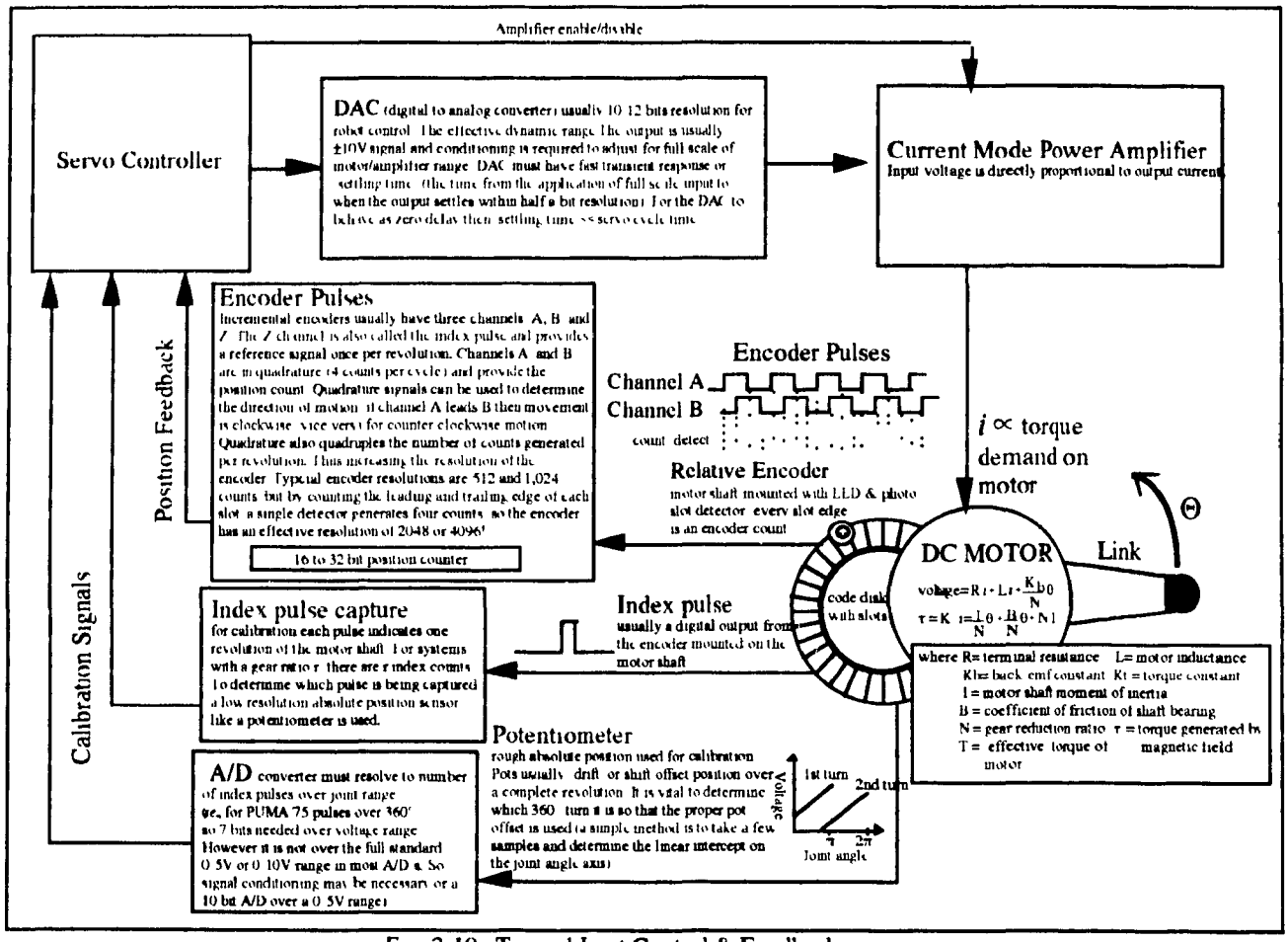


Fig. 2.10 Typical Joint Control & Feedback

2.6.1 Sampling Issues

There is no doubt that advances in technology will obviate—indeed, to a large degree, they already have obviated—the need for any consideration of the effects of discrete time controllers. The most obvious of which are:

- *Sampling frequency.* The approximation that the sampling rate of a discrete time algorithm is continuous which is usually acceptable if the controller sampling rate is sufficiently higher, *ie.*, 10 times higher [Franklin86], than the natural frequency of the system being controlled. Otherwise explicit

consideration of the discrete time effects must be taken in consideration (*eg.*, if a servo algorithm requiring one 'tick' is $O(n)$, then for two 'ticks' it is $O(n^2)$ instead).

- *Controller delay.* There is always a finite delay between sampling feedback variables and outputing resultant control signals. There is also a measurable delay in the power amplifier and in the current to the motor, and thus in torque at the joint. Such unmodeled delays should at be at least three times less than the control cycle time [Asada86].
- *Quantization noise.* Because the 'real world' is analog and feedback control signals are discretized through digital to analog and analog to digital converters, there is a quantization error in their measurement. This inherently limits the accuracy of any digital control system (*ie.*, a joint position servo usually gets to within ±1 encoder counts of demanded position) and can lead to drastic results in adaptive systems that accumulate round off errors.
- *Dynamic range and word size.* Both Simulation and practical experience have shown that at least 16 bits of angular resolution are required for adequate motion control (typical high precision pick-and-place robots use 20 bits of accuracy [Seiko90]). In the past, attempts to minimize computational cost using integer arithmetic are now supplanted by the era of floating point calculations—no more bit rolling and twiddling to keep significant digits on integer only machines! The IEEE 32-bit floating point format [Coonan80] has a 20 bit mantissa giving one per million accuracy or 1.2 arc seconds (just enough for current high precision robots, double precision may be needed in the future otherwise a retreat to 32-bit integer arithmetic, yielding an accuracy of 3×10^{-4} arc seconds, may be necessary).

2.6.2 Motors & Amplifiers

In theory, most DC motors produce torque in direct proportion to their armature current, which means that one needs to servo the armature current to a command voltage using a power amplifier controlled through a D ↓C (Digital to Analog Converter) in the digital domain. However, while accelerating the load, the motor is acting like a generator, and producing a 'back EMF' which is directly proportional to the motor (and load) speed. Therefore, in order to achieve a constant armature current, it is necessary to apply progressively more voltage to overcome the back EMF, which is often the limiting factor in the loop. One way to overcome the low-speed effect of back EMF is to put the armature in the feedback path of a DC power amplifier, or equivalently establish a 'current loop' servo (*ie.*, as a current mode voltage source). In fact, most sophisticated manipulator control schemes do adopt this technique, since generated motor torque is proportional to armature current, $\tau = K_t I_{armature}$.*ie.*, a computed torque servo control formulation).

A more efficient way to deal with the 'back EMF' is to use a switching amplifier such as a PWM (Pulse Width Modulation) amplifier which is designed to apply relatively high voltages to the motor for brief, variable time slices. In this case, it is required that the PWM amplifier is, at least roughly, matched to the motor's inductance so that power losses are minimized (*ie.*, motor must meet a minimum inductance for smooth wave form). This is expressed as the form factor for the amplifier and defined as the ratio of the RMS current to the average current in the motor ($k = I_{RMS}/I_{avg}$). It is dependent on amplifier switching frequency, the electrical time constant of the motor (*ie.*, L/R, see figure 2.10) and any other stray inductances ('ballast'). The power losses are proportional to the square of both the form factor and the torque generated (*ie.*, Power loss $\propto k^2\tau^2$). Thus once a motor has been selected and the torque requirements fixed through selection of gearing (if necessary), losses can be minimized by making k to as close to 1.0 as possible.

In general, power amplifiers are of two basic types: linear or switching. While linear amplifiers have excellent stability and control characteristics (*ie.*, no switching, so the form factor is 1.0), due to their linearity, they suffer from from heat generation in the output stage, because they operate in the high dissipation region of the transistor characteristic. Switching amplifiers overcome

this problem by controlling their output stage so that they are alternating between the fully on and off (*ie.*, transistor saturation or off) value of the output voltage. By modulating the duty cycle of the output, switching amplifiers generate a given voltage or current, depending on the variable of control. Due to the high current square wave frequencies necessary a high electromagnetic radiation usually accompanies switching mode power amplifiers. However a few new designs, have used very high switching frequencies together with small output filters that largely eliminate the radiation, but leave a wide performance bandwidth [Copley88]. Furthermore, unbeknownst to most researchers, switching mode amplifiers have typical closed loop current-mode bandwidths (not sampling frequency) of about 1-3 kHz. This is not a problem for slower geared manipulators with low sampling needs (*eg.*, less than 1 kHz), but for very high performance direct drive arms, which may need over 10 kHz sampling frequencies, this could become a problem as the control algorithm may respond to the current switching itself. Generally, the switching frequency must be high enough to ensure that the motor control system does not respond to the transistor switching itself (*ie.*, $f_{switching} > 10\, f_{bandwidth\ of\ control\ loop}$). However, hysteresis and eddy current losses in the motor increase with frequency, thus limiting the maximum switching frequency usable with a particular motor. On the other hand, the amplifier must be designed so that the $1/f_{switching}$ is large enough relative to the transistor switching delay time to ensure linearity, yet small enough so that power losses in the transistors (which increase in proportion to $f_{switching}$) are not excessive. Motor/amplifier selection is not as trivial as it first seems!

As far as motors are concerned, heat is the principal killer of performance; however, so long as it can be adequately removed, performance can be prevented from diminishing significantly (the same is true of power transistors and consequently power amplifiers). Electric motors can, in general, take peak currents many times higher than their continuous RMS values for short durations, *ie.*, the time the motor can, in effect, 'sink' the excess heat generated. [Fleischer88] provides a useful summary on motor characteristics and selection. See figure 2.11 below.

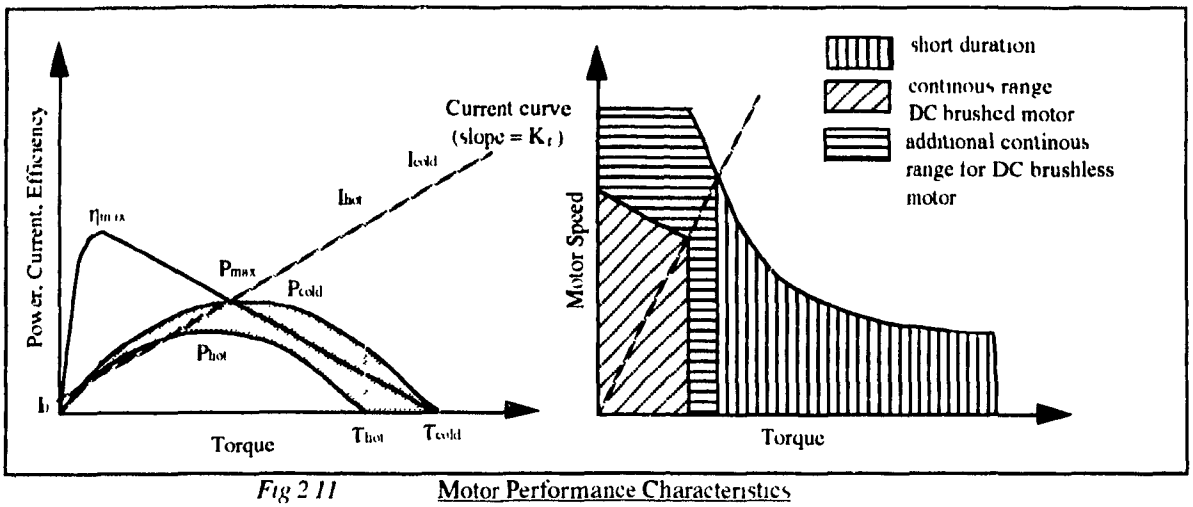


Fig 2.11 Motor Performance Characteristics

Performance 'band' (*ie.*, power, current, and efficiency) for electric motors is better when cold and diminishes considerably when hot (by as much as 50%) Brushless motors, because of superior heat dissipation and non-mechanical commutation, have a much larger operating envelope than their mechanically commutated, brushed counterparts

2.7 System Implementation Approaches

Since any system is a product of past experience and current influence, it is beneficial to examine the many possible overall system approaches to robot controller problem, each implementing to various degrees different levels in the hierarchy. Figure 2.12 below illustrates. A workstation host with graphical user interface is usually employed to perform application level programing and modeling/planning (A). This includes everything from basic robot motions, with on-line path modification in Cartesian end-effector space integrated with sensor I/O, to task level programming systems. The user is presented a C language library interface or increasingly a C++ [Stroustrup87] library interface to which user code is linked. Applications are built by creating, combining, and manipulating functions or objects defined from system primitives.

In high level motion planning (level 3) systems like 'Handey' [Lozano-Pérez88], the approach is to concentrate the code solely on higher level code and employ a commercial controller for all lower level control intricacies. It is convenience to do so, since point to point motions is all that is required. On the other hand, lower level, more control oriented systems like RCCL [Hayward86, Lloyd88] actually perform trajectory generation on the host workstation in real-time

(UNIX kernel modification needed) and use the commercial controller only to perform position servoing and other associated I/O (A, B).

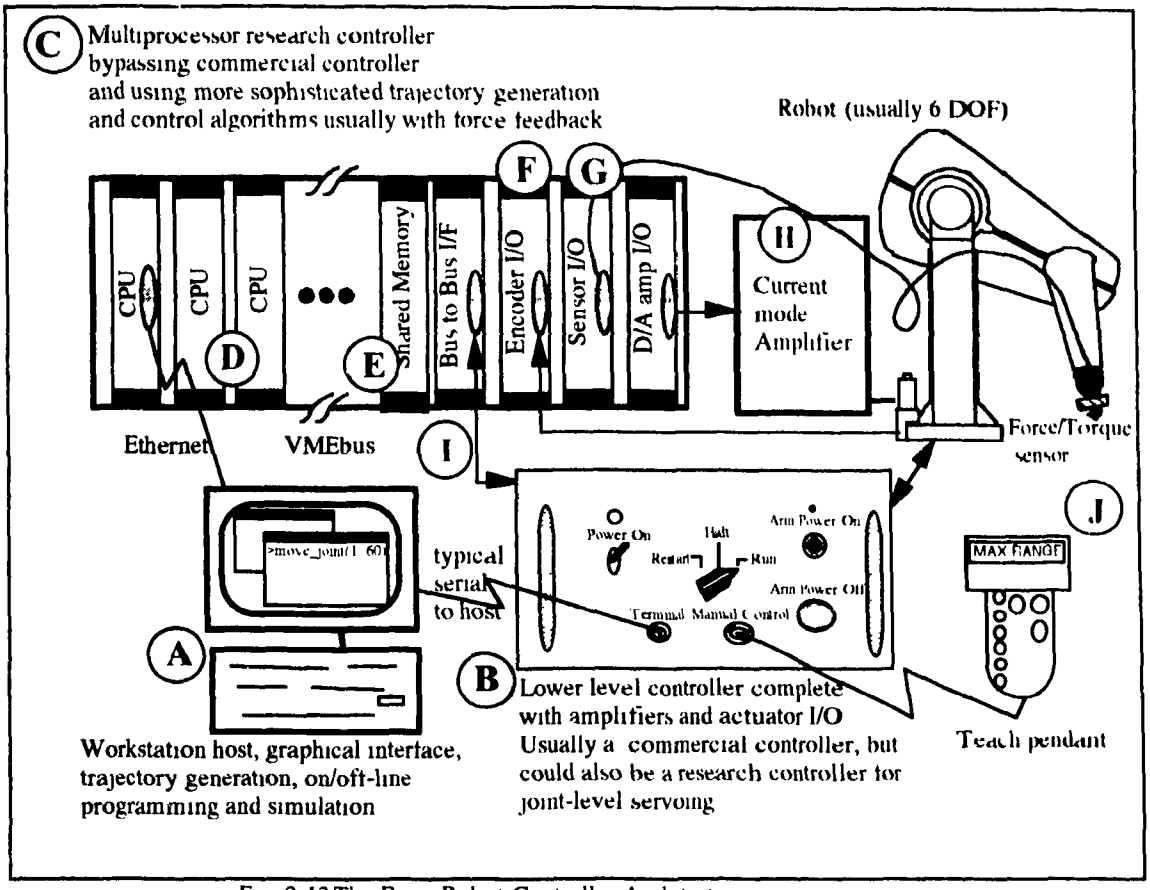


Fig. 2.12 The Basic Robot Controller Architectures
Implementations of the robot controller model vary from complete ground-up systems (A, C, D, E, F, G, H) to high level only versions (A, B) using existing commercial controllers, and all the variations in between Such as (A) programming host, (C,D,E) traj gen , and (B) servoing with actuator drive Or incorporating the commercial controller for robot I/O, but not for sensors (A, B, C, D, E, G, I) It is also usually the case that some form of teach pendent (J) or handcontroller (whether home-brew or commercial) is part of such systems

The most common approach, however, is to employ an external real-time multiprocessor system (C, G) to perform various intermediate level functions (usually level 1 tasks) such as sensor based control (*eg.*, vision, tactile, etc.) and/or various control hierarchies. Examples of which are CONDOR II [Narasimhan88] (for Utah/MIT hand), CHIMERA II system [Schmitz89, Stewart89] (multisensor fusion, a separate joint-based PID servo hardware provides level 0 robot control [Kanade84]), and *Kali* [Hayward88, Backes89] (multi-robot control and coordination). Yet others build upon previous research controllers, such as HIC (hierarchies of servo loops) [Clark89]

implemented on a CONDOR II platform which provides better scheduling and asynchronous event handling than its host system.

Because of the many complexities involved in real-time control, most systems have tended to place a heavy emphasis on their own real-time multiprocessor operating systems which consist largely of scheduling strategies and interCPU communication mechanisms through shared memory on a common backplane (D, E). The majority of these systems consist of everything from robot interface and motor amplifiers to real-time operating systems and application programming (A, F, G, H). Examples of these include COSMOS/NYMPH (NS32016 based, shared memory) [Chen86], GEM/CHAOS (i80x86 based, message passing) [Schwan85, Schwan86], CONDOR II (MC680x0 based, shared memory), SAGE (MC680x0 based, message passing) [Salkind89], HIC (MC680x0 based, shared memory), CHIMERA II (MC680x0 based with specialized coprocessors, both shared memory and message passing), etc. However, it is only recently, in systems like RIPL (MC680x0 based, shared memory, VxWorks kernel, A—E, G, I) [Miller90], and *Kali* (MC680x0 based, shared memory, VxWorks kernel, A, C—H) [Hayward88, Backes89], that commercial kernels together with off-the-shelf hardware are being employed together so as to reduce the development time as much as possible and concentrate on the robot control. Thus, de-emphasizing the previous trend in which robot controllers were largely exercises in hardware and real-time operating system design.

Going still deeper, to the lowest layers, one delves into the realm of computational hardware design. As previously described, many researchers, even in the recent past, have built their own specialized processors to speedup calculations. At first these were designed as highly parallel machines implemented as ASICs (Application specific Integrated Circuits) [Orin85, Orin86], to perform faster matrix calculations [Nash85, Nigam85, Orin85], but later more general purpose processors were designed using off-the-shelf bit slice parts, such as the RIPS system [Wang87, Butner88] (A, C &D with custom bus, E, F, H), or JIFFE [Andersson89] (SUN backplane as host for real-time system A, custom D, E, F, G). It is more likely that ASICs or other specialized hardware will not be used as main processors as some have argued [Leung88(2)], but rather will be

used as specialized I/O chips or coprocessors as some commercial products have already demonstrated [Olsen89].

2.7.1 Robot Controller Roundup

Table 2.2 below presents this diverse collection of research robot control systems over the past ten years covering the full range of applications, architectures, and design approaches to the robot controller problem.

Table 2.2 Summary of Research Robot Controllers

Author(s)	Intended Application	Type	Design Level	Programming	Architecture: Function Units	Architecture: Communication	Architecture: Control	Performance features
[Unimation80]	Industrial robotics (PUMA controller)	Multi proc	system & robot soft	interpreter, VAL language, cartesian moves	6 6503 joint processors, 1 LSI 11 master	Q bus, shared memory	LSI 11 user prog, 6503 fixed control store	control 6 DOF at 1kHz sampling 36Hz traj gen
[Klein82]	Walking machine (ASV, GEM)	Multi proc	system	joystick operator console	5 LSI 11 processors	dedicated parallel links message passing	distributed control program	control 18 joints at 10ms servo
[Golwasser84]	Hand controller torque feedback	Multi-proc	system	none	8/16-bit processors	4 level hierarchy, fixed links	hardwire & control store	6 joints & sensors at 10ms servo
[Kanade84]	Control direct drive robot with dynamics compensation	Multi proc	system	C, cartesian moves	6 TI32010DSPs, Marinco APB 3204 MC 68010 master	Multibus, shared memory	hardwared & control store	IDS & servo at 500Hz 10 MFLOPS
[Nash85, Przytula88]	Linear Algebra	Systolic array	system & processor	machine code	16x16 array processor ASIC	2 D mesh	lockstep synchronization	100-500 MIPS
[Nigam85]	Inverse Dynamics	Multi-proc	concept	machine code	6 processor module	global bus, local bus in module	centralized, microprogram	S 2.5, F 0.42
[Orin85, Orin86]	Inverse Dynamics Jacobian	Multi proc	system & processor	machine code	N 1 processors, custom FPU	pipeline	lockstep	ASIC @ 2Mhz S 1.5, F 0.2
[Kasahara85]	Inverse Dynamic	Multi proc	system	"C"	N 8086/8087 48k RAM/ROM	shared global memory	centralized	0.01 MFLOPS per processor
[Schwan85, Schwan86]	Robot control (GEM, CHAOS)	Multi proc	system & kernel soft	object based C	8 8086/8087, VAX 11/780 host	Multibus, message passing	distributed	2 msec per message
[Killough86, Herndon89]	Teleoperation with torque feedback	Multi proc	system	none	3 MC 68000s	Multibus, shared memory	distributed	Force feedback @ 100Hz with IDS
[Chen86]	Hand controller torque feedback (NYMPH, COSMOS)	Multi proc	system & kernel soft & control soft	C, control	8 NS32016/FPU, SUN-2 host	shared global memory	distributed	Force feedback @ 100 Hz with IDS
[Hayward86, Lloyd88]	Robot control (RCCL)	Multi proc	system & robot soft	kinematic equations in C control	Workstation host, PUMA robot controller	16 bit par from host to PUMA controller	host as trajectory generator, servo in PUMA controller	VAX 11/780 IKS in 2000, s SUN SPARC station 1 IKS in 170, s
[Kazanzides86]	Control (SIERA)	Multi proc	system & kernel soft	"C", cartesian moves	Armstrong array processors, MC68000 host	Multibus for host 2 D mesh serial links for array processors	control propracontrol	6 DOF robot
[Korem86]	Control and automation	Multi proc	system	C	N MC 68000s modules	global bus local bus in module	distributed	1.5 MIPS per processor
[Leahy86]	Robot control	Multi proc	system	"C", control, cartesian moves	VAX 11/750 host PUMA robot controller	Q bus from host to controller, shared memory	host trajectory generator, servo in PUMA controller	VAX 11/750 IKS in 2400, s
[Lee86, Chang88]	Pseudo-inverse Jacobian	Multi-proc	system	none	25 CORDICs dedicated function	pipeline configured from algorithm	hardwire	IKS in 40, sec S 1.38 F 0.06
[Narasimhan86]	Hand controller (CONDOR)	Multi proc	system & kernel soft	"C", control, cartesian moves	5 MC68000s SUN-2 host	Multibus message passing	distributed	6 MIPS
[Leung86]	Direct kinematics solution	Vector proc	logic	none	18-bit ALU	dual bus	hardwired	73 cycles for 6 DOF robot
[Paul86]	Force control	Multi proc	system & control soft	"C", force control	7 8086/8087, VAX 11/785 host	Multibus shared memory ethernet to host	distributed	hybrid control at 250Hz

Table 2.2 (cont'd)

Author(s)	Intended Application	Type	Design Level	Programming	Architecture: Function Units	Architecture: Communication	Architecture: Control	Performance features
[Juvaheri87]	Inertia Matrix	Systolic array	system	assembly	32 bit FPU 4 16 bit ports	global bus or 2 D mesh	microprogram	1< S< 2.2 E< 2.2%
[Bejczy87]	Telerobot control (UCCS)	Multi proc	system	none	NS32016 FPU 160k RAM/ROM	shared memory on Multibus	3 level hierarchy control store	14 joints at 1kHz servo
[Levin87]	Control	Multi proc	system	'C'	T414 transputers WEITEK FPU, PC/AT	dedicated links host shared memory	distributed control program	0.65 MFLOPS per processor
[Seshadri87]	Direct kinematics solution	Single proc	processor	machine code	AT&T DSP16	16-bit parallel bus	control store	DKS in 10 μsec with 16-bit accuracy
[Wang87, Butner88 Mangaser89]	Control (RIPS)	Multi proc	system & processor	'C' on TI DSP assembly on other processors	Discrete parts proc, TI32025 DSPs, 2 CORDIC chips	VMEbus for host, local bus shared memory	control prog from host 3-stage pipe	3x3 matrix-vector mult in 900ns IKS in 50 μsec
[Bilm88]	Robot control	Single proc	system	'C', cartesian moves	Intel 310, 80286 at 6 MHz	Multibus	centralized	100Hz servo
[Hug88]	Inverse kinematics & dynamics	Vector proc	processor	microcode	32 bit FPU	Vbuses	microprogram	6 MFLOPS S< 3
[Hayward88, Backes89]	Multi robot control (RALI)	Multi proc	system & robot soft	multi robot kine & dynamics equation in 'C', control	N MC 68020/68881 SUN-3 host	VMEbus shared memory	control program	two 6 DOF robots + sensors
[Narasimhan88]	Hand controller (CONDOR II)	Multi proc	system & kernel soft	'C', control, cartesian moves	5 MC68020/68881 SUN 3 host	VMEbus, shared memory	distributed	20 MIPS, 32 actuators @ 400 Hz
[Shalom88]	Control (SPARTA)	Multi proc	system & processor	assembly on ASIC 'C' on PC, cartesian moves	N 32-bit processor ASICs, PC/AT host	PC-bus	control program	5 DOF robot at 5kHz servo
[Andersson89]	Control (JIFFY)	Single proc	system & processor	'C'	VLIW Bit slice, 64 32-bit regs 4x32-bit buses locally	host on VMEbus, 4 local buses for Proc	control program	18 MFLOPS IKS+IDS+traj gen. in 48 μsec
[Clark89]	Hand controller (HIC)	Multi proc	system & kernel soft	'C'	4 MC68020/68881, SUN 3 host	VMEbus shared memory	distributed	20 MIPS
[Lee89]	Multisensor fusion	Multi proc	system & kernel soft	'C', sensor integration, cartesian moves	N microVAXs	ethernet, message passing	distributed	5 msec per message
[Kircanski89]	Robot Control (DESCARTES)	Multi proc	system & processor	assembly	Bit slice 32-bit FPU	2 D mesh on Multibus	control program	control law IDS for 6 DOF in 100 μsec 1< S< 2.2, E< 0.7
[Takamushi89]	Force control	Multi proc	system	'C' on PC assembly on DSP	NEC μPD77230 DSP, PC/AT as host	host on PC bus local bus for DSP	control program	force control at 1.4kHz
[Salkind89]	Robot control (SAGE)	Multi proc	system & kernel soft	'C', cartesian moves	MC68020s, PUMA robot controller	VMEbus message passing	distributed, servo in PUMA controller	traj. gen. at 100Hz from sensors
[Schmitz89, Stewart89]	Mult sensor fusion and robot control (CHIMERA II)	Multi proc	system & kernel soft	'C', cartesian moves	N MC68020/68881, Mercury 3200 FPU SUN 3 host	VMEbus shared memory and message passing	control program	cartesian state space control with sensors at 25Hz
[Wang89]	Robot control	Multi proc	system	'C'	4 MC 68020/68881, PC/AT host	VMEbus, shared memory	distributed control program	1.5 msec for IDS
[Miller90]	Cell control, sensor integration (PIPE, RIPL)	Multi proc	system & robot soft	'C++', RIPL language	N MC68020/68881, SUN-3 Host, commercial robot controller	VMEbus shared memory, user program runs on host	distributed, user on host, traj gen. on VME and servo in commercial controller	12 MIPS
[Fijani91(2)]	Inverse dynamics	Systolic array	Logic design	Load tables	multiple 32-bit functional units	2-D Mesh	hardwired	S>6
[McMillan91]	Dynamics simulation	Vector proc	Simulation	FORTRAN	CRAY Y-MP8 8 CPUs, each CPU 7 64-bit vectors of length 77	CRAY supercomputer bus	control program	S=5.2, E=0.67

MIPS=million instructions per second, MFLOPS=million floating point operations per second, FPU=floating point unit, ASIC=application specific integrated circuit, VLIW=very long instruction word, DOF=degree of freedom, IKS=inverse kinematic solution (6 DOF), IDS=inverse dynamic solution (6 DOF), DSP=digital signal processor. Parallel processing performance metrics used [Amdahl67]: S (speedup factor, where P is the number of processors and f is the fraction of computation that must be done serially) = $P/(fP+1-f)$. Note when S=1 then f=1 (no speedup), and S=P when f=0 (everything parallel). E (efficiency in executing algorithm): S/P. E is limited by the fundamental parallelism in the algorithm.

† The industrial PUMA controller is included because of its traditional role in robotics research systems. This table is adapted from [Leung88]

It is clear several trends seem to have emerged: in hardware, multiple processors on a common VMEbus backplane is undeniably the dominant architecture, perhaps because of its easy implementation, its low cost or the fact that the fastest commercial processors are always available

first on boards for this bus; in software, the era of special robot programming languages and home brew real-time operating systems is ending, most researchers now opt for a C language library interface (with C++ soon to follow) and off-the-shelf real-time kernels, similarly, in processors the use of high performance commercial microprocessors (with integral floating point units) has come to dominate; in sensors, integration is increasing rapidly, especially in force control and vision, lastly, coordinated multi-robot control systems have begun to emerge.

No doubt the embedded real-time 'black-box' approach offers the greatest hardware and I/O flexibility, however the basic architecture expounded by RCCL (workstation hosted supervisory layers with servos in the robot controller) may appear to be the 'way-to-go' in the near future, since it still offers the greatest prototyping and development environment bar none. It suffers from only one problem—the single processor workstation computing architecture and the limited real-time capacity of UNIX. For the moment, it appears that tightly coupled, parallel processors running real-time kernels, commercial or not, remain in fashion. However this approach is not always the panacea people believe it to be—*eg.*, porting code from SunOS to VxWorks is painful since they are, at best, compatible only superficially, starting with incompatible names of header files and escalating; the lack of process memory management leads to discarded memory every time a function is reloaded (quickly filling memory thus requiring a reboot which is impractical, not to mention annoying, for development);VxWorks pipes aren't the same as SunOS pipes; VxWorks task and semaphore routines don't have direct counterparts under SunOS; the VxWorks standard C library has differences from the SunOS C library, etc. The only question is when will workstation technology and real-time UNIX [Cole90] overcome their real-time limitations (see figure 2.13 below)? With the emergence of machines like the HP RISC-PA (HP/Apollo series 700), its astounding 34 MFLOPS capability, the continuing 'workstation wars' among the major vendors and microprocessor houses, and ever more specialized requirements for such CPUs [Horning91] (*ie.*, circuit boards at 100 MHz, wire-bonded multi-chip modules, etc [Mendelsohn91]), it will be increasingly difficult for single board computer manufacturers and their real-time kernel counterparts to keep up, let alone home-brew research hardware (*eg.*, SUN Microsystems already has 40 MHz SPARC based workstations whereas only a few SBC manufacturers even produce

SPARC-based boards and then only with the CPU speeds @ 25 MHz [Child91]; not to mention the fact that VxWorks is still in beta test for SPARC-based boards and the port to Mips-based boards has been abandoned[†]).

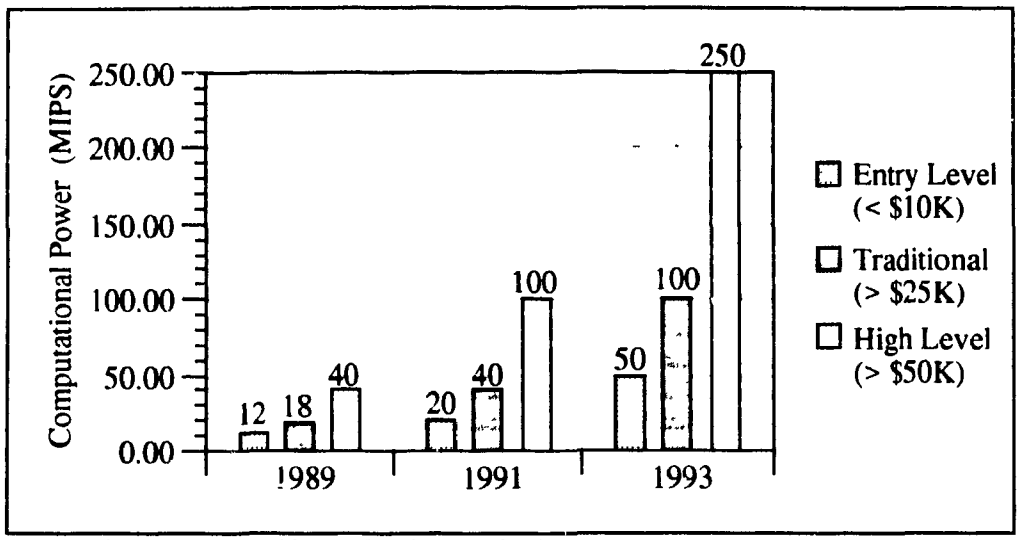


Fig. 2.13 Current and Future Workstation Performance
Workstation performance is heading up rapidly Even though the MIPS rating is an inaccurate absolute measure of computer performance, it is useful as a long term relative metric on system change No doubt, workstations in 1993 will number crunch robotics code at least 4 to 5 times faster than they did in 1989 Source [Leibowitz90, Rosing90]

2.7.2 Commercial Competition

Before closing this chapter it is important to take a brief look at the current developments in commercial systems. Even though most researchers dismiss such industrial systems out of hand, they have vastly improved in recent years. Most research controllers would now do well just to reach the performance levels and usefulness of modern commercial systems: advanced controller architectures, targeted at sophisticated force and vision sensing applications, employing multiprocessor architectures based on 32-bit CPUs and megabytes of memory around a common bus and providing powerful programming and motion control capabilities integrated with sensor feedback [Agapakis90, Scheinman88], even at the task level [Campbell90]; sophisticated servo algorithms applied with dynamic feedforward compensation (including observers) to counteract for

[†] To the best knowledge of the author, based on information from Wind River Systems in Sept 1990 Of course, things change fast in this industry

joint coupling and other nonlinear effects [Hiroshi90, Adept90, Seiko90]; the use of high speed digital signal processors to achieve very high servo sampling rates (above 3 kHz) and superior performance, *eg.*, 1.1 sec cycle time, 10 m/s velocity, 5 g acceleration [ANSI/RIA89, Intelledex90, Seiko90, Adept90]; compatibility with state-of-the-art off-line programming and simulation systems such as CimStation™ [Craig88(2)]; and aggressive pursuit of many novel applications, especially by the Japanese [Whittaker90].

™ A trademark of Silma Inc.

3 Kali Implementation

For 'tis the sport to have the engiuer hoist by his own petar...
—William Shakespeare (1564-1616)

3.1 Kali Overview

As a robot control system, *Kali* is an instantiation of the robot control hierarchy following the principles previously set forth. However it is only a level 1 design (trajectory generator) which is targeted at control programming for multiple cooperating manipulators. The traditionally thorny issues of user interface design, task simulators, programming (level 2), and servo design are not treated as direct concerns, but are considered independent of *Kali* and are treated more as 'applications' for the user to implement as he wishes. Hence at its core, *Kali* is merely a collection of C language library functions that provide the user a set of primitives for multi-arm control Applications are built by creating, combining, and manipulating functions or objects defined from these primitives. The most basic of these is the notion of a motion system—two or more arms are lumped together as one motion system which describes the movements and constraints associated within the desired control frame. As with more traditional robot control systems like RCCL [Lloyd88], the user describes the control frame in terms of synchronization, destination, velocity, and force. The coordinated motion of cooperating manipulators is achieved by kinematically constraining the manipulators to a control frame. The manipulators can move in synchronous, close cooperative fashion, able to grasp and move about objects together (*eg.*, forming closed kinematic chains through the common load and operating within sufficiently accurate mutual spatial-temporal constraints). When using such a scheme, a motion system can be seen simply as a point in the velocity and force subspaces. This has particular application in low gravity environments like outerspace where inertia forces become dominant—the original impetus behind *Kali* [Backes89, Hayward89, Hayati90]—or in redundant manipulators like macro-micro arm pairs

and in robotic hands. For more details on the theory behind *Kali*'s coordinated motions see [Hayward88, Hayward91]. Trajectories themselves are specified as a series of Cartesian straight line segments where velocity is controlled. Between these segments are transitions or blending phases where acceleration is controlled Overall this approach treats the concept of motions systems much in the same way that more traditional manipulator oriented systems treat individual robots—essentially enhancements of [Paul81]. For details on the trajectory generator and programming consult [Hayward88, Hayward88(2), Nilakantan88].

Because Kali deals with the motion of one or more robots in cooperation, there may exist a great deal of mechanical coupling among the manipulators which means such a system can only be effectively controlled with a control loop that closes around the entire system, not just around each individual manipulator. As a result, this leads to the need for one very sophisticated trajectory generation task which takes into account all spatial-temporal constraints of the entire system Consequently, the basic architectural premise in *Kali* is that there is only one trajectory generation task whose purpose is to compute nominal set-points for every motion system. All other supporting tasks, such as the servo processes for the manipulators, must be synchronized to this one master process. Within the trajectory process itself, however, motion systems are treated much as processes that traverse from one state to another state, like from 'running' to 'terminated', and they are time shared for every *Kali* quantum period, *ie.*, set-points for each of the motions systems are generated one after the other [Nilakantan89].

In addition, the *Kali* motion control library is completely system independent of both robot hardware and processor architectures, *ie.*, the output of its trajectory generator is simply a nominal trajectory specification in Cartesian space. The challenge is to implement all the support hardware and software necessary to run this library. It is the purpose of this chapter to describe in detail the first implementation of *Kali* made at McGill University as an example of a robot control system.

3.2 Implementation

The implementation follow very much the typical robot controller anatomy as described in figure 2.2 (section 2.2.3). A set of processes is needed which are either high priority synchronous processes running at the trajectory generator rate, synchronous processes running at the servo rate (a multiple of the trajectory generator rate), or low priority asynchronous processes. Because at the time *Kali* was conceived one CPU was deemed to be vastly insufficient for advanced robot control, the *Kali* supporting processes are distributed over an array of processors connected by a simple backplane bus[1], though the main trajectory generator process cannot be parallelized. The synchronous processes are synchronized by a global clock interrupt called the 'wall clock'. Each processor has a maximum of two resident processes: one asynchronous running in the foreground, and one synchronous (implemented as an interrupt routine tied to the global clock) running in the background.

3.2.1 Real-Time O/S

In order to make the minimum of assumptions with respect to the synchronization and interprocessor communication facilities, as well as the performance of the underlying operating system, no explicit process facilities like creation or deletion are necessary; however a global (across CPU boards) clock to which can be attached one interrupt routine per processor and a shared memory mechanism are required:

- *Wall Clock.* A mechanism for interrupting all the CPUs using a single clock. This is most easily accomplished by designating the hardware timer on one of the CPUs as the wall clock. An interrupt routine tied to this timer then issues interrupts to the other CPUs on the bus (this can be accomplished via the mailbox[2] facility available for most single board computers). This scheme results in a slight clock skew on the other processors as the interrupt

[1] As chapter 2 shows many processors in parallel are no longer a prerequisite for robotics

[2] A dual-ported memory location on the processor board that when accessed interrupts the CPU.

routine takes time to get around to interrupting all the CPUs, but it is not significant, *eg.*, if there are 8 CPUs on bus and it takes 1 µsec to generate an interrupt across the bus (very conservative), then the maximum skew is 8 µsec—less than 1% in a 1 msec servo loop. The *Kali* trajectory generator also requires that the clock be accurate to within a millisecond and that it keep a total running time in milliseconds. In addition, as a safety feature, a reliable hardware mechanism must be used to ensure that should a servo algorithm fail to execute (*eg.*, a hardware fault or software crash) or if the servo requires longer than one interval to complete (*eg.*, programmer miscalculates the execution time or other processes make many accesses to shared memory servo data structures which delays servo processing), then a hardware mechanism to halt/disable all manipulators is engaged.

- *Shared Memory.* Since multiple CPUs were deemed necessary, a means of creating shared data structures in memory is required, *eg.*, queues between the processes. These could be allocated in fixed memory locations at start up or with dynamic creation/deletion capabilities. Since more than one process may wish to update a shared memory location, at least simple binary semaphores and queues are required with the following types of synchronization:

  Type 1: Queues or FIFOs from asynchronous processes to synchronous ones which may or may not reside on the same CPU.

  Type 2: FIFOs from synchronous processes at one rate to synchronous processes at another rate (which is an integer multiple of the first process' rate).

  Type 3: Atomic flags or semaphores raised by asynchronous processes and inspected by synchronous processes, ostensibly to temporarily synchronize a data access. This can be accomplished by the 'test-and-set' mechanism for

simple binary semaphores found on modern computer buses (see Appendix A).

Type 4: Data updated or utilized by any process with no need of explicit synchronization, *eg.*, data output to an operator console.

- *Give up or Abort.* This is the software fail-safe to shut down the system when a major error condition occurs and the system doesn't know how to proceed (*eg.*, trajectory generator hits a kinematic dependency or limitation of the arm). Since all critical processes in the system are tied to the wall clock, this shut down can accomplished by first disabling power to all the system manipulators and stopping the system clock. The user can then clear the error condition and restart the clock (thereby resuming the system). Asynchronous processes (other than the user code) in the system are designed to operate from data provided by synchronous (recovering from such software crashes is easy in *Kali*).

3.2.2 Servo Control

Since *Kali* only requires that the manipulator servo control maintain set-point positions, many forms of control architecture (level 0) are possible. For the current *Kali* implementation no standard control algorithm is furnished, but instead a hardware independent programming interface is provided for individual joint controllers. Early on, a major goal for *Kali* was the need for a rapid prototyping environment for control design. Thus it is desirable for every joint for each robot controlled by the system to have one of many user programmable control algorithms attached to that joint with its parameters changeable on-line, so that these servos could be changed on-the-fly permitting the user to test many different algorithms in quick succession and modifying parameters as desired. For advanced Cartesian based control programming a similar interface with same capabilities is also desired, but not yet implemented.

3.2.3 Other Considerations

- *High level language programming.* Programming in the high level language, C, is required from all levels in the system, from joint-level control to highest supervisory level. There is never the need to program in assembly language. Supervisory levels have software designed more around the objects that comprise the system and its environment rather than just on the functions it performs (*ie.*, kinematic loops of transform objects and functions).
- *Transparent host programming environment.* The host system must provide the user access to his familiar development environment. Since UNIX® is 'the' popular system, it is desired to have as similar environment as possible on the real-time target including standard system level and library calls. Ideally, the user should not be aware the he is executing code on the target system, since it appears so much like his 'natural' environment. VxWorks™ [Wind87, Williams90] is the real-time kernel of choice, in this case, for its high degree of integration with the UNIX host workstation (it frequently happens in VxWorks that users mistakenly reach for the mouse expecting a certain graphical-user- interface-like functionality). However real-time UNIX when it achieves comparable performance is the ideal environment.
- *Portability:* To protect the large investment in human resources necessary to create a complex real-time control system, every provision must be made to ensure compatibility with changing hardware and software. Needless to say this includes CPU and bus architectures, as well as multiprocessing & real-time programming paradigms and programming languages.

® UNIX is a registered trademark of Unix System Laboratories, formerly wholly owned by AT&T
™ A trademark of Wind River Systems Inc

- *Reliability:* A predictable, reliable system is a necessity for experimentation, safety, and portability. Popular processor/bus architectures and software development environments are usually the most flexible and reliable (the two are not always opposing constraints).
- *Third Party Support:* How well are the third party system hardware and software supported and maintained by the vendor? Portability ensures a high degree of independence, and manufacturer supported extensibility and maintenance, in both hardware and software, considerably simplifies portability and flexibility. Poorly addressing this aspect can compromise all of the above.

3.3 Process Model

For the McGill implementation, the *Kali* processes are as follows:

3.3.1 Trajectory Generator Process (TG)

As in RCCL, the main synchronous process is the trajectory generator whose task is to compute the nominal set-points for the all the manipulators in the system. This process is the heart of the system, everything is synchronized to it. It has a two function interface to connect the lower servo layer: write_t6 outputs the homogeneous transform representing the Cartesian set-point for a manipulator ('T6' is the traditional term for the end-effector link in a 6 DOF robot), the transform is converted via inverse kinematics to joint angles before being sent to the servo control process (SIO) (a check for singularity is also made at this point, if it fails the system aborts; however work is underway to remove this limitation); and read_t6 which reads back the current Cartesian position. The motion library also supplies a joint mode interpolator which moves a manipulator directly in joint-space, in this case the outputs go directly to the SIO.

3.3.2 User Process (UP)

This is the process that contains the 'robot program'. Its main functions are to setup kinematic loops, define motion system control specifications (*eg.*, max velocity, transition time, etc.), issue motion requests, interface to the user and the external world. The user process runs on the same CPU as the trajectory generator. It is the foreground process whereas the trajectory generator is the interrupt or background process. The user code makes its motion requests asynchronously which are synchronized to the trajectory generator via a queue (as described in figure 2.2 and section 2.2.2). The user process runs freely until a wall clock interrupt transfer control to the trajectory generator which then creates a set-point for each robot in the system. Since the user code and trajectory generator share the same processor with the TG having priority, it is vital that the trajectory generator execute as quickly as possible to allow enough time for user code to run. Performance results show that a 20 MHz MC68020/68881 processor can handle one or two motion systems at 25 Hz trajectory generation rate, provide information from the computation of the dynamic models is made available by another processor.

3.3.3 Servo I/O Process (SIO)

This process runs at the servo rate and gathers sensor information, such as joint position from encoders or force readings from force sensors, converts them to standard angles (*ie.*, degrees) and forces (*ie.*, Newtons). It also checks against maximum bounds (*ie.*, maximum allowable mechanical movement of each joint) and tracking error (a simple means of checking against a 'run away' robot by measuring servo error). It then 'feeds' the servos by first interpolating the TG set points and then placing the resulting position demand, position, etc in shared memory The interpolation is accomplished as a FIFO interfacing the different data rates: TG set-points on the input and servo position demand on the output (as described in figure 2.2 and section 2.2.2) The SIO performs a dynamic loading scheme among the servo CPUs so that the user need not know the exact timing of his or her algorithms. The only requirement is that any servo must be able to

complete one servo cycle on its designated CPU. After the servo cycle completes, the SIO picks up the torque demand from the servos, checks against motor current limits and outputs this demand to the amplifiers. The SIO is, in effect, the bottleneck in level 0 of the *Kali* system.

3.3.4 Servo Process (SV)

These servo processes may reside in one or more CPUs and each may run one or more joint servos up to the number of joints controlled in the system, *eg.*, worst case one joint servo per CPU. Each SV process run on a separate CPU, of course, is a servo synchronous process (the SV processes are effectively gang-scheduled by the wall clock). There may be as few as one or as many as eight SV processes (the max number of robot joints). Each SV is a servo execution handler that performs a simple dynamic loading algorithm. That is a user servo need not know on which SV CPU it will run. This is determined dynamically at run time. As the user's servo algorithm becomes more computationally intensive it will automatically consume more CPU resource. All variables for the servo algorithm are stored in shared memory making this possible. At present, each servo is independent and servos one joint. It is executed at every servo interval with the basic servo parameters, *ie.*, position demand, current position, output torque, etc. as well as a pointer to private data used internally by the servo algorithm. Thus from the servo programmer's point of view he need not be aware that all his data is in some shared memory segment since all synchronization and access rights are transparently taken care of. A Cartesian servo mode, still in development, will be forth coming. This will consist simply of a different process which is activated instead of the joint servo one. At present temporary 'hacks' have been made to test Cartesian control methods such as damping and impedance control using a force sensor at the end-effector [Hayward88(4)].

Each joint may have many different (user definable by a *Kali* system compile) servo algorithms, any one of which is active for any joint at a time (*ie.*, joint #1 may have an adaptive PID whereas joint #2 may have a plain PID.) In addition, the same code may be used for each joint since the data used by the servo algorithm for each joint is user-instantiated at run-time. The user may switch servo algorithms and modify them while on-line to facilitate experimentation. In addition,

the servo algorithm closes the loop around the kinematic angle position and motor torque instead of encoder counts and DAC (Digital to Analog Converter) units (both are simple linear transformations). This makes for cleaner programming as the servo is truly I/O hardware independent and is a function only of the electromechanical properties of the robot The appropriate conversions like encoder to radian angle, limit checking and amplifier to torque conversion are defined in the robot/amplifier driver and executed by the SIO before feeding the servos.

Because the SIO translates encoders to angles and torque demand to DAC units (thus they execute in the servo loop and take precious execution time away from the servos), almost as much processing time is spent doing this as in executing a the servo algorithm like a simple PID. Consequently, a MC68020/68881 can only run 4 PID servos @ 500 Hz (2 msec using 32-bit floating point arithmetic) in the SV process. Thus 2 SV processes are required to servo a six DOF robot. Normally such a processor could easily handle 8 or more PIDs using integer arithmetic @ 1 kHz with all servo data local to the CPU, however it was deemed worthwhile to sacrifice speed for easy programming and experimentation.

3.3.5 SIO/SV Dynamic Loading Algorithm

The SIO is basically divided into three parts: (a) get joint position, (b) feed servos, (c) get servo torque demand and output. The dynamic loading algorithm is based on simple binary semaphores one for each servo which is cleared by the SIO after getting previous sample servo data in part (b). The SV processes in the meanwhile each use test-and-set to see if the servo has been executed. This mechanism does introduce a delay to the servo processing time for the SIO code, in the worst case: time for code block (a) + time for code block (b) (see figure 3.2). However it is still much less than 2 cycles from joint position input to torque output.

3.3.6 Feedforward Dynamic Compensation

This is an asynchronous process running a separate CPU. Update is slow (around the TG rate) and it need not be synchronized [Kircanski86] (see section 2.2.3). The SIO reads the

compensation torque and adds it to the amplifier torque demand automatically. The dynamic compensation has yet to be included in *Kali*, but the software 'hooks' are already in place.

3.3.7 Viewer Process

A separate asynchronous process called the 'viewer' runs on a separate CPU and simply picks the relevant data from shared memory and displays it at about 25 Hz refresh so that the user can monitor the system status like robot position, tracking error, etc.

Figure 3.1 below summarizes of all processes and their basic interrelationship.

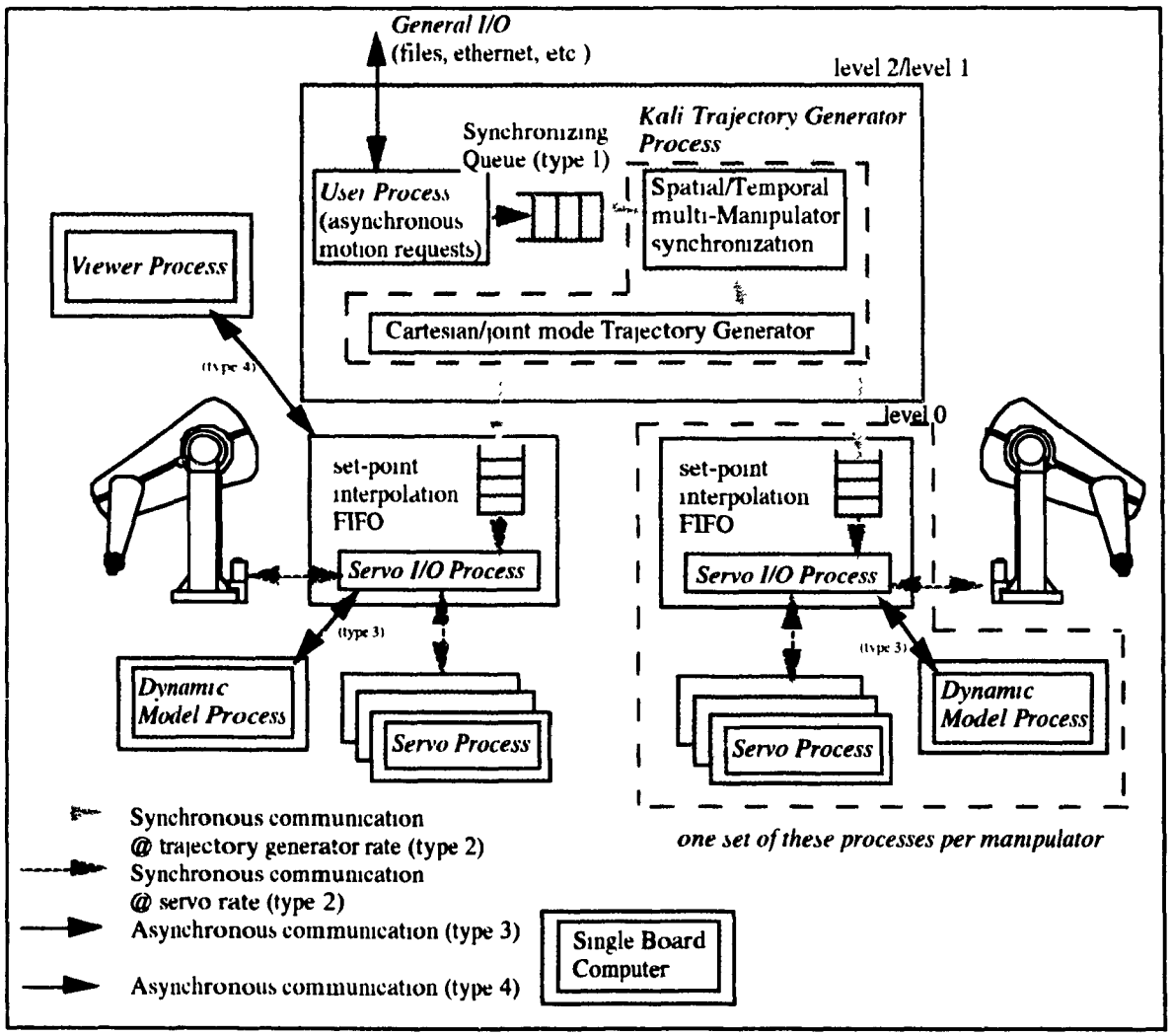


Fig. 3.1 The *Kali* Process Organization

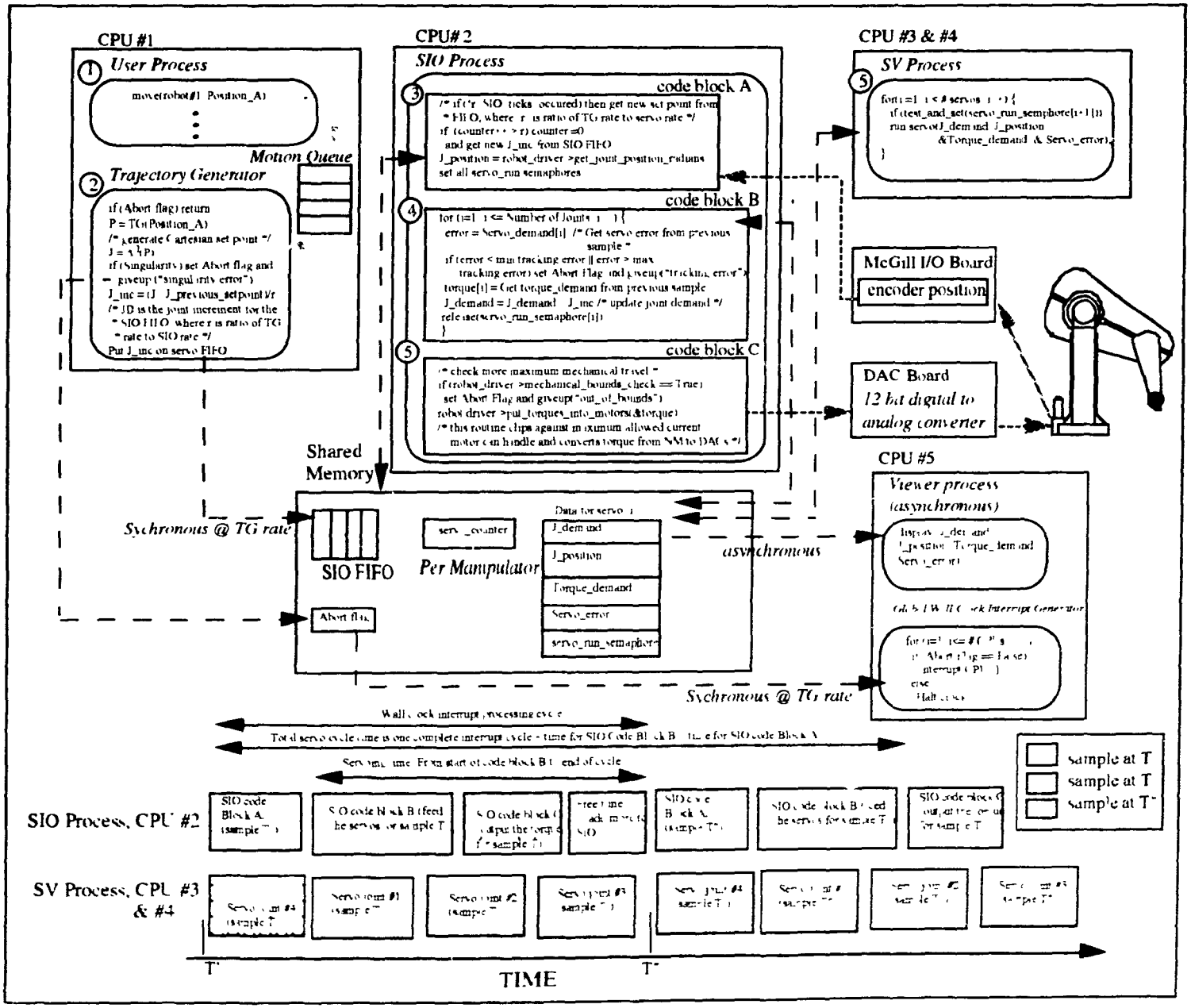


Fig. 3.2 Detailed Block Diagram of *Kali* Processes & Timing

3.4 Servo Programming Interface

The servo programming interface permits the user an easy way to test servo algorithms without needing to know intimate hardware detail of the system, *ie.*, addresses for I/O boards, etc. Since most research control algorithms are based on the computed torque method, output from the servos is a torque demand should drive a linear torque mode actuator.

The user must write his/her servo function to accept the following software interface:

```
int servo_function(p_servo, demand, position, p_torque, p_error)
/* pointer to structure private to this servo code for a particular joint on a particular robot */
  ServoFunctionStruct  *p_servo;
                Real   demand,          /* demand from trajectory generator */
                       position,        /* current robot position */
                       *p_torque,       /* desired torque put here are computed */
                       *p_error;        /* previous error, put new error after
                                           computed */
```

The algorithm may update its private data structure any way it wishes, but it must update the error, pointed to by p_error, and return a desired torque, pointed to by p_torque. The demand, position, and error are all in radians in kinematic angle coordinates. The torque is in Newton-Meters (usually at motor shaft, depends on robot and driver). The algorithm returns a status condition of the servo. The interface also requires that the user provide an initialization function for each servo. This routine is not called by the used directly, rather it is executed automatically upon a starting the *Kali* trajectory generator. The user may have a precomputed static structure pointed to by p_init_data for each different joint using the same servo algorithm. The initialize servo function has the following format:

```
void init_servo_function(p_servo, p_init_data, sampleInt)
      ServoFunctionStruct *p_servo,     /* pointer to private data structure */
                          *p_init_data; /* initial data for that structure */
             MsTime       sampleInt;    /* sample interval time in millisecs */
```

The user must also provide his/her own functions to alter the private servo algorithm data using the get_servo_data and put_servo_data functions for on-line servo data manipulation Since these functions are executed by the asynchronous user process, they require a synchronization with the SV process lest a corruption of data occur. This involves delaying the execution of servo until the 'put' or 'get' function is performed while a structure block move operation takes place to transfer the servo data structure. In practice it may happen that the servo is actually 'skipped' (not executed) for one interval, but experience has shown this does not have any noticeable (*ie.*, hazardous) impact on motion performance. The function interfaces are illustrated below

```
put_servo(robot_id, joint_id, servo_id, p_data, size)
        int  robot_id,
             joint_id,
             servo_id,
        char *p_data;
        int   size,

get_servo(robot_id, joint_id, servo_id, p_data, size)
        int  robot_id,
             joint_id,
             servo_id;
        char *p_data,
        int   size,
```

The data pointed to by p_data into the current user servo state from joint joint_id for servo algorithm servo_id on robot robot_id. The data to transfer is size bytes long Return '0' if successful or '-1' if bad id. A host of other functions exist which simply change one element of the servo data structure, such as enable or disable an individual servos, enable or disable the range checking, or set the tracking error tolerance (*ie.*, the 'safe' variable in the 'ServoStruct' used as the maximum servo error tolerance, this is easy to set since servo error is always in joint angles, not encoder counts). All access to this critical shared memory structure (*ie.*, the 'ServoStruct') must be synchronized. Figure 3.3 shows the data structures used by the SIO and SV processes and hence the user servo algorithm Figure 3.4 a shows a complete servo code example for a simple PID.

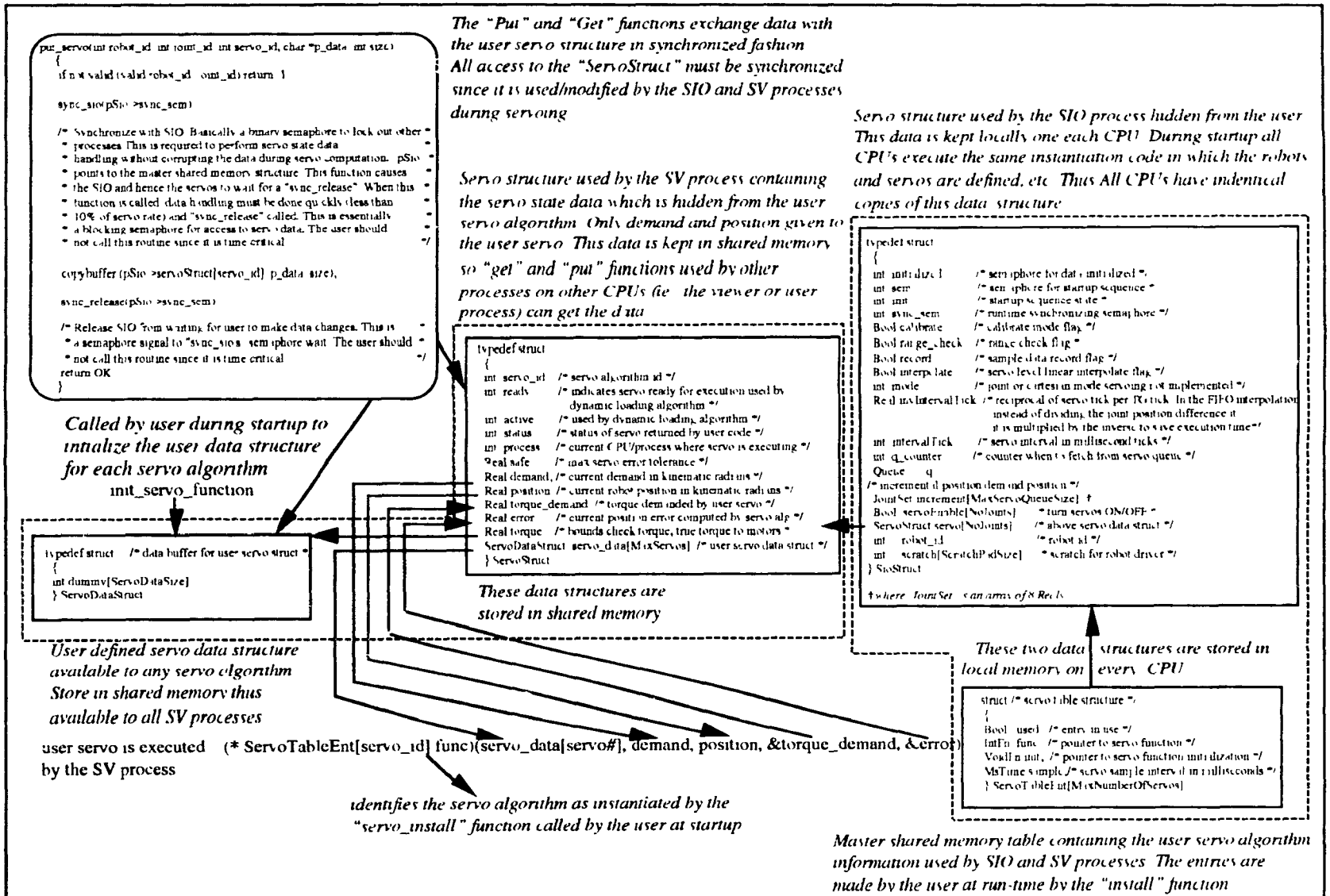
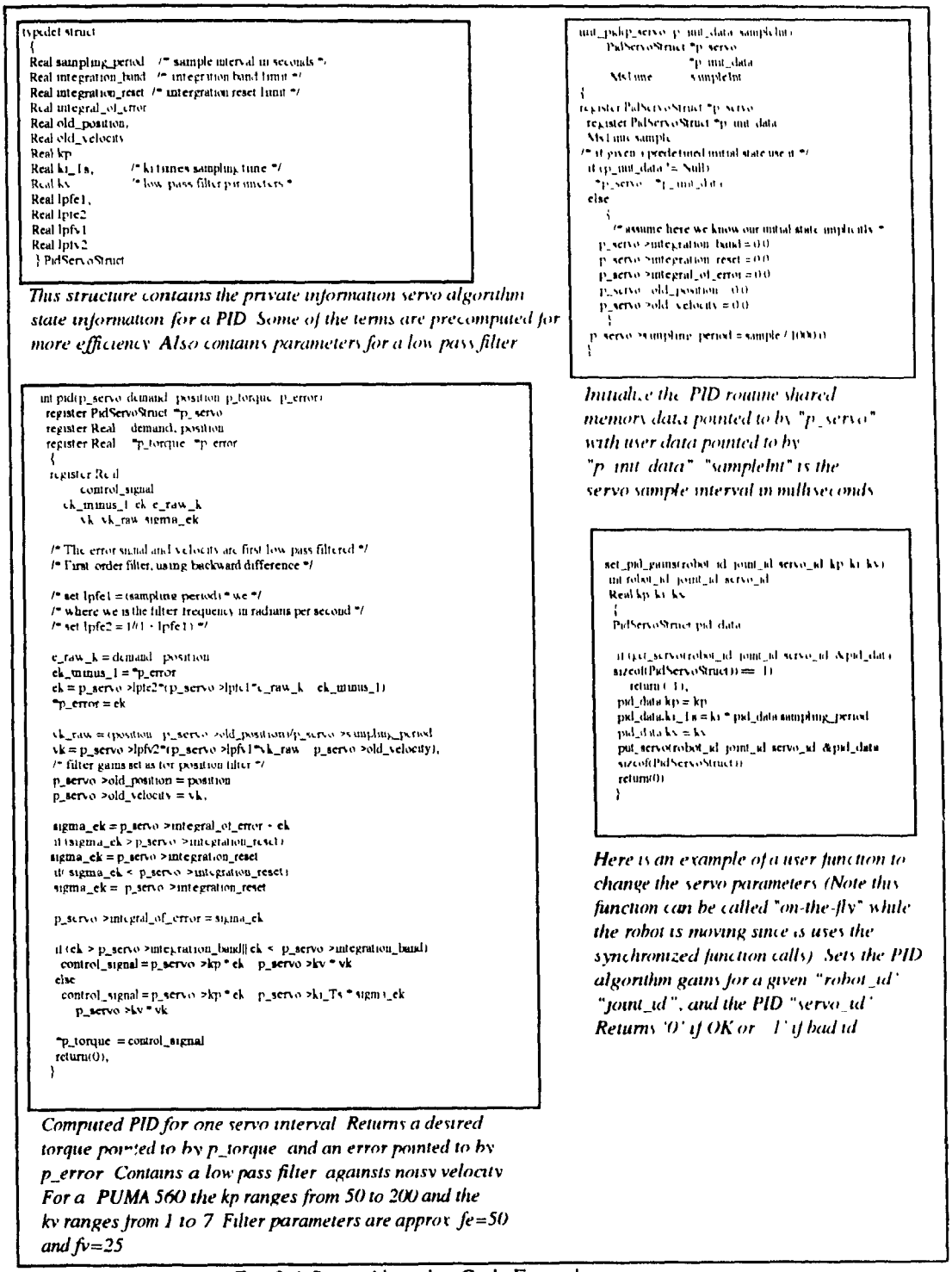


Fig. 3.3 Servo Data Structures



```
typedef struct
{
Real sampling_period   /* sample interval in seconds */
Real integration_band  /* integration band limit */
Real integration_reset  /* intergration reset limit */
Real integral_of_error
Real old_position,
Real old_velocity
Real kp
Real ki_Ts,          /* ki times sampling time */
Real kv              /* low pass filter parameters */
Real lpfe1,
Real lpfe2
Real lpfv1
Real lpfv2
} PidServoStruct
```

This structure contains the private information servo algorithm state information for a PID Some of the terms are precomputed for more efficiency Also contains parameters for a low pass filter

```
int pidp_servo_demand  position p_torque  p_error)
 register PidServoStruct *p_servo
 register Real    demand, position
 register Real    *p_torque  *p_error
 {
 register Real
       control_signal
    ek_minus_1  ek  e_raw_k
      vk  vk_raw  sigma_ek

 /* The error signal and velocity are first low pass filtered */
 /* First order filter, using backward difference */

 /* set lpfe1 = (sampling period) * we */
 /* where we is the filter frequency in radians per second */
 /* set lpfe2 = 1/(1 + lpfe1) */

 e_raw_k = demand  position
 ek_minus_1 = *p_error
 ek = p_servo >lpfe2*(p_servo >lpfe1*e_raw_k   ek_minus_1)
 *p_error = ek

 vk_raw = (position  p_servo >old_position)/p_servo >sampling_period
 vk = p_servo >lpfv2*(p_servo >lpfv1*vk_raw   p_servo >old_velocity),
 /* filter gains set as for position filter */
 p_servo >old_position = position
 p_servo >old_velocity = vk,

 sigma_ek = p_servo >integral_of_error + ek
 if (sigma_ek > p_servo >integration_reset)
sigma_ek = p_servo >integration_reset
 if( sigma_ek < -p_servo >integration_reset)
 sigma_ek = -p_servo >integration_reset

 p_servo >integral_of_error = sigma_ek

 if (ek > p_servo >integration_band||ek < -p_servo >integration_band)
  control_signal = p_servo >kp * ek   p_servo >kv * vk
 else
  control_signal = p_servo >kp * ek   p_servo >ki_Ts * sigma_ek
     p_servo >kv * vk

 *p_torque  = control_signal
 return(0),
}
```

Computed PID for one servo interval Returns a desired torque pointed to by p_torque and an error pointed to by p_error Contains a low pass filter againsts noisy velocity For a PUMA 560 the kp ranges from 50 to 200 and the kv ranges from 1 to 7 Filter parameters are approx fe=50 and fv=25

```
int pidp_servo_p_init_data  sampleInt)
     PidServoStruct *p_servo
                 *p_init_data
    MsTime       sampleInt
{
register PidServoStruct *p_servo
register PidServoStruct *p_init_data
MsTime sample
/* if given a predetermined initial state use it */
if (p_init_data != Null)
  *p_servo   *p_init_data
else
   {
    /* assume here we know our initial state implicitly */
   p_servo >integration_band = 0.0
   p_servo >integration_reset = 0.0
   p_servo >integral_of_error = 0.0
   p_servo  old_position   0.0
   p_servo >old_velocity = 0.0
   }
p_servo >sampling_period = sample / 1000.0
}
```

Initialize the PID routine shared memory data pointed to by "p_servo" with user data pointed to by "p_init_data" "sampleInt" is the servo sample interval in milliseconds

```
set_pid_gains(robot_id, joint_id, servo_id, kp, kv)
int robot_id, joint_id, servo_id
Real kp, kv
 {
PidServoStruct pid_data

if (get_servo(robot_id, joint_id, servo_id, &pid_data,
sizeof(PidServoStruct)) == -1)
   return(-1),
pid_data.kp = kp
pid_data.ki_Ts = ki * pid_data.sampling_period
pid_data.kv = kv
put_servo(robot_id, joint_id, servo_id, &pid_data,
sizeof(PidServoStruct))
return(0)
}
```

Here is an example of a user function to change the servo parameters (Note this function can be called "on-the-fly" while the robot is moving since it uses the synchronized function calls) Sets the PID algorithm gains for a given "robot_id" "joint_id", and the PID "servo_id" Returns '0' if OK or '-1' if bad id

Fig. 3.4 Servo Algorithm Code Example

3.5 Robot Drivers

The *Kali* robot driver is a set of functions entered in a table. Figure 3.5 below illustrates

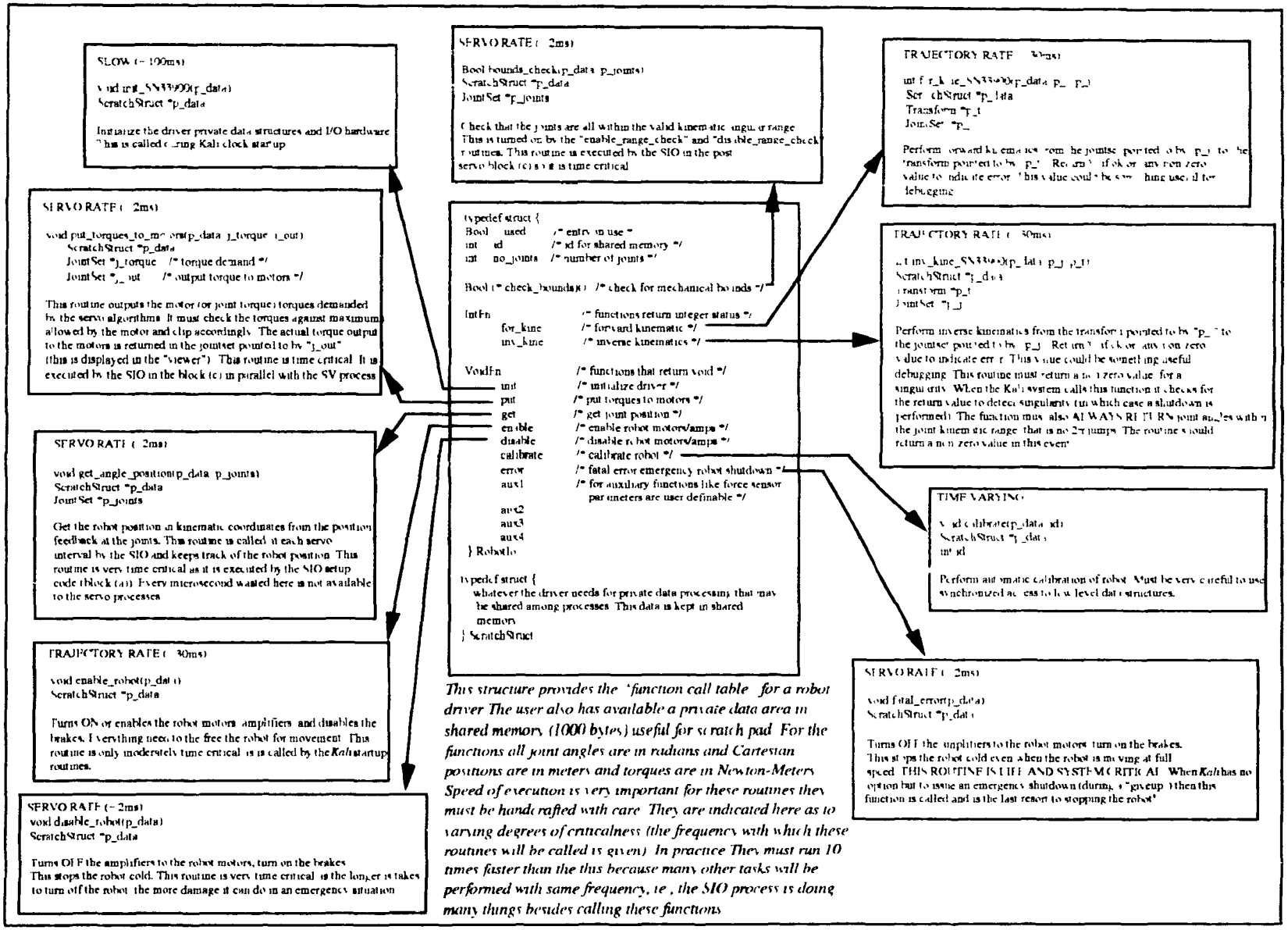


Fig 3.5 Kali Robot Driver Interface

3.5.1 Calibration

Robot calibration is is performed at two different levels: absolute kinematic calibration of the manipulator by precise external measurement (*eg.*, laser interferometry), and by using joint level absolute transducer sensing mechanisms (*eg.*, potentiometer feedback). The former is immune to robot link and joint variation (*ie.*, gear backlash and joint thermal variations), but requires much time a difficult setup. The latter can be performed on-line by the robot controller and so is discussed here (the assumption is that an absolute kinematic reference data is available for this operation). Most robots use encoder (index reference)/potentiometer method for calibration. This method involves noting that a) the encoder provides one index or reference pulse per revolution, b) the encoder is usually motor shaft mounted, thus the number of index pulse equal the gear ratio, and c) the A/D (analog to digital converter) reading the potentiometer must be able to reliably distinguish between the 'gear ratio' number of index pulses (*ie.*, at least one bit resolution than necessary). The idea is to assign a potentiometer value to each motor index pulse using the accurate external calibration reference. The exact kinematic angle must be determined when the robot is at a given index pulse and the potentiometer value noted. A simple trick here to save time with a minimal loss in precision is that once that exact potentiometer value for one index pulse in a joint is determined, the external reference is no longer necessary, because the joint can be servoed to each subsequent index pulse and the potentiometer value noted since the index pulses are a known angular distance apart determined from the precise gear ratio (*eg.*, for the PUMA gear ratio is 75:1, therefore index pulses are precisely 360°/75 = 4.8° apart). Another even shorter method involves taking only a few of these data pairs and linearly interpolating the potentiometer values for the other index pulses. RCCL uses this technique and it gives calibration results precise to the manipulator accuracy for a PUMA.

The actual on-line calibration procedure simply involves servoing a joint to an index pulse, getting the potentiometer reading and comparing against the known potentiometer/joint angle

(index pulse) table. Which ever entry in the table is closest to the reading determines the joint angle (see Figure 3.6 below).

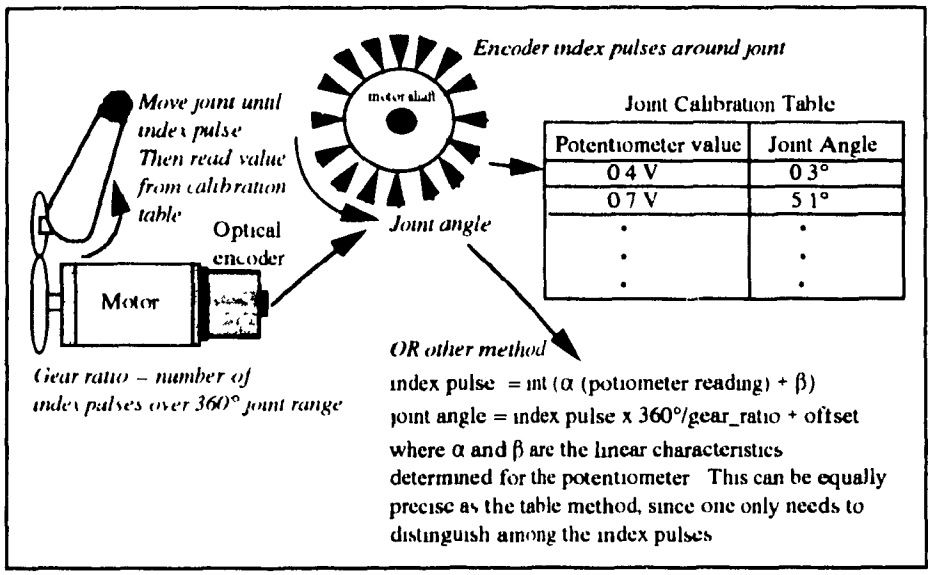


Fig. 3.6 On-line Robot Calibration Methods

Kali uses the potentiometer linear interpolation method for on-line robot calibration.

3.6 Real-Time Interface

The real-time layer forms a critical part of the *Kali* system. It consists of basic shared memory management and wall clock synchronization mechanisms described as follows:

3.6.1 Shared Memory

The *Kali* shared memory management is a somewhat simplified version of the UNIX System V (version 2 or later) scheme [AT&T87]. A 'master' linked list of the shared memory 'ids' or keys point to the individual blocks of shared memory. Access to the list for creation or deletion is through a synchronizing semaphore. All shared memory pointers are stored in relative offset form (*ie.*, pointer = base + offset), so that shared memory addressing is independent of the CPU board bus address windows (the *Kali* code itself, though, does not maintain its data pointers in this manner, assuming instead that all CPU boards have the same address window. Remember, the shared

memory scheme is an implementation dependent sub-system supporting the *Kali* library) See figure 3.7 below.

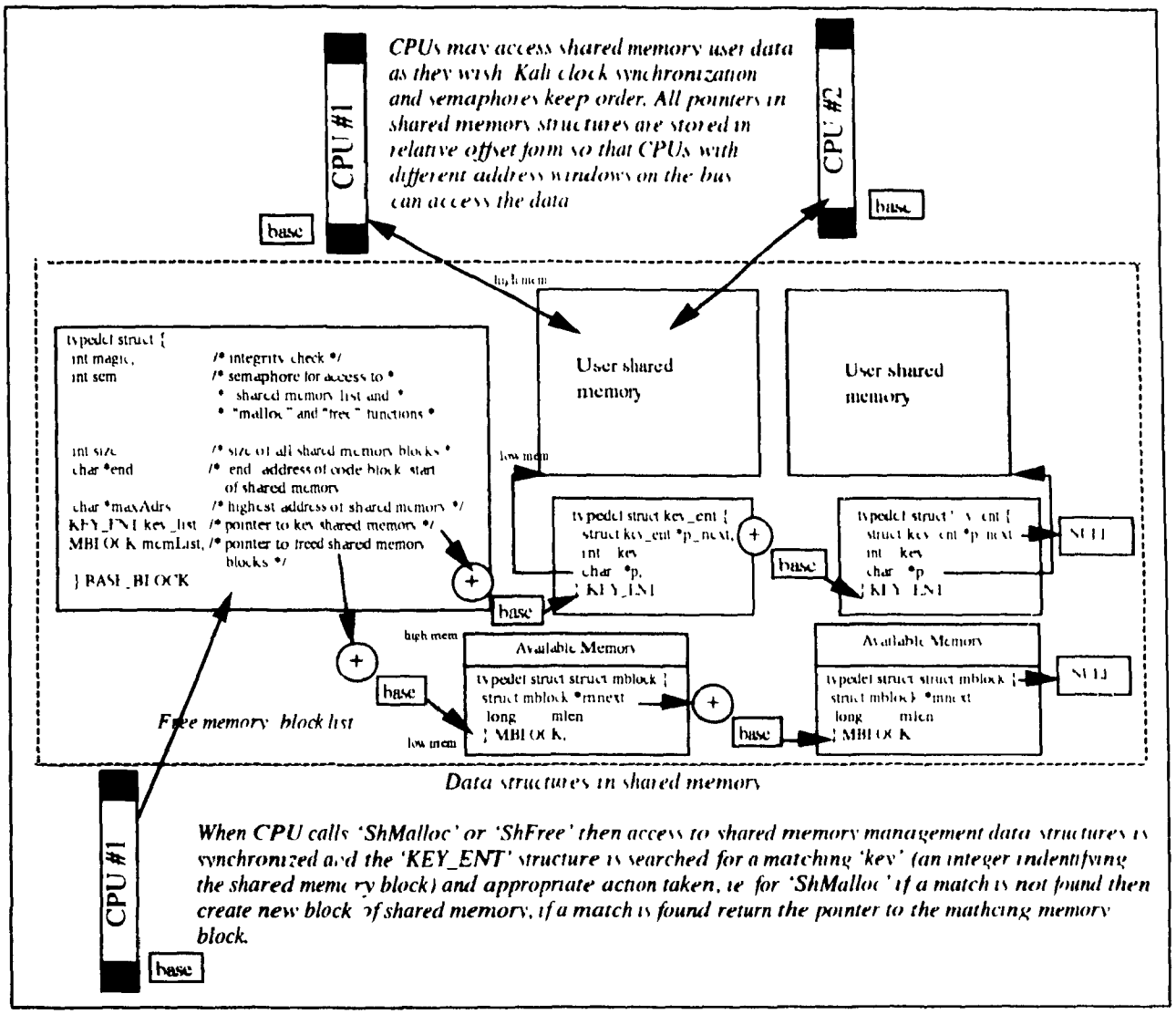


Fig. 3.7 *Kali* Shared Memory Data Structures

The shared memory functions are defined as follows:

```
char *shMalloc(key, size)
int key,    /* shared memory 'id' */
int size;   /* size of shared memory block to allocate in bytes */
```

The function searches the 'master' list to see if the key is already in use, if it is then that block of shared memory has already been allocated and the function merely returns the pointer to the shared memory (in full pointer form). In this way synchronization in creating/accessing shared memory blocks is made easy. All CPUs needing a particular block of shared memory execute the

same code (only one CPU actually creates the block). The function returns '-1' in the event that the insufficient shared memory exists to satisfy the request.

```
int shFree(key)
int key,
```

This function deletes the shared memory block identified by key. Returns a '-1' is the The memory is not cleared, but only moved to the pool of memory available on the free memory block list Memory is always collected into the largest possible blocks when some is freed (a sort of garbage collection routine) Initially the entire shared memory consists of one block, but it is slowly broken up into smaller pieces (external fragmentation) as it is allocated/freed and the free memory block list grows. This can lead a high degree fragmentation of available memory, so that subsequent allocation requests cannot be met even though enough sufficient memory exists in different blocks ([Randell69] discusses some of the basic issues involved in memory fragmentation). In addition, these routines must execute quickly and efficiently. Their design is difficult, since they must not only be fast (*ie.*, efficient garbage collection), but they must also not disturb existing allocated shared memory that may be accessed simultaneously by other CPUs.

3.6.2 Wall Clock

As described previously, the wall clock mechanism is a synchronized gang-sceheduled interrupt for all the system CPUs. One processor is designated as the wall clock master that is tied to a hardware timer. It interrupts the other CPUs at the rate set by each individual processor. *Kali* requires that the wall clock have one millisecond resolution, so the master CPU requires a one millisecond timer interrupt rate. However the other CPUs need only be interrupted at the rate of their synchronous processes which may be many multiples of one millisecond. Hence, the master CPU should be designated to a processor having a synchronous process rate close to one millisecond to avoid unnecessarily high overhead in a CPU with a slower synchronous process. Futhermore, to ensure that wall clock interrupts are received and processed within their designated

periodic intervals, a simple 'interrupt-followed-by-acknowledge' protocol is used (see figure 3.8 below).

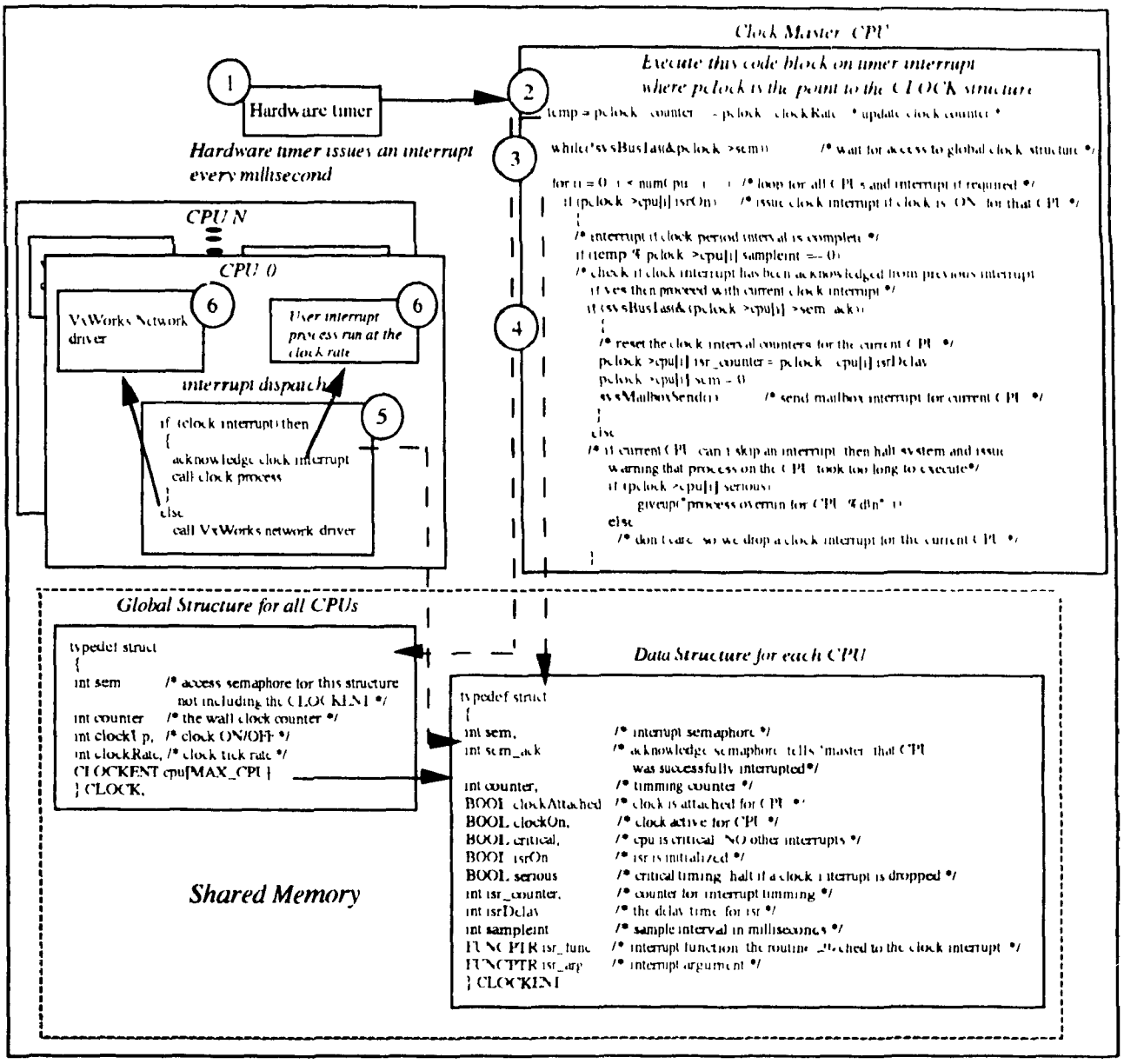


Fig 3.8 Kali Wall Clock Mechanism

The principal wall clock functions are as follows:

```
void control(func, serious, periodicity)
    FUNCPTR func,      /* pointer to a function which becomes the periodic process */
    Bool serious,      /* do not allow an interrupt to be 'skipped'
    int periodicity;   /* number of milliseconds for process period */
```

This is the 'core' wall clock function providing the user the means of assigning a function as a periodic process. Since it may be necessary for the user (from the user process running on the user CPU) to change the rate of a process on another CPU, the following function is provided (this is also useful in debugging and single-stepping periodic functions):

```
int set_control_rate(process, rate)
   int process,     /* CPU # indicating the periodic background process */
   int periodicity, /* the new number of milliseconds for the process period */
```

Returns '0' for successful completion or '-1' in case no process exists (or it is a undefined, *ie.*, '-1') number. Of course, rudimentary stop/start wall clock functions (halts/starts all periodic processes in the system) and wall clock time set functions are also furnished:

```
void start_kali_clock()

void stop_kali_clock()

Real set_kali_clock(time)
   Real time,
```

Finally, a means is provided the user to delete or release a function from being executed periodically:

```
void release(message)
char *message, /* message string to print, usually NULL */
```

The wall clock functions are essentially an extension of the original control and release functions found in RCCL [Hayward86].

3.7 System Hardware

The CPU hardware consists of five MC68020/68881 (0.12 MFLOPS) processor boards connected on a VMEbus with a 1 Mbyte shared memory board, an ethernet controller board, the robot I/O feedback interface board, and an 8 channel 12-bit resolution digital-to-analog converter (DAC) board to 'drive' the amplifiers, since this circuit would not easily fit onto the same board as the I/O (see figure 2.12 for examples of robot controller hardware). The VSBbus (secondary on the VMEbus) was initially connected, but not used because of 'test-and-set' bug in the CPU computer

board (see Appendix A). A PUMA 560 robot is used as the manipulator (section 2.7 describes the hardware relative to other robot control systems).

3.7.1 McGill I/O Board

Since at the time no commercially available encoder interface board existed that provided for six joints, index pulses, potentiometer interface (for calibration), and some digital control logic for miscellaneous functions like amplifier on/off control, it was necessary to create a board that provided all this functionality for basic robot I/O (some recent boards though do provide this in addition to DACs for amplifier control [Olsen89]).

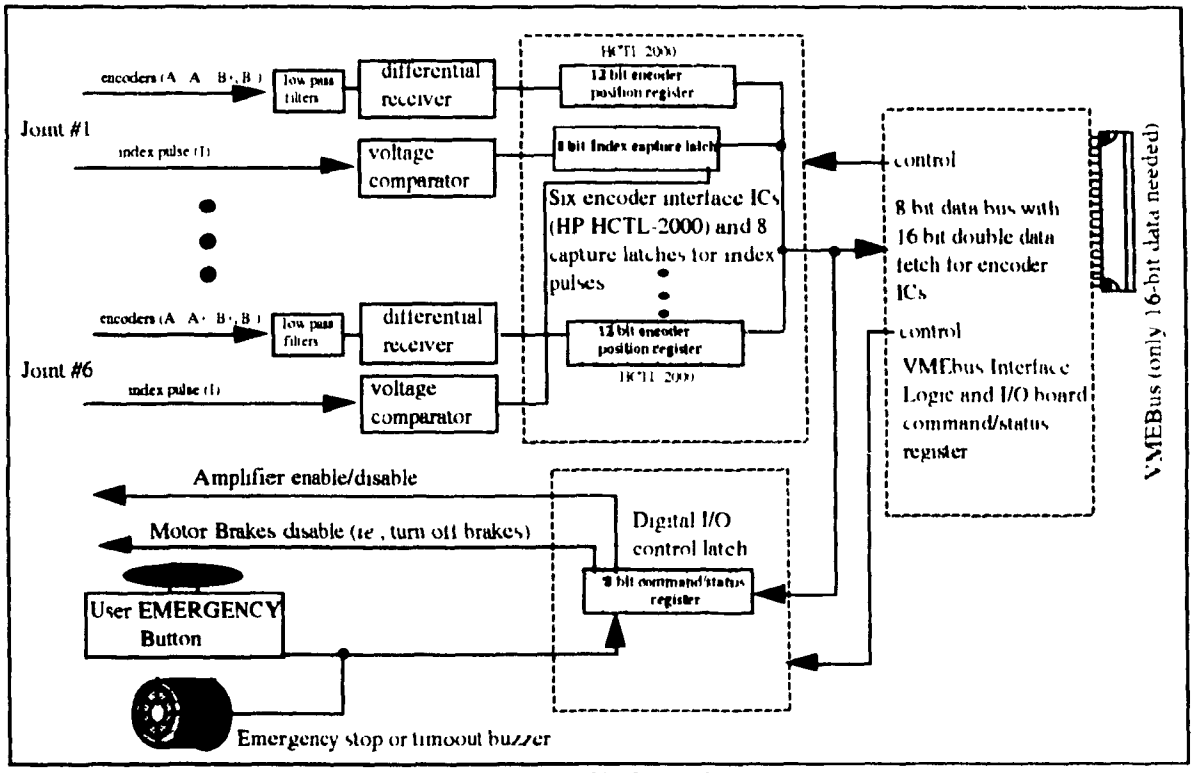


Fig. 3.9 McGill Robot I/O Board Overview

The McGill robot I/O board (figure 3.9 above gives a brief overview) is a 16-bit VMEbus board so processors can read joint position from the robot encoder's, read joint potentiometer values, control the motor amplifiers, and the motor brakes. The board also has a watchdog timer, so that in the event of a system failure (*ie.*, the software doesn't reset the watchdog

and it 'times out') the amplifiers are turned off and then the brakes are engaged. Note that the brake solenoids require ~50 msec to engage, whereas amplifier disables in less than 1 μsec, so the motor is off when the brakes actually stop the robot momentum. (this can also damage the brakes, but when a system crash occurs elegant recovery is not possible). The board also provides a user emergency stop input so that the user can halt the robot in an emergency (using an emergency stop button box). See section 2.7.1 for more details and Appendix C for the complete explanation and design of the McGill robot I/O board.

3.7.2 Motor Drive

PWM amplifiers are selected for the motor drive since they provide high power to weight ratio and cost less than linear amplifiers.

The worst-case (*ie.*, smallest) motor inductance of the PUMA 560 motors is 1.6 mH (see table 3.1 below), thus the amplifier must have form factor of near unity for this value of inductive load. A DC bus voltage for the amplifier is need such that a maximum current can be delivered to the motor while not exceeding breakdown thresholds. For the PUMA motors maximum voltage at peak torque is 14.5 V. The standard Unimation controller for this robot uses a 40 V bus which can deliver the maximum rated voltage at under 50% duty cycle.

The worst-case electrical time constant for the PUMA motors is 0.37 millisecond (L/R= (2.3 - 30%)mH / (3.9 + 12.5%)Ω) which is equivalent to a -3 db frequency of 400 Hz. Hence the PWM switching frequency must be over 4 kHz. The worst-case resonant mechanical frequency of the joint motor is approximately 200 Hz which must also be much smaller than the PWM switching frequency.

Comprehensive protection circuitry is desirable to ensure both motor and amplifier integrity. Since the system is intended primarily for experimental and research use overloads are likely to occur. Six Contraves CSR NC600 PWM amplifier boards mount in a power rack are used. They provide 5 kHz operation and 12.5 A RMS current capability per channel (they PUMA 560 requires only a total of 18 A RMS for all joints).

The amplifiers are controlled by ±10 V input signal proportional to the output current (*eg.*, 2.5 V input per amp output, in this case). The 12-bit DAC board provides 4096 increments between -10 V and 10 V, coupled with a 2.5V/A amplifier gain and 3.72 A max. current for the small PUMA motors, this yields less than twelve bits of resolution for peak current demand and a little less than eleven bits for RMS current (RMS ≈ half of peak, about 1900 increments over ±10 V range); however no signal conditioning is provided for the amplifier input. The large PUMA motors, having twice the current capacity make almost full use of the 12-bit DAC range.

Table 3.1 PUMA 560 Motor and Contraves Amplifier Parameters

	Small Motors (joints 4 through 6)	Large Motors (joints 1 through 3)
Peak torque	0.34 NM (48 oz-in)	2.12 NM (300 oz-in)
Max Continuous torque	0.17 NM (24 oz-in)	1.06 NM (150 oz-in)
Max voltage @ peak torque	14.5 V	13 V
Max current @ peak torque	3.72 A	8 A
Back EMF	0.091 V/rad/sec	0.261 V/rad/sec
Armature resistance	3.9 Ω (±12.5%)	1.6 Ω (±12.5%)
Armature Inductance	2.3 mH (±30%)	2.6 mH (±30%)
Max Winding Temp	180°C	180°C
Temp coefficient	3.5°C/W	1.6°C/W

Amplifier parameters (same for large and small motors)

Output current RMS	±12.5A
Output current peak	±25A
Form factor	1.01
(@ continuous current and min. induction)	
Bandwidth of current loop (@ RMS current)	0—500 Hz
Gain	2.5 A/V
Switching frequency	5 kHz

4. Conclusion

With aching hands and bleeding feet
We dig and heap, lay stone on stone;
We bear the burden and the heat
Of the long day, and wish 'twere done.
Not till the hours of light return,
All we have built do we discern.
—Mathew Arnold (1822-1888)

4.1 Summary

The basis for computing architectures and real-time computing practices as applied to robot controllers has been presented as well as an implementation of the *Kali* system.

Today, a 'competitive' robot controller can be only established through diligent investigation of user needs, comprehensive problem analysis, proper design and thorough testing. Without the awareness, persistence and foresight to recognize and take advantage of state-of-the-art technology that maximizes the advantage to both system builder and user alike, a robot controller project is, at best, doomed to obscurity. This is equally true of research projects as of commercial ventures.

Unlike mass market applications which attempt to trade-off functionality at the expense of programmability, in research robotics the goal is to minimize development time of new algorithms and ideas at the expense of higher controller hardware cost. Whereas the hardware costs exceed that of designs suitable for mass market production, low cost systems require considerably longer time to design and build, during which the original design usually becomes obsolete. Only a research team minimizing its engineering time by leveraging leading-edge technology while adhering tenaciously to 'open' standards can hope to succeed. The fundamental aim of all designers is to produce a 'good' design, yet it is rarely achieved without experience, and even then only if the designer sticks to his or her own area of expertise while building upon the know-how of others.

4.2 Lessons Learned, The Hard Way

This thesis is not the prophetic culmination of all previous works, but rather it seeks to distill from former endeavors that which is useful and discard that which is worthless From the record of past failures and successes, one concludes the following necessarily simple, yet profound principles that are quite self-evident. It is not a self-indulgent philosophy of design, but practical advice learned the hard way from direct experience and that of others:

- *Stop trying to out do the major hardware manufacturers.* Commercial processors from big companies like Intel or Motorola become more powerful quicker than any hardware researchers can build on their own. Current computational hardware is more than adequate for robot motion control, and multiprocessor systems are capable of much more demanding tasks like multisensor fusion. Since hardware performance is doubling approximately every year, software short cuts to meet performance requirements on current hardware are unnecessary—next year's processor will make up for any shortfall today.
- *Stop trying to out do the commercial real-time software developers.* Commercial real-time kernels are now better supported, more reliable and have higher performance than researchers can build on their own.
- *Stop trying to invent new robot languages.* Because of the complex hierarchy of models and abstractions needed, it is better to decompose the problem for a particular application and implement that solution in a conventional computer programming language.
- *Ensure Portability.* To protect the large investment in human resources necessary to create a complex real-time control system, every provision must be made to ensure compatibility with changing hardware platforms and software paradigms. Needless to say this includes processor and bus

architectures, as well as multiprocessing scheduling and communication algorithms and programming languages.

- *Use conventional, off-the-shelf hardware and software as much as possible.* Be it software or hardware, it is no longer feasible to create even a fraction of such a system's components 'in-house.' The reader is cautioned to take advantage of what is available and to leverage it so as to minimize development time. Compatibility with the designer's favorite tools (those with which he or she is most proficient) should be considered as a critical system requirement. The most useful software environments and development tools often appear first on the most common commercial systems. Researchers should develop those parts of a robot control system that they are best at and obtain the other parts either from commercial sources or from other researchers.
- *Concentrate on system flexibility and programmability for quick and easy experimentation.* Because of the research nature of such projects, the hardware and software must allow for easy changes and experimentation during development and use. The maximum ease and range of possible changes and future enhancements to both hardware and software during development as well as run-time is well advised. Robot control algorithms that are new, not well understood and hence very experimental will be tested.
- *Remember Maintenance:* Often forgotten, but increasingly of critical concern in complex software projects. The software must be designed as modular and straightforward as possible, so as to make future enhancements and changes as simple as possible. One starts by getting a core function up and running correctly first, so that the system can be tested and the design verified rather than attempting to include many initial 'bells and whistles'. If the designer is careful to provide 'hooks' in the system, these can be added

as and when they are necessary. The best designs usually take this maintenance-oriented approach.

- *Emphasize system integration.* The biggest stumbling block in robotics systems are often the endless hardware and system dependencies that must be explicitly programmed. They must be minimized, otherwise the researcher is condemned to a morass of annoying detail.
- *Don't forgo Reliability.* A predictable, reliable system is a necessity for experimentation and portability. Popular processor/bus architectures and software development environments are the most flexible and reliable (the two are not always opposing constraints).
- *Uphold Design Integrity.* As the last and perhaps most important culmination in the building of robot controllers, one concludes that whatever approach or design philosophy one chooses, it is unquestionably better to have a system reflect one set of cohesive ideas, omitting superfluous and anomalous features, than to select many worthy though independent and uncoordinated ideas which are thrust together Experience shows, beyond doubt, that it is necessary to adopt, uphold, indeed embrace, that elusive balance in applying the elements and rules of theory to produce a system as unified in concept as in design— that down to the last detail, the single most important principle in system building is design integrity [Brooks75, Brooks87, Lawson90]. Failure to uphold it inevitably results in a 'hack' or 'kludge' and a failed system. Too often designers implement complex and ornate systems in the mistaken belief that the more intricate or 'clever' it is, the more useful it will be. The exact opposite is true.

Appendix A. Buses & CPUs

A.1 Buses

Since processors and buses form a critical component for robot controllers, a detail comparison of currently available hardware is offered. We start with a comparison of backplanes.

Table A.1 A Comparison of 32-bit/64-bit Buses

	VMEbus	FutureBus	EISA	Multibus II	NuBus	SBUS
Standard	IEEE P1014/ Draft 1.2 VMI/64 (64 bit version doubles performance)	IEEE P896.1 and FutureBus+ P896.2 (64 bit version)	Extended Industry Standard Architecture Specification extension to PC AT bus	IEEE P1296	IEEE P1196/Draft 1.3	SUN Microsystems Inc SBUS Spec rev A.2 32 bit and rev B.0 64 bit
Bus Protocol	Asynchronous	Asynchronous packet switched	Synchronous 8.33 MHz clock	Synchronous 10 MHz clock	Synchronous 10 MHz clock	Synchronous 16/25 MHz clock
Primary data path	32 bit non multiplexed	32 bit multiplexed	16 bit PC AT bus non mux	32 bit multiplexed	32 bit multiplexed	32 bit non mux 64 bit mux
Second data path	32 16 8 bits with 16 bit justification	32 128 256 bit non justified	32 16 8 bit fast mode host translates justification	32 16 8 bit 16 bit justified	32 16 8 bit non justified	32 16 8 bit 32 bit justified
Address space	2^{24} (P1 only) 2^{32} (P1 P2)	2^{32}	2^{24} (PC/AT) 2^{32} (EISA)	2^{32}	2^{32}	2^{28} (2^{32} virtual address)
Interrupts	7 lines virtual interrupts via board specific mailboxes	Via message interrupts no dedicated interrupt lines	11 lines dedicated on bus, 4 for host 15 total	Fully virtual interrupts no dedicated interrupt lines	Virtual interrupts no dedicated interrupt lines	Dedicated interrupt lines synchronous
Bus Expansion	21 slots masters/slaves VSB secondary bus on P2	21 slots (19 in rack) master/slaves	16 slots masters/slaves	19 slots masters/slaves 1 slot arbiter	16 slots masters/slaves	8 slots masters/slaves typical SUN has only 4 slots
Performance	Continuously variable	Continuously variable	Quantized 120 ns wait states	Quantized 100 ns wait states	Quantized 100 ns wait states	Quantized 62.5 ns wait states or 40 ns wait states (25 MHz)
Peak Bandwidth	40 Mbytes/sec	120 Mbytes/sec (boards shipping)	33 Mbytes/sec (burst DMA block)	40 Mbytes/sec (burst)	37.5 Mbytes/sec (16 cycles block transfer)	80 Mbytes/sec (burst DMA) 146 Mbytes/sec (rev B.0)
Single transfer, 4/T	ransfer (avg delay=1/3 backplane length, double figure for 64-bit version) (Mbytes/sec)					
T_{mem} = 0 ns	25.0	37.0	16.7 †	20.0 †	20.0 †	50.0 † (25 Mhz)
T_{mem} = 50 ns	19.0	25.3	16.7 †	20.0 †	20.0 †	33.3 † (25 MHz)
T_{mem} = 100 ns	15.4	19.2	11.1 †	13.3 †	13.3 †	16.7 † (25 MHz)
Burst transfer, 4/T	ransfer (handshaken, infinite block length, avg delay=1/3 backplane length, double figure for 64-bit version) (Mbytes/sec)					
T_{mem} = 0 ns	27.9	95.2	33.3 †	40.0 †	40.0 †	80.0 † (25 MHz)
T_{mem} = 50 ns	20.7	43.5	16.7 †	20.0 †	20.0 †	50.0 † (25 MHz)
T_{mem} = 100 ns	15.4	28.2	16.7 †	20.0 †	20.0 †	25.0 † (25 MHz)
Ultimate future performance	~90	~500	33.3 †	40 †	40 †	~250 †
Basic limitation	Bus driving, problem logic delays prop delay	Bus driving, problem logic delays prop delay	120 ns between signal sampling due to clock	100 ns between signal sampling due to clock	100 ns between signal sampling due to clock	40 ns between signal sampling due to clock
Arbitration	4 level daisy chain round robin & priority	Fully distributed asynchronous	Centralized arbiter synchronous central clock	Distributed synchronous central clock	Distributed synchronous central clock	Centralized arbiter synchronous central clock
Algorithms	ROR or RWD ‡	Fairness, Priority	Fairness	Fairness, Priority	Fairness	Fairness
T_{arb}	200-500 ns	250 ns	240-480 † ns	300 † ns	200 † ns	80 † ns (25 MHz)
T_h (hold time for burst)	Selectable, 3 µs typ	Unconstrained	64 $T_{transfer}$	32 $T_{transfer}$	16 $T_{transfer}$	64 $T_{transfer}$
T_{get} (priority)	T_{arb} + 2T_b	T_{arb} + T_b	T_{arb} + (N-1) T_b	T_{arb} + 2T_b	T_{arb} + (N-1) T_b	T_{arb} + (N-1) T_b
T_{get} (fairness)	T_{arb} + (N-1) T_b	T_{arb} + (N-1) T_b	T_{arb} + (N-1) T_b	(N-1) T_b	T_{arb} + (N-1) T_b	T_{arb} + (N-1) T_b
Total time for T_{arb} +	16 transfers with 100ns slave memory					
T_{get} (best)	8.3 µs	2.6 µs	57.8 µs	6.7	48.2 µs	28.9 µs
T_{get} (worst)	54.3 µs	43.4 µs	57.8 µs	3.	48.2 µs	28.9 µs
Message Passing	Not supported	Fully supported also read/write to multiple slaves	Not supported	Fully supported also read/write to multiple slaves	Not supported	Not supported
DMA control	Burst mode master/slave protocol	Burst mode master/slave protocol	Central DMA controller	Burst mode master/slave protocol	Burst mode master/slave protocol	Central DMA controller MMU for address translation requires extra bus cycle
Bus Interface	TTL mixture of 48 & 64 mA 3 state & open collector	BTL (backplane transceiver logic) 80 mA	TTL mixture of 24 & 5 mA 3 state & open collector	TTL mixture of 48 & 64 mA 3 state & open collector	TTL mixture of 48 & 64 mA 3 state & open collector	TTL or CMOS levels mixture of 3-state & open collector
silicon	Multiple sources	National/AMD transceivers variety planned	Intel EISA chip set including bus master	Intel MPC and MPI	TI NuBus transceivers and master controller	LSI logic SBUS master and slave controller

† Ignores clock latency

‡ ROR is Release On Request which releases the bus for new arbitration only if another master requests it. RWD, Release When Done releases the bus for new arbitration after every access. Fairness indicates each master is allocated equal time on the bus (not counting DMA). Priority means one master may supersede another for bus access. Information sources Borrill85, Warren90, Whang90, Sha91, SUN89, IEEE87, IEEE88, IEEE/ANSI87, IEEE/ANSI89

In multiprocessor systems, bus bandwidth is critical As one can see from the above chart memory access time and arbitration are the two key critical factors, *ie.*, bus cycle = T_{arb} + T_{mem} For VMEbus, for example, moving from 100 nsec memory to around 10 nsec saves 10 Mbytes/sec on bandwidth (the arbitration overhead is always the limiting factor) So for systems involving a high degree of contention, the fastest possible memory is critical Of course, selecting a bus also depends on price and availability of I/O boards. VMEbus offers the greatest variety of industrial I/O, but EISA is growing in popularity and almost the same performance with the added bonus of being platformed on low cost PCs. MultiBus II and FutureBus have fast message passing hardware, but at a high cost. SBUS, on the other hand, has tremendous speed, but limited slots and board selection

Another factor in the efficiency of interprocessor communication is the mechanism by which processors notify each other that new information is available If a processor is awaiting some data from another processor, it should not have to repeatedly disrupt its other activities when polling a semaphore or flag to determine if the data is available, unless of course it is not performing other overlapping tasks and the semaphore 'lock-out' time is far less than the interrupt processing time A point to point interrupt mechanism can be used to increase efficiency On the VMEbus the dedicated interrupt lines quickly run out, so 'mailbox' addresses are used (an interrupt is generated on a processor whenever a certain on-board dual ported address is accessed To avoid conflict each CPU board has its own hardware settable mailbox trigger addresses). Other buses, such as MultiBus II and FutureBus have sophisticated message passing coprocessors which can act as virtual interrupt controllers (the message itself selects the interrupt handler). Due to the performance degradation caused by memory contention, tightly coupled designs are generally better suited to a small number of processors.

A.1.2 VMEbus/VSBbus

Since most researchers intend on using the VMEbus, a few useful notes are offered:

- Do not use the customary arbiter provided by CPU board manufacturers, they do not always work, provide limited time-out facility and, in some cases, even slow down CPU processing. Use a separate arbiter in slot 0.
- Do not use priority bus access to try and prioritize system events, these and other events can quickly 'hog' the bus preventing other boards and/or events from getting on the bus. Use the round robin fairness arbitration with all boards at the same priority.
- Do not use the ROR arbitration mode as it requires much longer to re-arbitrate subsequent bus accesses. The idea in ROR is for long DMA blocks so the same bus master need not wait for arbitration on every access. For many CPUs contending for memory RWD is the better choice.
- Make sure the bus power supply is large enough to handle the anticipated load, most of the newer CPU boards require power from both P1 and P2 connectors and consume as much as 12A @ 5 V.
- VME requires termination on both ends of the bus, whereas the VSB (VME Secondary Bus) requires termination on ONE end only. The VSB bus [Newton89] provides a second 32-bit bus on the spare pins of the VME P2 connector. This bus can have up to 6 slots and is designed as a high speed memory or peripheral bus. It supports multiple masters, slaves, and has performance equivalent to the VMEbus. The VSB is a good way to alleviate memory contention on the main VMEbus, *ie.*, one can have network traffic I/O proceed on the VME while simultaneously CPUs communicated through shared memory on the VSB. Most VME CPU boards support VSB accesses along with an on-board bus arbiter.
- A note on CPU boards, the test-and-set mechanism is the sole hardware means for synchronization and mutual exclusion (*ie.*, semaphores) on the VME and VSB buses. It is critical that the CPU boards used in a multiprocessor configuration support the test-and-set feature properly. This

was a great problem with Heurikon V2F CPU board used in *Kali* (fortunately a fix was possible by using the bus lock line to block out all other masters while the test-and-set was executed, this fix though requires a substantially longer time for this semaphore operation).

A.2 Commercial CPUs

Since the a robot controller's processor so affects the performance, availability of software and hardware tools, and third party support, its selection is the most critical hardware decision when building a robot controller, because most of the effort in a robot controller design today is in the software. To that end, the following is a brief summary of current commercial CPUs, their performance, future versions, and availability of developments tools and real-time kernels.

A.2.1 Motorola

The worn out 68020 and its microprogramed floating point coprocessor the 68881 and the newer 68030/68882 once the leading edge of performance (still used in the majority of multiprocessor robot controllers) are now being greatly surpassed. The main problem is a dismal floating point performance since each operation requires a sequence of micro-instructions to execute. The best case register to register multiply (single precision) requires 2.8 μsec @ 25 MHz and 1.4 μsec @ 50 MHz (the maximum for the technology), well below 1 MFLOP [Motorola87]. In real programs the situation is even worse since fetching operands from main memory is constant. When both source operands and the destination are resident in memory over 200 cycles are required to complete the operation, only a third which actually do the arithmetic! On the other hand, the newest addition to the family, the 68040, does manage some impressive performance. A completely redesign chip, it incorporates a low instruction cycle integer core and executes basic moves, jumps etc. in a few cycles like a RISC processor, but uses more cycles for complex indirect addressing and other less frequently used instructions. For this CPU floating point performance is considerably enhanced, since the coprocessor interface is removed and the high speed FPU is on-chip. At 25

MHz the 68040 performs single precision adds and multiplies in a noteworthy 3-5 cycles (memory to register) reaching about 2-3 MFLOPS in practice [Edenfield90, Motorola90] Needless to say the MC680x0 family of processors are the most popular among VMEbus vendors both in CPU boards, development tools, and real-time kernels (almost every real-time kernel has been ported to, if not down-right developed on, this processor family)

The 88000, Motorola's original all RISC chip was designed to supplant the 68020 architecture, but achieves little more than the 68040 (to redeem itself from the embarrassment Motorola is designing a newer much faster version of the 88000, the 88110). The 88000 floating point unit uses heavy pipelining (a five stage pipe with one multiply per stage) to reduce operations to five cycles [Motorola88] and the adder is a three stage pipeline. At worst case an operation requires a few cycles for memory access, 3-5 for the operation, and another few to output to memory. Whereas the theoretical maximum, for the chip @ 20 MHz, is 20 MFLOPS (assuming a full pipeline and a single step for each operation), the practical performance is around 2-3 MFLOPS. Unfortunately, the 88000 does not enjoy the popularity of its sibling, thus the availability of boards, tools, and real-time kernels have been somewhat less than that for the 680x0, but the 8800 still has as much if not more in the way of third party support than processor families from rival companies (except the Intel 80x86 family).

A.2.2 Intel

Ironically, the 80x86 architecture almost bankrupted the company until the triumph of the IBM PC. A host of processors are now offered, the CISC-based 80386 with 80387 FPU and the 80486 (on-chip FPU), the RISC 80960KB, and the newest host performer, the lightening fast 64-bit i860 The 386/387 pair is a poor performer (0.5 MFLOP @ 33 MHz), the 386/WEITEK(3167) FPU does better at around 1.3 MFLOPS, and the 486 with or without the WEITEK(4167) FPU does about the same [Intel87, Intel90]. The great advantage of this architecture is the huge availability of software and development systems. The i960 family of CPUs, targeted at embedded applications, offers around the same floating point performance of the 386 family, but with faster integer

instruction processing. The newest members of this family, i960CA, utilizes a superscalar approach [Intel88] capable of 66 native MIPS (peak @ 33 MHz). A new version of this processor is to include an FPU with 27 MFLOPS single precision LINPACK performance @ 40 MHz The great potential for robotics, however, lies in the i860 and its descendants. With a 64-bit data bus, three stage floating point multiplier and parallel adder, this processor can achieve 80 MFLOPS peak when all pipelines are filled [Kohn89] Single operations require 2 cycles, so a 20 MFLOP rate for the current 40 MHz chip is sustainable With current compilers, however performance is around 10 MFLOPS Better compilers and faster chip versions will greatly improve performance

A.2.3 SPARC

Developed by SUN Microsystems in response to Motorola's slowness in bring out the 88000, the SPARC architecture is now the most widely used RISC chip [Glass91] Initially without an integer or floating point multiply/divide instruction and suffering from lack of optimized silicon [SUN87], the improved instruction set and low cost licensing available now from SUN, has given rise to many different implementations from a host of companies are available: Fujitsu was the first to make SPARC CPUs, initially as a gate array now in optimized silicon, including everything from the SPARCLite chip set aimed at embedded systems to 33 MHz CPUs, Cypress Semiconductor has available 40 MHz CPUs & FPUs as well as a familly of support chips; Solbourne/Matsushita were the first to integrate a complete CPU, MMU, and FPU on one chip, LSI logic has a whole line of SPARC CPUs and chip sets as well as ASIC standard-cells, finally, TI and WEITEK offer the fastest FPU implementations for SPARC [Birman90, Darley90]. In addition, TI, Cypress, and LSI are also racing ahead to develop next-generation super-scalar SPARCs and 64-bit versions (SUN's SPARCstation 3 will be a 64-bit design). Performance of the manufacturers (at the same CPU frequency) vary moderately and it is somewhat acknowledged that the SPARC architecture is a little slower in integer and floating-point performance than the MIPS R3000 CPUs, but better than Motorola's 88000. The 25 MHz WEITEK SPARC FPU does about 2.75 MFLOPS (SPARCstation 1+) and the TI 40 MHz implementation does about 4.5 MFLOPS (SPARCstation 2) Future versions

will improve performance dramatically especially as 64-bit versions arrive (this is necessary as the IBM RS/6000 workstation and the new HP/Apollo PA-RISC achieve much higher floating point performance from 32/64-bit data paths). For the SPARC architecture a number of real-time single board computers are available from a vareity of manufacturers, although limited in CPU speed. Most real-time O/S's are available, though some are still in beta test like VxWorks.

A.2.4 Mips

Start-up Mips, initially funded to build workstations, has come into its own as a RISC design house Its much touted R3000 CPU and R3010 FPU are among the fastest RISC CPUs available. With a strategy similar to SUN's SPARC, many semiconductor vendors implement Mips CPUs. LSI with 25, 33 and 40 MHz versions of both CPU and FPU as well as some specialized embedded versions, Integrated Device Technology (IDT) with CPU & FPU available separately and with integrated cache modules, Performance Semiconductor with the first integrated CPU & FPU on a single chip, Siemens, Toshiba and others are or have plans to manufacture the chips. The Mips CPUs are used in DECstation workstations, Silicon Graphics workstations, as well as some recently announced (for future delivery) by Compaq, Sony, Siemens, Zenith (Bull), and others (the ACE Advance Computing Environment, consortium). Performance is excellent with the 25 MHz version achieving about 25 MIPS (one instruction per cycle) and 4.9 MFLOPS (versus SPARC 18 MIPS and 2.75 MFLOPS) [LSI88]. A better handcrafted chip design and superior compiler technology with more efficient use of a more conventional register file (as compared to the SPARC's after thought compiler and large number of register windows which suffer from a high reloading penalty when the windows are exhausted) give this CPU family a slight performance edge [Kane88]. The intense debate between SPARC and Mips continues on which is the most efficient for various instructions, *ie.*, compare and jump, square root, etc [Williams91]. The one definite Achilles heel of MIPS processor family, though, is the lack of software and development tools as compared to the SPARC as a workstation CPU and to the AMD29000 family as an embedded processor. Few single board computer implementations are available (such as a VMEbus board) and the better real-time

O/S's are generally not available for this CPU family, in particular VxWorks (though LynxOS has been ported). The newest version of the family the R4000, transitions from 32 to 64 bits and features superpipelining, which allows the computer to issue two instructions per clock cycle It includes Integer and floating point units, an 8 kByte data cache, cache control, a memory management unit and full multiprocessing capabilities. Initial speed should be over 60 MHz with about 60+ MIPS integer and 20+ MFLOPS floating point performance.

A.2.5 Advanced Micro Devices (AMD)

The high performance 29000 processor family sold by AMD was among the first to implement true a Harvard architecture (2 data/address buses off-chip, one for data, one for instructions). Originally offered as a workstation CPU, the 29000 was withdrawn from the 'race' and is now used as in a variety of embedded applications, especially in floating point intensive areas like graphics where it performs very well The 29000 CPU uses the 29027 FPU chip for floating point operations, but unlike most FPUs this chip provides a pipelined mode and a lower latency mode in which the arithmetic logic unit is totally combinatorial This offers far lower latency than found in other more conventional serial-parallel (interlocked multi-stage pipelined) FPUs In flow-through mode the 29027 performs 32-bit floating point adds/multiplies in 5-6 cycles, about 200 nsec @ 25 MHz [AMD89], in pipelined mode an operation requires 4 cycles (160 nsec @ 25 MHz) This results in 5-6 MFLOPS peak performance. The newest member of the family, the 29050 offers an integrated FPU with much enhanced capability. A three stage adder/multiplier pipeline that performs operations in one cycle as fast as 25 nsec (@ 40 MHz)! 4x4 matrix multiplies in 0 9 µsec with a sustained 40 MFLOPS [AMD90] throughput for this operation—great for robotics

Availability of design and development tools is excellent, though the full feature real-time O/S's like VxWorks have generally not been ported to this processor family.

A.2.6 DSPs

Digital signal processors were heralded as a new age in computing for many application areas, these processors are designed to perform digital filtering at very high speeds, especially multiply-accumulates for dot product and matrix multiplication (all implement overlapping multiply/adder pipelines). The newer DSPs are all floating point now (however some sacrifice IEEE floating point compatibility to gain speed) and most have full C compilers making them eminently suitable for high speed robotics applications. Performance varies between the many DSP families of which there are a host from many different vendors [Andrews90, Lee90]. The most prominent ones are:

- AT&T DSP32C: 25 MFLOPS (peak @ 50 MHz), 3x3 by 3x1 matrix multiply in 36 cycles=720 nsec, 32-bit floating point format conversion to/from IEEE single precision in hardware [Fuccio88];
- Motorola MC96002: 18 MFLOPS (peak @ 27 MHz), 3x3 by 3x1 matrix multiply in 24 cycles=890 nsec, full IEEE single and double precision floating point operation using transparent conversion on load/store to memory (interstingly, this DSP uses a 96-bit internal floating point format which has the same latency for single or double precision arithmetic operations) [Sophie88];
- TI TMS32C30. 33 MFLOPS (peak @ 33 MHz), 3x3 by 3x1 matrix multiply in 29 cycles= 879 nsec, 32-bit floating point (non-IEEE compatible, but with conversion in software) [Papamichalis88, TI89]. TMS32C40: 40 MHz version of the 'C30 20% faster, has six symmetric multiprocessor communication ports (used for large multiprocessor arrays), and has hardware conversion to/from IEEE single precision format [TI91], unlike its predecessor. This family of DSPs is far and away the most popular

with the best support both from TI and third party hardware/software vendors.

In general, development tools and systems for DSPs are few and remain confined to the chip manufacturers themselves. Real-time O/S's and kernels are few and far between, the best system is 'SPOX' [Manuel88] developed initially for the TI family and being ported to the others. A number of other 'block-mode' graphical programming systems aimed at creating signal processing applications seem to be the direction DSP software is taking, unfortunately unsuitable for general control problems like Cartesian-based robotics control (except for single joint controllers). However, reseachers could develop such systems for robotics.

A.2.7 Transputers

From Europe Inmos' (now a division of Thompson-CSF) much touted idea is the Transputer. Following the philosophy of Occam's Razor ("keep it simple, stupid"), the Transputer design involves the connection of many small but powerful microprocessors joined by high speed (30 Mbits/sec) serial links. Based on a very simple three register architecture coupled with a RISC like instruction set, the T800 with integrated floating point unit performs only about 1.5 MFLOPS @ 25 MHz. Later versions like the T805 make some moderate throughput improvements and attain higher clock frequencies. However, the latest Transputer, the H1, is a pipelined superscalar design capable of 60 MIPS and 10 MFLOPS sustained performance @ 50 MHz (200 MIPS, 25 MFLOPS peak performance) [Williams91(2)]. The unit includes 16 kBytes of on-board cache memory and a programmable multichannel serial data link crossbar.

Availability of development and software tools is adequate, though only one or two real-time O/S's or kernels, of any size or sophistication, have been ported to the Transputer.

A.2.8 National Semiconductor

Sluggish sales, no supported bus architecture, and only above average performance plague this company. The NS32332/32380 32-bit microprocessor and floating point coprocessor

performs at about the same level as the MC68030/68882. The newer NS32532 does have some impressive performance. 15 MIPS peak about 8-10 MIPS sustained (not quite as good as the MC68040 or the i80486). Acknowledging the dismal performance of its previous floating point coprocessors, National decided to use the WEITEK 3167 as the FPU for the 32532 (note that Intel already uses the next generation WEITEK4167 with the 'i486) which can achieve 1-2 MFLOPS. Typical add/multiply is 4 cyles (2 cycles instruction transfer from CPU instruction stream plus 2 cycles for the operation itself). A short 2 stage adder/multiplier pipeline allows the throughput to achieve operations in 2 cycles for a 15 MFLOPS peak rate on short bursts @ 30 MHz [Iacobovici88], though in practice only 1-2 MFLOPS is achieved by the 32532 (about the same as the WEITEK coprocessor achieves with other CPUs like the i486 or SPARC). In addition National's microprocessors are only supported by a small group of third party hardware and software vendors.

A.2.9 IBM

Having invented RISC and then having failed capitalize on it, IBM stunned the computing world in 1990 with its System/6000 workstation and RS/6000 CPU chip set. Then IBM, Apple Computer and Motorola announced that they were in negotiation (in secret since 1990) for Motorola to produce the RS6000 chip set, thus making it available to third party workstation and CPU board manufacturers. Performance is second only to the HP/Apollo PA-RISC with 55 MIPS (Specmark benchmark [Weicker90]) and 21 MFLOPS on the LINPACK (double precision for the latest 50 MHz CPU) [Oehler91]. The processor consists of a five chip superscalar RISC design. The main instruction decode & dispatch unit communicates to the FPU and IU (integer unit) over a 128-bit bus and decodes four instructions simultaneously. Data and instruction cache units provide 32K and 64K respectively of four-way set-associative caching, thus providing very high hit ratios (moreover, error correction (1 bit) and detection (2 bits) is provided on all interchip data and address paths). The FPU has a 64-bit data bus, and in addition to standard single cycle

multiply/add/divide overlapping execution units, special single cycle accumulate and multiply hardware is provided like in DSPs [Oehler90].

So far, the only operating system provided is IBM's version of UNIX, AIX, which has some, though limited, real-time capabilties.

Appendix B, Real-Time Kernels

B.1 O/S Classification

The world of real-time operating systems/executives and/or kernels is arcane, complex and diverse. Designers face a difficult, involved process of learning what real-time operating systems or kernels are available, whether they are creating simple pick-and-place robot controllers or complex object oriented, multi-sensor fusion systems. Selection of the O/S that best suits their application is difficult as more than fifty different products are commercially available. To aid the reader in this endeavor, this appendix attempts to classify and evaluate these products, beginning with a basic taxonomy:

Table B.1 Classification of Selected Real-Time O/S's

	Architecture Independent (not necessarily open architecture)	Proprietary
Embedded	VRTX (Ready Systems) SPOX (Spectron Microsystems, DSP kernel) C Executive° (JMI Consultants) USX (US Software) Precise/MPX (Harmony OS, Precise Software Tech) (many other kernels on a variety of 8,16 and 32-bit CPUs)	CX/RT (Harris) VME Exec (Motorola) iRMK (Intel) FlexOS (Digital Research) RTE (Hewlett-Packard) (other Vendor specific kernels)
Hybrid	pSOS (Software Components Group) MTOS/UX‡° (Industrial Programming Inc) PDOS‡ (Eyring Reasearch Inc) VxWorks‡° (Wind River Systems) OS-9, OS 9000 (Microware) UniFlex (Technical System Consultants) CHORUS! (Chorus Systems) VRTXvelocity+ (Ready Systems) (many other kernels, the fastests growing category)	FlexOS (Digital Research) iRMX (Intel) Versa DOS (other Vendor specific real-time UNIX variations)
Full O/S	LynxOS^ (Lynx Real-Time Systems Inc) REAL/IX+ (Modcomp Corp) QNX (Quantum Software Systems Ltd) CHORUS/MiX! (CHORUS Systems)	Venix+ (VentureCom Inc) ELN (DEC)

Embedded	• Traditional Embedded • No I/O for File System • Hard Real-time systems • Cross development • Targeted at OEM applications	Hybrid	• Embedded or Sophisticated (disk-based ~1Mbyte) • I/O and File System • Cross development usually self-hosting • Targeted at End-user, OEM or VAR
Full O/S	• Non-Embedded • Disk based, over 1 MByte code • I/O and File System • Soft real time • Self-hosting • Targeted at end-user or VAR		+UNIX derivative ^Independently created UNIX compatible free of AT&T source code ‡Approximate UNIX functionality (library and file I/O source level compatible) °Open Architecture, source code available

As mentioned previously, the wisest course of action in creating a robot controller is to divide the code into two parts: a) the real-time, system, dependent kernel, and b) the robot control code itself, which must, to some degree, depend on the particular real-time kernel selected. The real-time kernel, therefore, must also meet all the general requirements of the robot control code, specifically:

- It should be easily portable to other systems, despite dependence on the bus and CPU boards.
- It should contain all system dependent code, not including robot configuration information, though an effort to minimize this should be made.
- It should be as flexible as possible in terms of source code, networking and third party support
- It must perform well enough for the robot control task, not only on the development system, but equally on other hardware of similar capability Primary performance characteristics include, task switch time, interrupt latency, effective scheduling algorithms, intertask and interCPU communication mechanisms (*ie.*, shared memory with semaphores and/or message passing).
- It must have support from the vendor and continued improvements as well as porting to newer hardware as it becomes available

B.2 Evaluation & Commentary†

In order to provide the reader a useful comparison of the most popular commercially available real-time kernels, the following evaluation and commentary gleaned from the product

† Some of the following comments from the respective representatives and product literature from Wind River Systems, Ready Systems, Eyring Systems, Software Components Group, Modcomp Inc , Lynx Real Time Systems, and from [Williams90] Additional commentary provided by Sung Han of Commonwealth Edison, Illinois, and of course, the author

literature, personal experience of the author and that of others is offered (unfortunately for the reader, only five kernels are reviewed):

B.2.1 pSOS+

Summary: pSOS+ (Vendor Software Components Group, San Jose, California)

(on 16 MHz MC68020)
☆☆☆ Interrupt latency 10 µsec
☆☆ Task switch 250 µsec
☆☆ Message send/receive 340 µsec

☆☆☆ Priority-based preemptive scheduling, user programmable
☆☆☆ Fully dynamic process management and interprocessor communication with semaphores, including across CPUs
☆☆☆ Networking support including full BSD 4.3 sockets, NFS and RPC
☆☆ Small-sized ROMable target application modular component architecture
☆☆☆ Multiprocessor support
☆☆☆ Integrated development and debugging tools
☆☆☆ Widespread use and proven reliability
☆☆ Portability MC680x0, MC68332, i86, i80386, no source, though large number of such CPU boards are supported

pSOS+ is a solid, reliable system used used in a variety of aerospace, medical and communications products. Performance is excellent with fully integrated debugging and multiprocessor support, in addition to essential networking support.

pSOS+ is the follow-on to the venerable pSOS kernel, from Software Components Group. The system includes the pSOS+ kernel, pROBE+ board-level monitor/debugger, and the pNA+ Network manager. Also included is compiler package from Microtec, and XRAY+, which is a joint product by Microtec and Software Components Group.

The pSOS+ kernel is highly flexible, and its feature set is competitive with the others. However, the approach used to interface pSOS+ components to user code is different from the other systems. While the other systems link the user code with the system code to resolve references, all of kernel functions are called through software traps, which vector directly into the pSOS+ kernel. The kernel then determines which pSOS+ component to call to service the request. And while this provides position independence and eliminates the need to link user code with pSOS+ routines, it also creates some overhead, since every call to a pSOS+ system component means a service trap. This is reflected in the performance figures provided later (see section B.3). Note, however, that

this applies to the software trap kernel library, KJM (Kernel Jump Module) interface. An alternative interface, CIL (C Interface Library) providing direct linking, gives better performance.

The pSOS+ kernel's messaging facility allows up to four long words of user data to be passed between tasks. VRTX only allows a single word, which usually means that memory has to be allocated in order to send any meaningful data through messages (the length of VxWorks messages is unlimited). Also, event flags in pSOS+ can be directed either at a specific task, or be global. Events in VRTX are always global, and VxWorks does not have event flags. pSOS+ also allows each task to have an asynchronous signal handler. This signal facility is not entirely like UNIX signals however, as the signal handler will not be called until the user task makes a pSOS+ system call.

The pNA+ networking package provides full Berkeley 4.3 sockets support. The competing network packages provide similar functionality, but only pNA+ has a socket-sharing mechanism, which can be convenient if it is required to deal with numerous independent connections on a single board computer.

The XRAY+ source level debugger proved to be a versatile, reliable debugger. Its user interface was a bit awkward, however. The debugging format used is IEEE695; this format is fully supported by the accompanied Microtec Research compiler tools package (compatibility problems exist in using IEEE695 output from other compilers). The company also provides provides NFS† and RPC‡ support through their pNFS product which is available in conjunction with their pHILE+ file management system.

Software Components Group also offers pSOS+/M, which is a multiprocessor version of the pSOS+ kernel. It provides (nearly) seamless usage of standard pSOS+ calls over multiple processors connected through various types of media including backplane and ethernet.

† Network File System A standard for client/server file sharing on UNIX machines popularized by SUN Microsystems
‡ Remote Procedure Call A means of executing a function from one computer to another across a network. RPC is a necessary layer for NFS

B.2.2 VRTX Velocity

Summary: VRTX Velocity (Vendor Ready Systems, Sunnyvale, California)

(on 16 MHz MC68020)
☆☆☆ Interrupt latency 10 µsec
☆ Task switch 265 µsec (worst case 530 µsec)
☆☆ Message send/receive 375 µsec

☆☆☆ Priority-based preemptive scheduling, user programmable
☆☆☆ Fully dynamic process management and interprocessor communication with semaphores, including across CPUs
☆☆☆ Networking support including TCP/IP, RPC, and NFS
☆☆ Small-sized ROMable target application, modular architecture
☆☆☆ Multiprocessor support
☆☆☆ Integrated development and debugging tools
☆☆☆ Widespread use and proven reliability
☆☆ Portability MC680x0, no source, though large number of such CPU boards are supported PC/AT also provides development/target, but only for VRTX32 kernel, not Velocity extensions

VRTX Velocity is a sophisticated development environment, and provides an unsurpassed level of flexibility, automation and ease of use However, its complexity can be daunting at times, especially to the beginner Decent kernel though VRTX32 is, it falls short of pSOS+ in terms of performance and VxWorks in terms of features

VRTX Velocity is a combination of target software components and host-based tools. The target software evaluated consisted of the VRTX32 kernel, RTScope board-level monitor/debugger, TNX TCP/IP† Network manager, RTL runtime C library, and RTShell target shell. The host-based tools included the RTSource remote source debugger, Hyperlink ethernet downloader/command center, and an Oasys Compiler tools package. The host side of the Velocity package runs only on SUN3/SUN4 workstations in SUNView.

VRTX Velocity offers two approaches to development. One is where TFTP‡ is used to load the system software and RTshell at boot time using TFTP. The shell then can be used to load software onto the board off a network through NFS. An incremental linking loader is provided, which accepts standard Unix format relocatable object files. The shell allows functions to be called, and can evaluate most C expressions. The alternative approach is to manually download system

† Transmission Control Protocol/Internet Protocol A set of communication and addressing protocols used primarily by UNIX machines, the U S government, and by association most research institutions
‡ File Transport Protocol An O/S independent file transfer protocol using TCP/IP

software to the target board using Hyperlink, an intelligent downloading program. Hyperlink can also be used to launch other Velocity programs, such as RTScope and RTSource.

The VRTX32 kernel provides a unique approach to identifying tasks. Each task has a numeric ID, from 0 to 255. They cannot have names, which are allowed in pSOS+ and VxWorks. If more tasks are needed, there can be multiple tasks with an ID of zero, up to 16384 (signed 16-bit integer) of them. These tasks are referred to as 'anonymous' tasks, and they usually cannot be directly referenced by other tasks. For instance, they cannot be deleted by ID, since an ID of zero used in system calls usually refers to the caller. Fortunately, VRTX32 allows tasks to be referenced by priority groups, so this is one way to deal with anonymous tasks.

The VRTX kernel has some features which are lacking in the other kernels; specifically, it provides character I/O operations at the kernel level; and mailboxes, which are single slot message queues. It is also lacking in other areas—for instance, there are no built-in time/ calendar functions, aside from a tick counter (though complete timer functions can be easily built by the user, the same is true for VxWorks). Furthermore, when creating a new task, there is no easy way to pass parameters to it. Also, creation of tasks is always a single-step process, *ie.*, create the task, and if its priority is higher than the caller, pre-empt the caller and run the task. This is different from pSOS+, where task creation is a two-step process, *ie.*, create, then activate. VxWorks allows the use of either approach. Also, there is the odd restriction that once a message queue is created, it cannot be deleted. Ready Systems claims that this is to prevent fragmentation of system memory. Finally, unlike pSOS+ and VxWorks, VRTX32 lacks asynchronous signal handling.

The RTScope board level monitor/debugger is highly flexible, and provides two modes of operation: command mode and task mode. In command mode, all user tasks are halted, interrupts are disabled, and the standard suite of memory operations and task control commands can be used. In tasking mode, RTScope runs as a task, alongside user tasks, with interrupts enabled. The system can therefore be monitored in action without impeding on user tasks. Note that commands which infringe on normal system operations cannot be used in this mode, *eg.*, memory patching, task suspension, task creation, etc, are not available. Also, RTScope, unlike pROBE+, will not

automatically halt all tasks if a single task crashes (usually). The other user tasks will continue to run normally, if possible (although whether this is a safe practice may be questionable).

RTSource is a good, user-friendly source debugger It appears easier to use than XRAY+; however, reliability seemed to be inferior, as it gets more easily lost in the code RTSource also uses a proprietary debugging format, so the Oasys compiler tools have to be used.

Also available from Ready Systems is MPV, a multiprocessing version of the VRTX kernel, which is similar in approach to pSOS+/M. Also available is the IFX manager, along with NFS and RPC support. In addition, unique to Ready Systems is the availability of VRTX Designer, which is a CASE tool for real-time system design. It uses known VRTX timing data to project the performance of user designs and spot potential bottlenecks. These products were not evaluated.

VRTX Velocity provides automated scripts to build EPROMs, applications, or even 'makefiles'. The scripts are interactive, and the user chooses from a list what components he or she wants to include in the target. The appropriate 'makefile' is then created that builds both the VRTX system software and the user application.

B.2.3 VxWorks

Summary: VxWorks (Vendor Wind River Systems, Alemeda, California)

(on 16 MHz MC68020)

☆☆	Interrupt latency	9 µsec (worst case though 75 µsec)
☆	Task switch	200 µsec (worst case though 500 µsec)
☆☆	Message send/receive	375 µsec

☆☆☆ Priority-based preemptive scheduling, user programmable
☆☆ Fully dynamic process management and interprocessor communication with semaphores
☆☆☆ Networking support including TCP/IP, RPC, and NFS
☆☆ Small-sized ROMable target application (200 kBytes with minimal networking)
☆ NO Multiprocessor support
☆☆☆ Integrated development and debugging tools
☆☆ Widespread use and proven reliability
☆☆☆ Portability MC680x0, SPARC beta test, source available, large number of CPU boards are supported

VxWorks enjoys a good reputation among its customers Its networking support is second to none, and it has a flexible kernel with strong interprocessor communication facilities However its remote source level debugging facilities are not as sophisticated as the competition In addition, VxWorks is conspicuously void in the area of a multiprocessor kernel, which others provide

The current version of VxWorks is 5.0.1a (as of 1Q 1991), from Wind River Systems. It includes the WIND kernel, a target shell, a TCP/IP networking manager, an extensive C runtime library, and the GNU[+] compiler toolkit. The GNU tools include the C compiler, assembler, linker, and VxGDB, which is the GNU debugger modified by Wind River Systems to work in the real-time environment. VxWorks, like VRTX, normally boots by loading its system code off the network. Whereas VRTX uses TFTP to do this, VxWorks uses 'rsh' on a host workstation.

As an interesting note, prior to version 4.0, VxWorks utilized the VRTX kernel from Ready Systems. Eventually, however, the Wind River Systems packaged their own WIND kernel in VxWorks.

The WIND kernel provides comparable functionality to the other kernels. However, it lacks true round-robin scheduling, and event flags are absent. In addition, the WIND kernel does not offer any real-time clock functions, only a tick counter. On the other hand, its semaphore

[+] GNU = GNU is Not Unix Developed by the Free Software Foundation, Cambridge, Massachusetts

support is excellent—binary, counting, and mutual exclusion semaphores (with priority inversion) are included. Both pSOS+ and VRTX offer only counting semaphores Likewise, pipes are absent in both pSOS+ and VRTX32, but they're present in VxWorks VxWorks also features UNIX-style signals, which are truly asynchronous, *ie.*, a task will stop whatever it's doing at the moment, and service the signal.

WIND insists on routing interrupt routines through its kernel; however, this can be bypassed. Handling interrupts yourself should not be a problem with any of these kernels Nevertheless, VxWorks tries to convince the user to let it handle interrupts, and thus provides several interrupt-related commands in the kernel.

Unlike pSOS+ and VRTX, there is no separate mulitprocessing version of the WIND kernel. However, Wind River Systems does provide a loosely coupled backplane protocol where boards on the same bus can communicate via a socket interface.

VxWorks does not have a true board-level monitor/debugger like pSOS+ or RTScope It does however have a target shell that is a cross between the board level debugger and the RTShell type of shell, providing c expression evaluation and interactive function calls. It provides an incremental linking loader that accepts UNIX object files, and it provides all of the memory and task oriented examine/modify commands provided in the board debuggers. In addition, it also offers extensive symbolic disassembly and debugging facilities.

The level of networking support provided by VxWorks is excellent NFS and RPC support is, of course, included. In addition, 'rsh', 'telnet', and FTP are available to and from the target.

The VxGDB debugger has its origins in the UNIX GNU GDB debugger. VxGDB seems to be a serviceable, if somewhat unremarkable debugger It's not a windowed, SunView application like XRAY+ and RTSource are, so it lacks their flash and glamour. In addition, it's more of a single threaded debugger, so it's more difficult to use when debugging several tasks in one system

B.2.4 LynxOS

Summary: LynxOS (Vendor Lynx Real-time Systems, Los Angeles, California)

(on 16 MHz MC68020)
- ☆ Interrupt latency 20 µsec (worst case though 140 µsec)
- ☆ Task switch 273 µsec (worst case though 400 µsec)
- ☆ Message send/receive 625 µsec

☆☆☆ Fully POSIX 1003.1 and UNIX System V compliant
☆☆☆ Priority-based multi-threaded, preemptive kernel
☆☆☆ Fully dynamic process management and interprocessor communication with semaphores
☆☆☆ Full UNIX Networking support including TCP/IP, RPC, and NFS
☆☆ High speed contiguous files
☆ Medium-sized ROMable target application, large standard version (>1 Mbyte)
☆ NO Multiprocessor support
☆☆☆☆ Integrated development and debugging tools
☆☆☆ Widespread use and proven reliability (selected by NASA for space station)
☆☆☆ Portability MC680x0, i80386, R3000, MC88000, no source but No AT&T code either, compatible with PC architecture and a number of CPU boards

Full UNIX with all capabilites is standard, thus the development environment is second to none All the long time complaints against UNIX real-time inadequacies are addressed However, this does exact some performance penalty It is used mainly in a wide variety of 'soft' real-time applications, but has also been used to in a robotics application to control the Salisbury robot hand

LynxOS has the rare distinction of being fully POSIX[†] 1003.1 compliant, as well as UNIX System V.3 binary compatible (with BSD extensions) while maintaining a real-time core that was written from the ground up, without any AT&T source code, to meet these demanding specifications. The rewritten kernel removes the classic problems with UNIX [Cole90] and incorporates a multi-threaded, preemptive, reentrant kernel instead. Task communication facilities include sockets, semaphores, UNIX signals, messages, pipes and shared memory (virtually all the interprocess communication schemes). Other advantages over typical real-time kernels (like pSOS or VxWorks) are a full virtual task address space and hardware memory management support without much performance penalty (see section B.3), as well as very deterministic task reponse times. Moreover development tools include any UNIX compatible utilities, editors, compilers, etc, along with the ease of self-hosted development.

[†] Portable Operating System for UNIX An Internationally recognized, now widely adopted, vendor independent standard Compliance with the standard is required especially for products sold to governments and large institutional computer users

B.2.5 PDOS

Summary: PDOS (Vendor Eyring Research, Provo, Utah)

(vendor claimed on 16 MHz MC68020)

☆☆☆	Interrupt latency	6 μsec
☆☆☆	Task switch	70 μsec
☆☆	Message send/receive	200 μsec

☆☆☆ Priority-based preemptive scheduling, user programmable
☆☆ Fully dynamic process management and interprocessor communication with semaphores (limited number of processes, 64)
☆☆ Networking support,however only including TCP/IP, FTP
☆☆☆ Small-sized ROMable target application (complete kernel on 36 kBytes)
☆ NO Multiprocessor support
☆ Integrated development and debugging tools (non-symbolic debugger,limited tools)
☆☆☆ Widespread use and proven reliability
☆ Portability MC680x0, no source, though large number of such CPU boards are supported

PDOS is a small, fast kernel with full POSIX file I/O and library functions (PDOS v 4.0), and self-hosted development system which is a bit primitive even by the VxWorks standard (non-symbolic debugging, and though it has a degree of POSIX compatibility, it still falls short of source compatibility with major UNIX development tools) Standard multiprogramming kernel features are provided, but only for 64 processes at any one time Network support is optional and provides basic protocols

Like pSOS and VRTX, PDOS is of a modular design, yet it comprises an amazingly small kernel (only 36 kBytes + 80 kBytes TCP/IP network support) Of course, the advanced shell and debugging features found the larger systems are missing. Nevertheless, PDOS does manage a reasonable degree of POSIX file I/O and library compatibility (support for diskette and hard disk drives is optional). The system is self-hosting using the VMEPROM extension (removable for final product embedding), but this environment is a bit sparse especially in the debugging which is more akin to a ROM monitor than a debugger. Most people use a cross-deveopment system, many of which Eyring supports, *eg.*, compilers/libraries for cross development on IBM PC-MS/DOS, SUN3, HP-UX. Another area of weakness (for which speed is the trade-off, no doubt) is that the multi-tasking features of the kernel only allow upto 64 tasks to exist simultaneously The kernel facilities, though, are complete with timer functions, event processing, and a standard priorized round-robin scheduler.

B.3 Performance Comparison

The following performance tests are provided for a select group of real-time kernels by Kalbfleisch [Kalbfleisch91] (at the Superconducting Super Collider Laboratory in Texas) in cooperation with the respective real-time kernel vendors[†]. Throughput measurements are tabulated in Table B.2. What follows is a brief description of each test as it appears in the table. Maximum, average, and minimum times for each are given so the reader can gain an insight as to how predictable (*ie.*, least variation in the test times) each kernel is. Hardware platform used for the test was a Motorola MVME-147S-1 VMEbus single board computer (MC68030/68882 @ 25 MHz) with 1 MByte of RAM.

Test 1) *Create/Delete Task.* This test measure the time it takes to create and delete a task. A task deletes itself as soon as it is created. The created task has a higher priority than its creator, so the time quoted actually includes a create, start, delete, and two context switches.

Test 2) *Ping Suspend/Resume Task.* A low priority task resumes a suspended high priority task. The high priority task immediately suspends itself. This measurement includes two task context switches and the time it takes to suspend and resume a task. There is no facility to suspend and resume a task on LynxOS apart from signals. Thus this test was not performed under LynxOS.

Test 3) *Suspend/Resume Task.* This is identical to the previous test except that a high priority task suspends and resumes a suspended lower priority task so that there is no context switching.

Test 4) *Ping Semaphore.* Two tasks of the same priority communicate with each other through semaphores. Task A creates a semaphore, gets the semaphore then creates Task B which blocks when it attempts to get the semaphore. Task A then releases the semaphore which immediately

[†] Such comparative tests are difficult to come by in the highly competitive real-time kernel arena with reluctant vendors

unblocks Task B. Task A then attempts to get the semaphore which causes it to block until Task B releases it The two tasks then alternate ownership of the semaphore thereby causing context switches VxWorks version 4.0.2 was used which requires two separate semaphores, because round-robin scheduling is not supported (it may be possible to avoid this VxWorks version 5.0.1, but it is still difficult to implement)

Test 5) *Getting/Releasing Semaphore.* The time reported includes the time it takes to get and immediately release a semaphore within the same task context.

Test 6) *Queue Fill/Drain.* A single task sends a message to a queue which the task immediately receives on the same queue There is no task switch nor are there any pending queue operations. The next test consists of two task with two queues. The two tasks alternate execution by sending to the queue while the other is blocked waiting to receive. The total time includes context switches, queue pends and sending plus receiving a message.

Test 7) *Queue Fill, Drain, Fill Urgent.* First the time it takes to fill a queue is with messages is measured, and the time it takes to drain the queue is measured. The two tests are repeated with priority messages, *ie.*, messages going to the head of the queue. VxWorks 4.0.2 does not support message queues but ring buffers with semaphores gives the functionality of a message queue. VxWorks 5.0 now has message queues. LynxOS has UNIX System V message queues with priority messages handled differently.

Test 8) *Allocating/Deallocating Memory.* The time it takes to allocates a number of buffers from a memory partition and the time it takes to return those buffers to the partition is measured.

Test 9) *Real-Time Response.* The real-time response of the kernels by measuring the interrupt service response and the interrupt task response The interrupt service response is the time it takes to execute the first instruction of an

interrupt service routine (ISR) from when the interrupt occurs. The Task response is the time it takes for a user task to resume execution from when the interrupt occurs.

Table B.2 Comparative Performance of Some Real-Time Kernels

	pSOS+(KJM) min/max/avg	VRTX32 min/max/avg	LynxOS min/max/avg	VxWorks min/max/avg	PDOS† min
1) Create/Delete Task	540/600/591	370/380/371	—	1378/1446/1423	1113
2) Ping Suspend	120/130/128	140/150/142	—	174/182/177	79
3) Suspend/Resume	80/ 90/ 83	80/ 90/ 87	—	68/ 74/ 69	27
4) Ping Semaphore	210/220/219	230/250/239	390/400/397	228/234/232	‡
5) Get/Release Semaphore	63/ 64/ 63	55/ 56/ 55	73/ 76/ 74	33/ 34/ 33	2
6) Queue Fill	40/ 50/ 46	20/ 30/ 26	136/146/140	19/ 21/ 20	‡
6) Queue Drain	40/ 50/ 43	20/ 40/ 29	126/136/132	21/ 25/ 22	‡
6) Queue Fill/Drain	90/ 93/ 91	50/ 70/ 59	280/290/278	43/ 48/ 44	‡
7) Queue Fill, Urgent	40/ 50/ 47	20/ 30/ 27	166/175/170	70/ 76/ 72	‡
7) Alt . Qs Fill/Drain	230/240/238	250/260/252	860/900/867	366/376/371	‡
8) Alloc Memory	40/ 40/ 40	20/ 30/ 27	34/ 79/ 57	67/ 71/ 68	‡
8) Dealloc Memory	30/ 40/ 38	30/ 40/ 33	20/ 21/ 20	82/86 /83	‡
9) Interpt. Svc Response	6/ 6/ 6	6/ 6/ 6	13/ 88/ 13	6/ 56/ 6	3
9) Interpt. Task Response	100/169/163	179/343/169	163/262/175	119/319/125	41

All times given are in microseconds The pSOS+ entries use the two programming interfaces a direct C linked Library CIL(faster), and a software trap scheme KJM (slower) Note that the message queue times were fastest with VxWorks, but that these are not 'true' message queues that were tested for VxWorks

† These numbers were provided by Eyring Research after the independent tests were performed by [Kalbfleisch91] Since Eyring did not participate in the original tests, the results are somewhat suspect

‡ No data, not provided

B.3.1 CPU Performance Impact

So that the reader also has idea of the various performance variations that exist between CPUs when executing real-time kernels, Table B.3 below provides an example of a collection of RISC and CISC based CPUs:

Table B.3 Comparison of a Typical Real-time Kernel on Various Processors

CPU	R3000	MC68020	AMD29000	SPARC
Speed	16 7 MHz	25 MHz	20 MHz	16 MHz
C compiler	Mips Inc.	Whitesmiths	MetaWare	Sun Microsystems
Context switch	10	17	29	20
Timer overhead	< 1	1	1	< 1
Write system call to null device	8	29	9	8
Write to queue (interlocked)	24	91	38	32
Read from queue	24	87	38	28
Per character	< 1	< 1	< 1	< 1

All times are in microseconds Source [Andrews90(2)] As can be seen, RISC CPUs tend to be about three times faster in executing real-time kernel functions as their CISC based counterparts

B.4 Standards

In order to facility the developer of real-time systems, a standardization effort is underway in two areas: the POSIX real-time extensions proposal 1003.4 aims to standardize a number of real-time extensions to UNIX, namely threads, guaranteed interrupt response, and real-time scheduling [Cole90, Singh91]; the VITA (VMEbus Trade Association) has proposed ORKID (Open Real-time Kernel Interface Definition) for real-time kernels as a standard, which is similar to SVID (UNIX Systems V Interface Definition) [Andrews89(2)] Ready Systems, on the other hand, the overwhelming market leader, has proposed its own system BIOS (Basic Input/Ouput System) as a standard [Williams90(2)]. Basically, each of these proposals must define a simple, though sufficiently comprehensive, interface to the underlying hardware making it kernel independent 1) task control, 2) queues for multiple task priority management, 3) semaphores, 4) clock and timer controls, 5) memory allocation facilities, 6) hardware interrupt support facilities, 7) event flags, and 8) exception handling mechanisms. Implementing a multiprocessor system is straightforward; the necessary local objects are simply flaged as global.

In addition, both the POSIX committee for real-time extensions and ORKID group have decided to pool their standardization effort (P.1003.13- Real-time Applications Study Group) so that future source code is compatible with either UNIX or compliant real-time kernels (even though, initially most real-time kernel vendors were opposed to ORKID).

Appendix C. McGill Robot I/O Board

C.1 Introduction

The McGill Robot I/O board is an interface board for quadrature encoders to the VMEbus. It is simple in design and easy to implement. The board interfaces up to eight servo motors each with an encoder position feedback, index, and potentiometer. Also included is a watchdog timer circuit which times out in the event of any system failure (*ie.*, the CPU doesn't 'refresh' the watchdog in time) and shuts down the robot. The board has been successfully interfaced to PUMA series robots and forms the basic hardware interface of the *Kali* robot control system.

Encoder signals maybe either digital or differential. Sine wave (analog) encoder signal are not currently supported. The board is based on a 16-bit VMEbus slave prototyping card by XYCOM. One need only purchase parts listed herein, solder wire wrap sockets on the board, and wire wrap according to the design. All design details needed are included

Also included is the design for a power switch to enable the robot motor amplifiers, enable/disable the brakes, and provide an emergency OFF switch. This is called the Robot Power Switch Circuit. It ties directly in with the timeout disable of the Robot I/O board to provide a complete system with a high degree of safety.

C.2 Specifications

- *Compatibility*: A 16-bit VMEbus board which interfaces up to 8 quadrature encoders with differential or digital outputs including index pulses. Also up to 11 Potentiometers with 0 to 5 V range. The XYCOM 'NIKL' specification with special PROM is not implemented.
- *Module Address*: The base address for module is switch selectable on 1 kByte boundaries in either the short I/O or standard address spaces. Determined by setting 8-position DIP switch. (see XVME-085 manual).

- *Data Transfer*: Data is transferred through 16-bit data path of VMEbus. Accesses can be of any type (byte, word, long word), but only read 16-bits at a time.

C.2.1 Electrical Specifications

- *Encoder interface:*
  - Digital: Quadrature encoder A, B channels, 330 kHz (max) encoder line frequency (1.2 MHz encoder 'count' frequency). Sink 1 mA @ 5 V (max). Threshold +1.4 V (min).
  - Differential: Channels +A, -A, +B, -B, 330 kHz (max) encoder line frequency (1.2 MHz encoder 'count' frequency). Sink 1 mA @ 5 V (max). Threshold ±25 mV (min).
  - Noise immunity:

    analog filter: -3 db at 100 kHz (min) to -3 db at 2.3 MHz (max), (input filter selected with op amp buffer: TLC27L4 for 100 kHz cutoff, TLC27M4 for 700 kHz cutoff, TLC274 for 2.3 MHz cutoff)

    digital filter: 300 kHz pulse rejection for independent noise on both channels (300 kHz (max) encoder line frequency for maximum noise rejection. See Filter Optimization in HCTL-2000 data sheets, reference in section C.8).
- *Index Pulse:* Sink 1 mA @ 5 V (max). Threshold adjustable from +0.2 V to 4.95 V (max).
- *Potentiometer:* Sink 500 nA (max), 0 V to 5 V (max), 8-bits ±1/2 LSB over temperature.
- *Watchdog:* Timeout adjustable from 30 µsec to 30 sec.
- Power Requirements (including XVME-C85 VMEbus interface): 3.0 A @ +5 V (max), 0.5 A @ -12V (max), 0.1 A @ 12 V (max).

- *Operating temperature:* 0° C to 50° C.
- *Power Switch Circuit:*
  - source drive capability ALL brakes & amplifier enable: 1 A @ 24 V (max) (can be increased by relay with larger contact rating. Note the above is sufficient for a PUMA 500 series robot).

C.3 Theory of Operation

In general, robots have several types of position feedback mechanisms. The most popular being incremental optical encoders. This type of position transducer is both very accurate and inexpensive.

To determine the absolute position of a robot using incremental encoders, on being first turned on, the robot must be moved to a known position from which all subsequent motions are measured. This is called calibration and is the major drawback when using relative transducers To perform this task, the incremental encoder has a recognizable index point (index pulse) passed for every fix number of encoder pulses (usually one revolution of the motor shaft) By using an independent, low accuracy, absolute transducer (like a potentiometer) in tandem with the incremental encoder, it is possible to assign absolute values to the many 'index points' and to be able to distinguish between them using an inexpensive absolute transducer. Calibration in such a system merely involves passing through the nearest index point, resetting the encoders the instant the index pulse is detected, and reading the current potentiometer value. Note that the potentiometer must be accurate enough to read half way between successive index pulses. For example, the classic PUMA 560 robot has a joint gear ratio of about 75:1. The index pulse is once per motor revolution, so one need only distinguish between 75 pulses. Hence one needs a 2 x 75 = 150 resolution absolute position transducer. This is easily achieved with a potentiometer and an 8-bit analog to digital converter. One slight problem, though, can present itself when the potentiometer voltage range is not over the full 0 to 5 V, but over a much narrower range. In this event, signal conditioning is needed (*ie.*, level shift and amplification) to bring the signal into the desired range, or an A/D with a greater resolution could be employed.

Optical encoders consist of an LED shinning through an opaque plastic disk into which holes have been made at regular intervals around the circumference. The light from the LED is detected by a phototransistor which outputs a sine wave as the light moves across the hole. Two rings of holes are used, one rotationally offset from the other so that where the first ring has a hole the other does not. Quadrature encoding (also called times 4 encoding) uses two channels (A and B) to transmit position counts and direction using a two phase (2 bits), or 4 state encoding system (the channels are 90° out of phase corresponding to the offset of the rings). For each pulse cycle both channels form four states or encoder 'counts'. The pulse cycle duration is called the encoder *line frequency*. Hence the encoder count frequency is four times the line frequency. This allows a four times higher resolution than the holes in the encoder disk can provide (see figure C.1 below). Direction is determine by the sequence of state transitions. Typical quadrature encoder pulse frequencies are of the order of 10 kHz to 100 kHz.

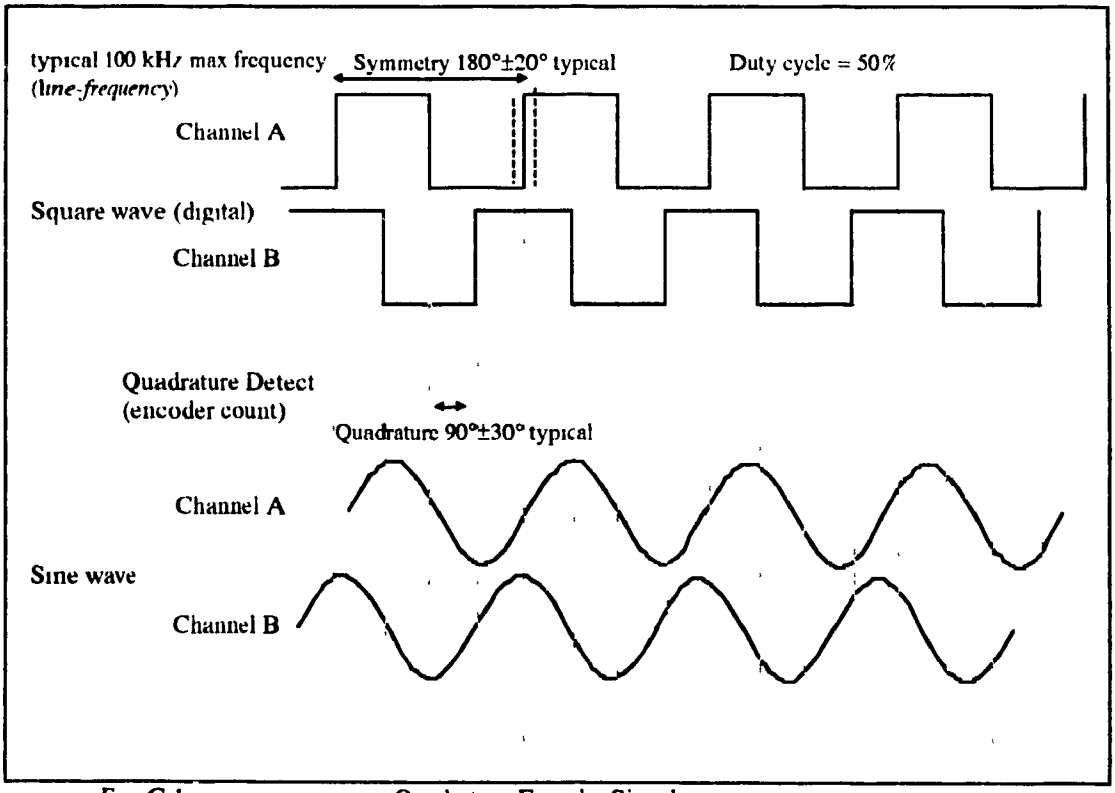


Fig. C.1 Quadrature Encoder Signals

Quadrature allows encoder signals to carry four times as much information ('counts') as their frequency by tallying rising and falling edges of two channels 90° out of phase Direction is determined by defining which signal leads and which lags, *ie.*, if channal A lags B then count up, if B lags A then count down

Quadrature encoders come in three different interface flavors. differential sine wave, differential digital, and digital. The simplest form is differential sine wave (sometimes also called analog), where the output is from the phototransistors through a differential amplifier. These differential output encoders, whether analog or digital, use transistor pairs in common and differential mode to provide two signals per encoder channel. Noise is eliminated by common mode rejection in a differential receiver. Lastly, the digital encoder provides TTL level square wave signals as the output (using comparators to convert analog sine wave to digital). Index pulse signals are either TTL or analog (see figure C.2 below).

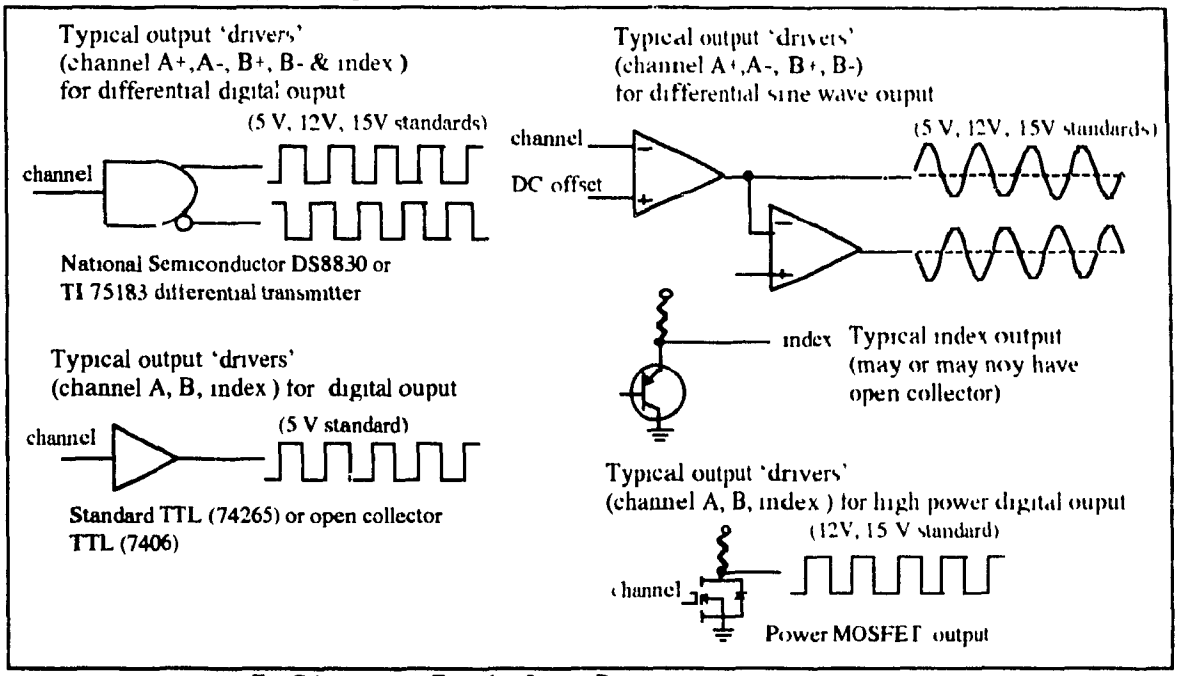


Fig C.2 Encoder Ouput Drivers

Typical output drivers for encoders include a variety of voltage and power levels (current drive) Most manufacturers offer the types shown above The DS8830 differential transmitter and the 74LS06 TTL output are the industry standards for differential digital and digital encoder drive output respectively

The Robot I/O Board interfaces with the motor optical encoders, index pulses and pots. The Hewlett-Packard HCTL-2000 12-bit quadrature decoder IC was selected as the encoder interface because it offers excellent noise immunity. But since the typical robot joint encoder range is greater than 4096 pulses, software must detect 12-bit overflow and adjust accordingly.

The index pulses are captured by PALs specially programmed to set an index bit high and reset the HP decoder chips (asynchronously) when an index pulse is received. This

asynchronous clear can cause a false reading of the decoder chip if the clear occurs during an access. This is NOT a problem since the index pulses are ONLY used during calibration when the software polls the index pulse and can discard an encoder reading just after an index pulse occurs.

The pots are read through an 8-bit A/D converter. A TI TLC532A was selected because of its convenient 11 channel input, very high input impedance, and fast conversion time. This converter also has a self-test voltage reference.

Because the *Kali* system is used mainly for experimentation a watchdog circuit is necessary for disabling the robot in case the controller fails (in software or hardware). This 'safety' circuit is simply a retriggerable timeout circuit. When a retrigger is not had within a given time, it sets off a buzzer and disables the robot.

The robot power switch is used to enable/disable the robot brakes and amplifiers simultaneously. This is not a problem. Because a servo amplifier will cut output microseconds after receiving a disable and motor brakes take milliseconds to engage, the amplifiers will be off when the brakes are applied during emergency stop. On power enable however, the brakes will be applied for some milliseconds while the amplifiers are ON. But since the robot is not moving (*ie.*, constant position servo) during power enable this will not strain the brakes.

C.4 Implementation

The XVME-085 VMEbus prototyping card incorporates the basic I/O addressing in its interface (see XVME-085 manual). For *Kali* software these addresses are dependent on the robot driver. The default PUMA 500 series driver requires these settings (1000h in short I/O space):

Switch Bank #1	Switch Bank #2
SW 8: closed	SW 4: open
SW 7: closed	SW 3: open
SW 6: closed	SW 2: open
SW 5: closed	SW 1: open
SW 4: closed	
SW 3: open	
SW 2: closed	
SW 1: closed	

The schematics for robot I/O board are presented on five sheets. The first describes the necessary circuit to control the data from the VMEbus and capture the robot index pulses The system clock on sheet #1 is set via four DIP switches The board operates at a maximum of 2 MHz (A/D chip and HP decoder chip limit the frequency). This clock is derived from the VMEbus timing service. Note that plain VME backplanes do not provide this. The bus must be setup to provide this service. Most bus arbiters as well as some CPU boards provide the system clock service.

Switch settings for the system clock are (see also listing for PAL #1 in section C.7).

	SW1	SW2	SW3	SW4
4 MHz	closed	open	open	open
2 MHz	closed	closed	open	open
1 MHz	closed	closed	closed	open
500 kHz	closed	closed	closed	closed
Single step	open	open	open	open

The variable clock was designed primarily to be used during board development (actually used very little) therefore it need not implement DIP switches. Hardwire 2 MHz.

Sheet #2 shows a safety circuit (watchdog timer) for disabling the robot in case the controlling CPU fails (in software or hardware). It is STRONGLY recommended that this circuit be implement. It works as a retriggerable time-out circuit. When a retrigger is not had within a given time, it sets off a buzzer and disables the robot. It can also be used to enable or disable the robot through software. Pot #1 controls the time-out, and Pot #2 controls the duration of the buzzer when a time-out occurs. The pots are set with the following formula:

$$3.3 \cdot 10^{-5} \cdot \textit{Pot resistance } (\Omega) = \textit{time in seconds}$$

Therefore for a timeout of 20 ms Pot #1 is set to 600 Ω. A buzzer duration of 3 seconds corresponds to a resistance of 90 kΩ for Pot #2. For standard *Kali* software set Pot #1 to 600 Ω.. Sheet #3 and #4 describe the schematics for ONE joint (or channel), either differential input (sheet #3) or digital input (sheet #4) encoders. Determine the highest encoder line frequency for robot intended to be used:

$$\textit{max speed of output shaft (revs/sec)} \cdot \textit{gear ratio} \cdot \textit{encoder counts per rev} \div 4.$$

For instance, a PUMA 500 series robot has about a 20 kHz line frequency. This determines selection of the appropriate op amp input buffer: TLC274 for 2 3 MHz cutoff,

TLC27M4 for 700 kHz cutoff, TLC27L4 for 100 kHz cutoff. Such a detailed selection of components is only to provide some additional noise immunity. One could get samples of each and check for wave form attenuation to make the optimal selection (this can be done after the board is wrapped, since the parts are pin compatible). Note though that a digital wave form has higher frequency harmonics and requires larger bandwidth to maintain its shape. If in doubt, one should use the TLC274. The original McGill Robot I/O board was built using the TLC274 The board has been in operation for many months with no problems.

The system clock frequency SHOULD NOT BE ADJUSTED to optimize the digital filter of the HCTL-2000, it is also used by A/D chip and the I/O state machines to determine access to the VMEbus. Setting the clock too low could cause timeouts on the VMEbus and setting it at 4 MHz is too fast for the HCTL-2000 and the TLC532A

For digital encoders the outputs may have to be pulled UP instead of the differential case where they must be biased. See the robot manual. For differential encoder PUMAs the circuit has been tested and is in use at McGill (see sheet # 3). Input impedance and impedance matching can also play a role since the encoder may only have a small drive capability, a relatively high frequency, and be a distance of many meters from the robot actuators.

The McGill Robot I/O Board should support any robot encoder which has either differential or digital outputs. For the digital robot circuit no settings are needed. However, for the differential output PUMAs one must set Pot #3, the index trim pot. This pot sets the reference voltage for the index pulse level. This reference voltage must 20% lower than the peak voltage of the index pulse from the robot, since the typical ripple voltage could be as much as 15%. It is easiest to do this with a dual trace scope by setting the reference voltage to one channel and the robot index pulse to the other (use a slow time base approx 10 msec).

Note that only six channels are described in the schematics, but that the board can logically support up to eight channels without adding any more support circuitry, *ie.*, sheet #1.

The two DB 25 connectors provide an easy interface to the robot. The enable and disable lines from the safety circuit can be used to turn the robot amps and brakes on/off through the Robot Power Switch Circuit.

The Robot I/O board makes NO attempt at checking for access type (*ie.*, 32-bit word request from an 8-bit port). This eliminates another PAL in the design (for the reader's convenience).

A word about grounding. For The Robot I/O board, the encoders themselves, and the index pulses, the system ground must be the same. In addition, for the potentiometers and the A/D chip (TLC532A), the reference voltage must be the same as the pot power supplied to the robot This can be a problem when using different power supplies as they are isolated The simplest solution to this problem is to derive encoder and pot power directly from the VMEbus backplane This guarantees equivalent nominal voltage levels. Note that cable shields should NOT be connected to electrical ground, but to the chassis ground of the VMEbus.

Note also, the single step clock circuit was designed only as a debugging aid to develop the board (in fact hardly used). It is not necessary to implement this circuit

Decoupling capacitors (0.01 μF) must placed from power to ground on all digital chips (Augat has sockets with these built-in). It may also be necessary to place small (about 0.1 μF) between the signal and ground for the potentiometer input lines to filter out high frequency noise.

Lastly a few words about circuit layout. The schematics only show the electrical layout of the circuit which has NO correlation with the physical layout of the components on the board It is recommended that a logical approach to laying out the components on the board be taken Since the I/O connectors are on the face plate and have the largest number of connections, the first 'layer' (op-amps of sheets #3 or #4) of the interface should be there with successive layers further away The support circuitry can fit mostly into the area adjacent VMEbus interface circuitry (sheet # 1). See sample layout in section B.7.

C.4.1 PAL Programming

Most people have heard of PALs, however many have still not used them in designs. They are programmable logic devices (PLDs) which implement AND-OR logic that greatly reduce the number of random logic ICs needed to implement a circuit. Most new circuit designs use some

PLDs. Since their introduction in the early '80's PALs have grown enormously in popularity, and indeed many more sophisticated programmable devices now exist Today most major IC manufacturers produce PALs (see your manufacturer's data book and reference in section C.8).

To program a PAL one needs a programmer (a hardware device) and the 'fuse plot' file for the PAL. It is programmed by burning 'fuses' in the device when relatively high voltages and currents are applied to the inputs. The device then behaves under regular TTL voltages and loads. It is the fuse plot that describes which fuses in the PAL are to be blown or not. It is generated from a high level description of the logic one wishes the PAL to perform. The most popular PLD language is PALASM2[†] from MMI. Most PAL programmers come with PALASM2.

The Robot I/O Board uses four different PAL programs written in PALASM2 (see section C.7. The fuse plot files in JEDEC format ready for PAL burning are available from the author on a floppy diskette). One does not need to know anything about PALs to use a programmer and create PALs for the Robot I/O board.

If the reader does not have a PAL programmer, he or she should consider purchasing one. They are very useful to have for any digital hardware project. These programmers range in cost from a few hundred dollars to about twenty thousand dollars. The prices denote the programming capability of the unit, the more expensive, the more variety of PLDs you can program. If the reader only intends to program PALs and other simple devices, then an inexpensive programmer is appropriate. But beware of which manufacturer's devices the unit will program. Cheap programmers usually program devices only from a limited number of manufacturers, and devices will have to be purchased from those manufacturers supported by the programmer.

C.4.2 Power Switch Circuit

The Robot Power switch circuit is a simple circuit. Basically it is an RS latch with the set input connected to 'enable' and reset input connected to 'disable'. The latch output controls a relay for enabling or disabling the robot brakes and amplifiers.

[†] PALASM2 is a trademark of Monolithic Memories Inc , now a division of Advanced Micro Devices Inc.

For robots with larger brakes a larger relay may be needed (the schematic shows a relay rated for PUMA 200 and 500 series robots). It may also be necessary to select fuses with different ratings. Note the power requirements for the encoders and potentiometers of the robot.

C.5 Programming

Programming the Robot I/O board is easy. It is simply memory mapped 8 or 16-bit registers. There is one 16-bit read only register for each of eight decoder chips, one 8-bit register for the index pulse capture where each bit represents an index pulse; there is one command/status register (in the XVME-085 interface), and three registers in the A/D chip.

Since the range of encoder counts for most robot joints is greater than 4096 (12-bits) the HP decoder chips cannot be used to store the absolute position of the robot joint. Rather they are used as relative counters which determine the joint displacement between updates (or servo 'ticks') The total displacement can now be stored as a 32-bit counter offering huge encoder count range However one must ensure that less than 2048 (4096 ÷ 2) counts occur between successive servo updates. This is well within the range of most robots. For instance PUMAs have a typical maximum encoder line frequency of 20 kHz, therefore at a slow 100 Hz servo sampling rate, the maximum encoder counts per sample are 20 kHz ÷100 Hz · 4 = 800 (see code in figure C.3 below)

```
#define encoder_address  0xXX1000        /* I/O board encoder address */
 short int                               /* 16-bit access to I/O board */
       raw_encoder, raw_encoder_old, delta,
 int  encoder_count,                     /* 32-bit integer holding encoder count */
 {
 /*Get raw HP encoder value */
   raw_encoder = GetIO(encoder_address) & TwelveBits,
 /*measure change between samples*/
  delta = raw_encoder - raw_encoder_old,
   raw_encoder_old = raw_encoder,        /* update raw encoder value */
 /*test for half maximum counter value, if true then assume the encoder *
  *counter has wrapped around */
  if (delta < -2048)
    delta = delta + 4096,
  else
  if (delta > 2047)
    delta = delta - 4096,
 /* update 32-bit equivalent encoder counter*/
   encoder_count = encoder_count + delta,
 }
```

Fig. C.3 Encoder Chip Interface Software

The TLC532A contains three registers (for complete details section C.8 for reference):

- Control register, write only; control start of A/D conversion and select channel of the analog multiplexer.
- Digital register, read only; high/low TTL value on some of the analog inputs.
- Analog register, read only; flag for conversion completed, 8-bit value of analog voltage ±1/2 bits.

For an example of programming the Robot I/O board see the diagnostic program source.

C.5.1 Hardware Addresses

The Robot I/O board is essential memory mapped 16-bit words (where XXXX = base address for short I/O space; FFXXXX = base address for standard memory space).

- The eight encoders are READ ONLY 16-bit words (only lower 12-bits significant) at addresses **XXC0h, XXC2h, .., XXCEh.**
- The encoder index pulses are a READ ONLY as an 8-bit register at address **X101h** or as a 16-bit word (LS-byte) at address **X100h**. This requires that the index pulse enable bit be set in the command register (see below). Each bit in the register represents the index pulse capture state for that joint. The least significant bit is for joint #0 and the most significant bit represents joint #7.
- The TLC532A analog to digital converter has three 16-bit registers at (for programming details see section B.8 for reference) :
  - WRITE ONLY control register **X110h.**
  - READ ONLY digital register **X110h.**
  - READ ONLY analog register **X112h**
- The board has an 8-bit CSR (command/status register) at address **XX81h** (8-bit) or **XX80h** (LS-byte of 16-bit access):

0: read/write -- Red LED

0 = Red LED on

1 = Read LED off

1: read/write -- Green LED

0 = Green LED off

1 = Green LED on

2: read Only interrupt pending. RESERVED FOR FUTURE USE.

3: read/write Enable robot (when safety circuit is implemented)

0 = no enable (allows disables)

1 = enable (allows disables)

- One must pulse the enable bit high then low. This allows disable pulses from the Robot Power Switch circuit to be used by the operator through an emergency OFF button. Note that to enable a robot via the Robot Power Switch circuit one MUST leave the enable bit high for approx. 15 msecs then set it low. This is the time required for the relay to switch in the Robot Power Switch circuit.

4: read/write Enable index pulse capture mode

0 = enable index pulse capture

1 = disable index pulse capture

5: read/write ReTrigger timeout circuit

0 = low state

1 = high state

- One must set high state then low state to perform a retrigger.

6: read/write Mode (safety timeout ON or not)

0 = safety timeout NOT ON

1 = safety timeout ON

7: read/write Robot Disable

0 = robot disable (overrides robot enables)

1 = robot no disable (allows enables)

- One must pulse the disable bit low then high. This allows subsequent enables by the operator through the Robot Power Switch circuit. Leaving this bit low disables the robot against any unwanted or accidental enables.

C.5.2 Diagnostic Program

A diagnostic program for testing McGill Robot I/O board is available from the author on diskette.

C.6 Schematics

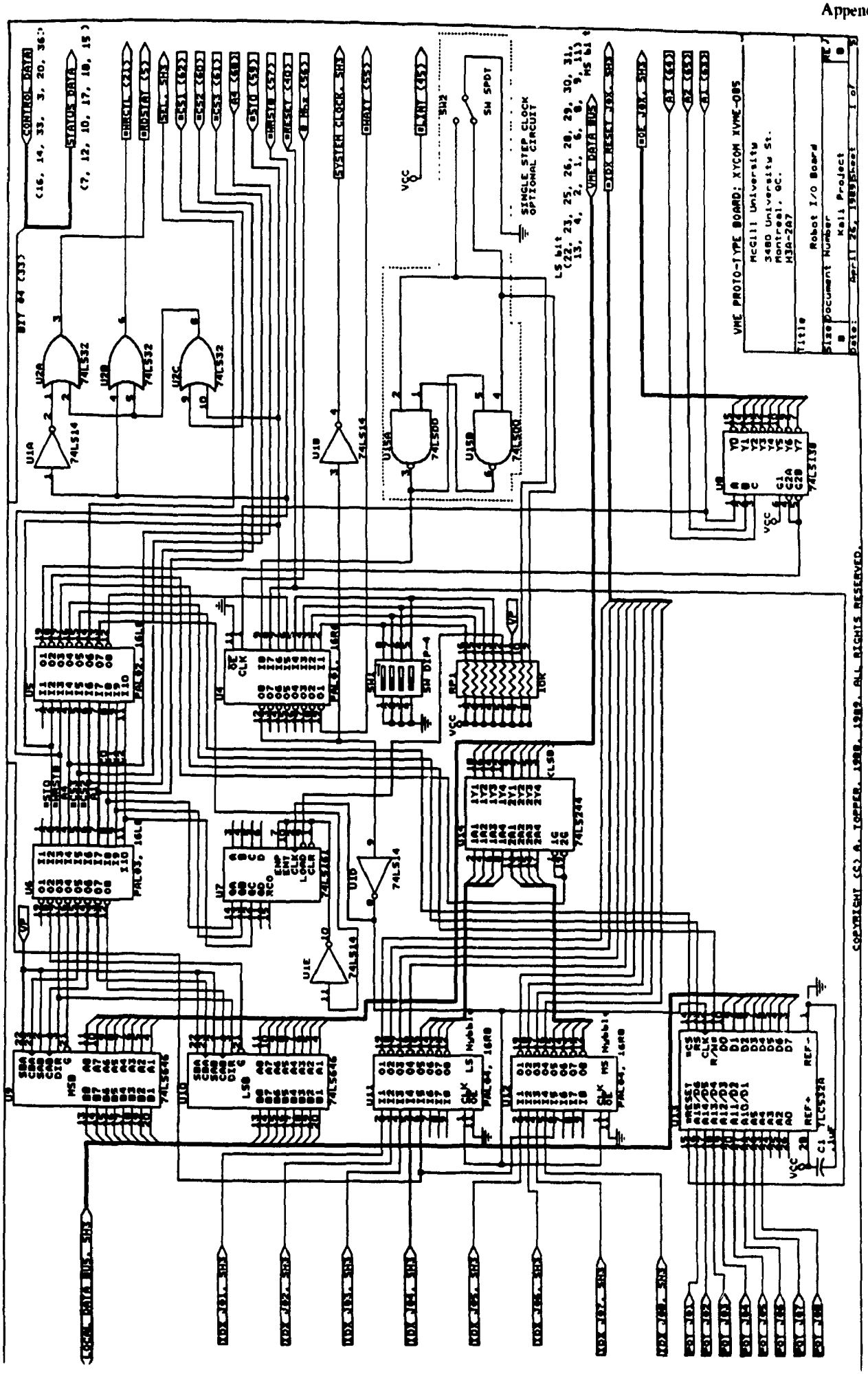
Notes for schematics:

a) ALL digital circuits must have 0.01 μF bypass capacitors between power (+5 V) and ground (this is not shown in the schematics).

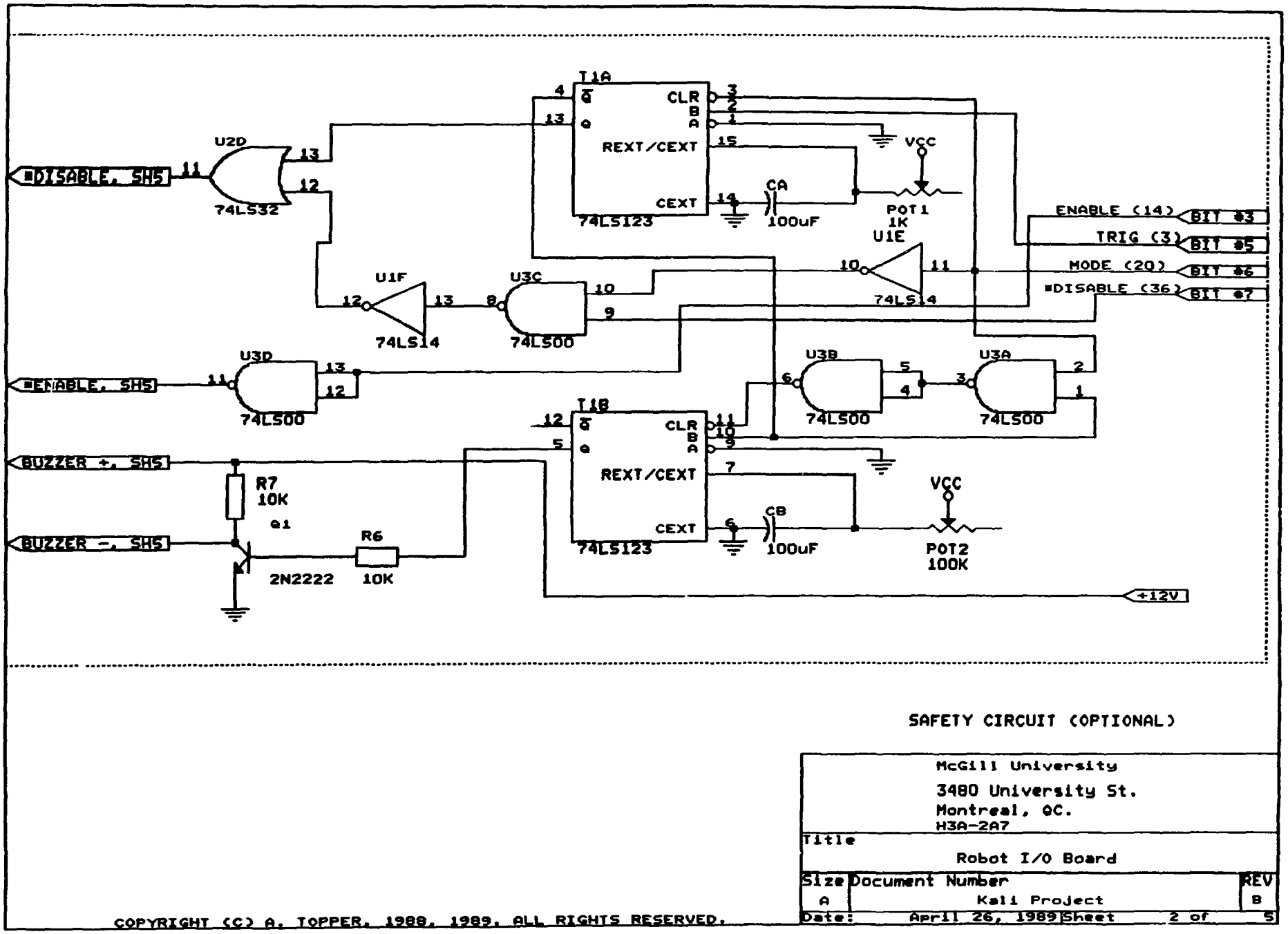
b) I/O ports indicate connection to other sheets, to external I/O connector, or XVME-085 board interface ports. An indication such as "IDX #1, SH3" means index for joint or channel #1 on schematic sheet #3.

c) For the Robot I/O board schematics, all numbers enclosed in brackets indicate a port (*ie.*, pin) number of the XVME-085 board interface (see XVME-085 manual for description of each port).

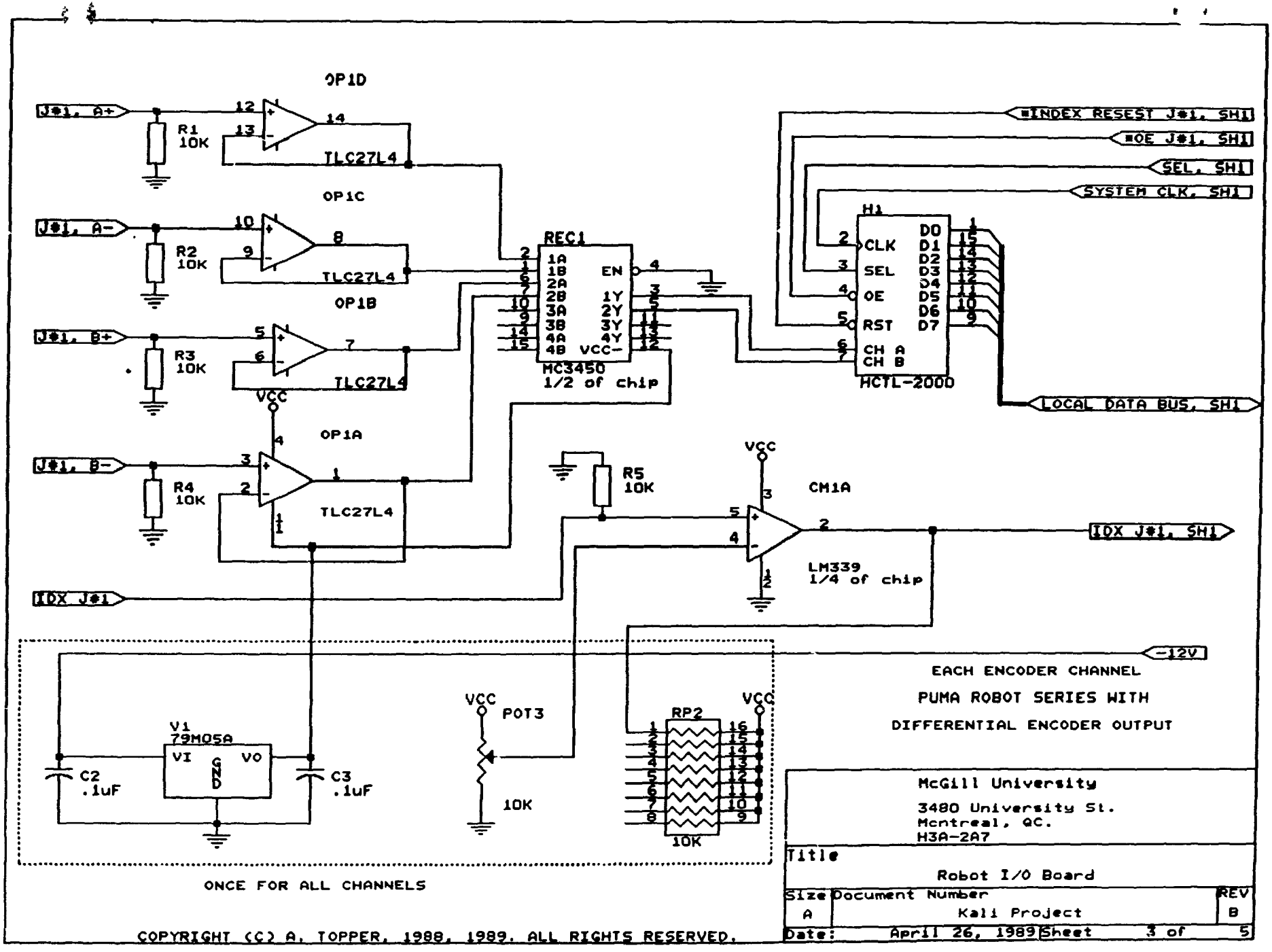
d) Circuits enclosed by dashed lines indicate a special condition on those circuits, *ie.*, optional, only once for all channels, etc. It is always accompanied by a one line explanation.



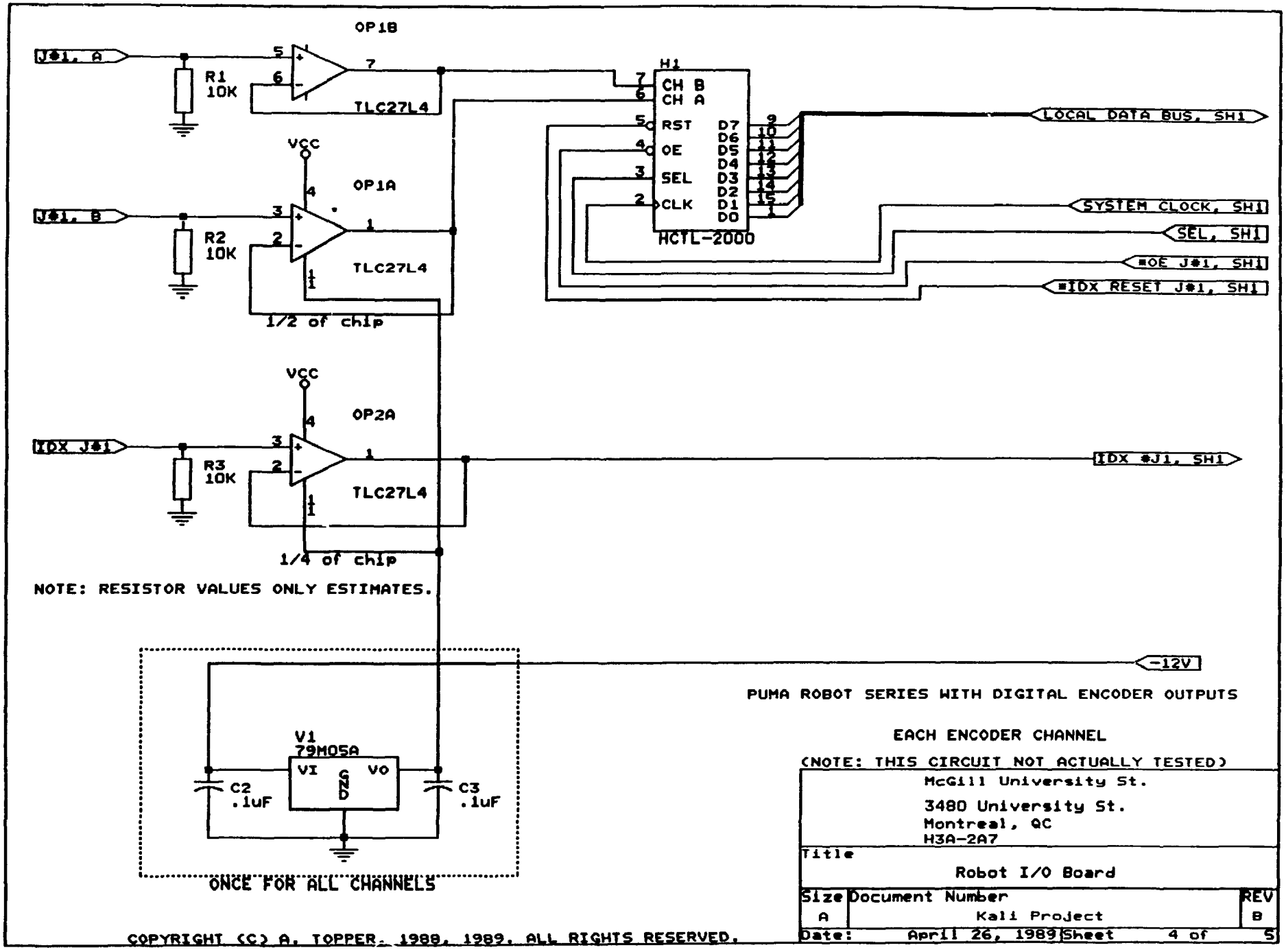
Schematic #1—Main Control Circuit



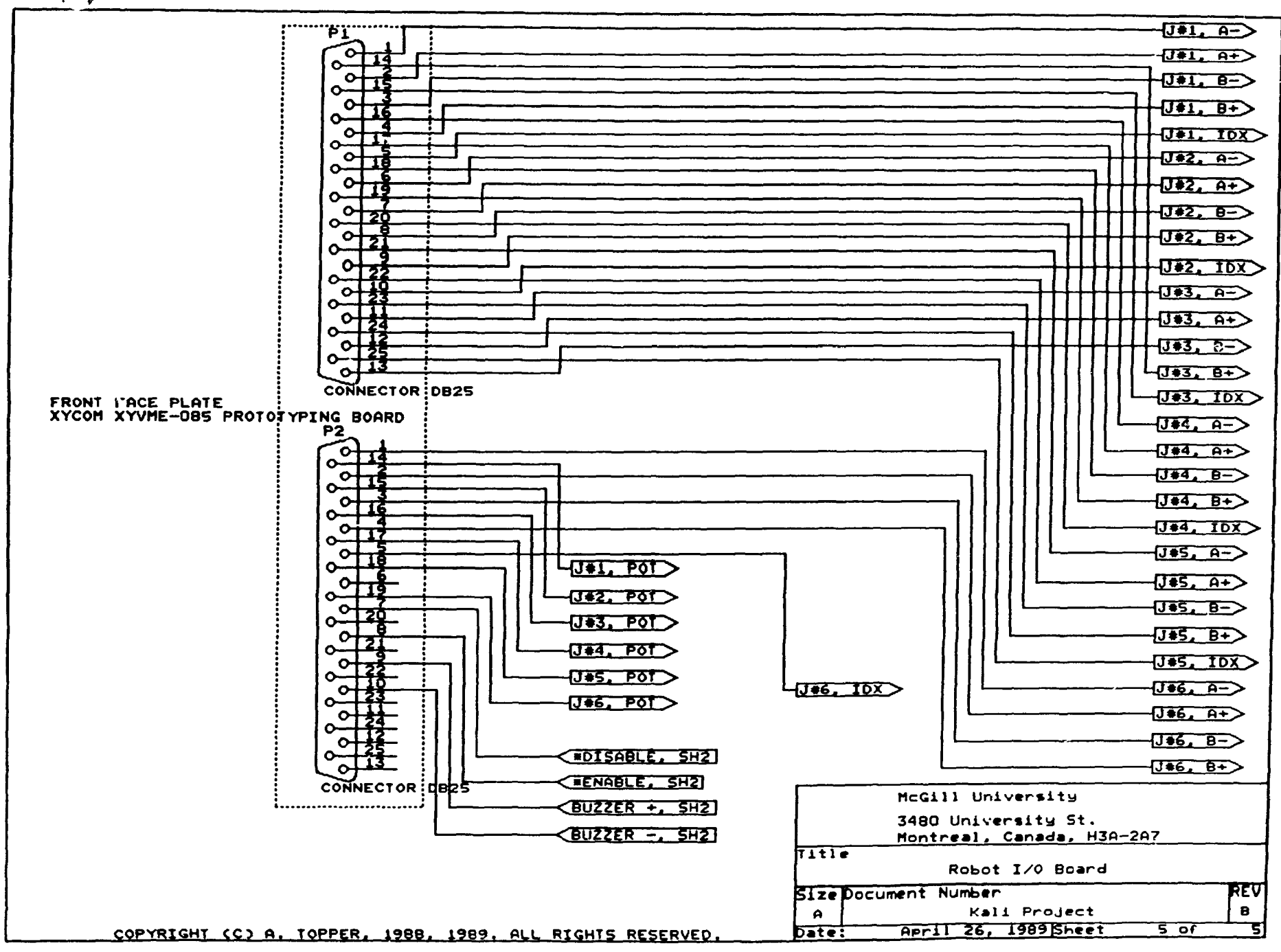
Schematic #2—Safety Circuit



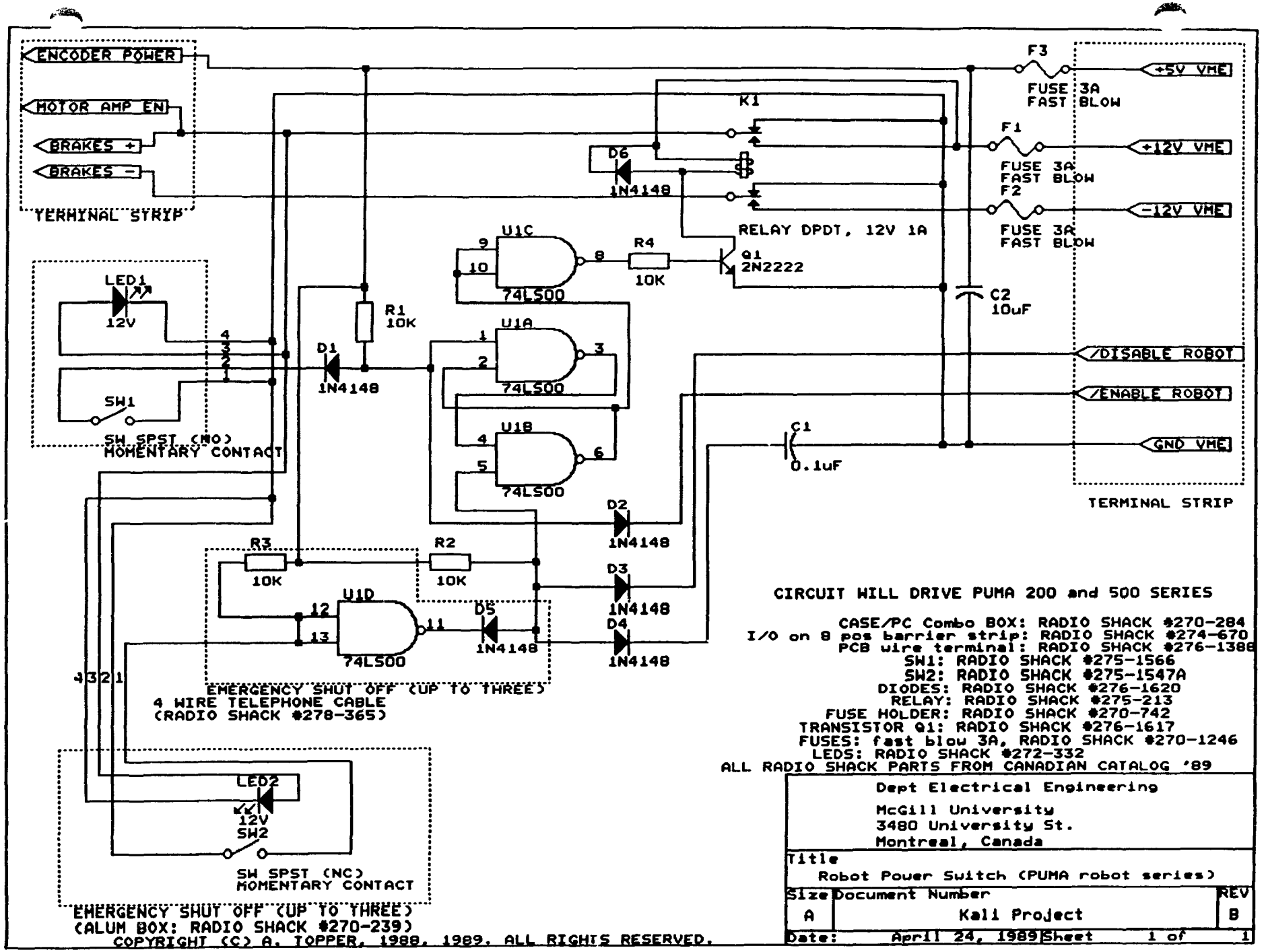
Schematic #3—Differential Encoder Interface Circuit



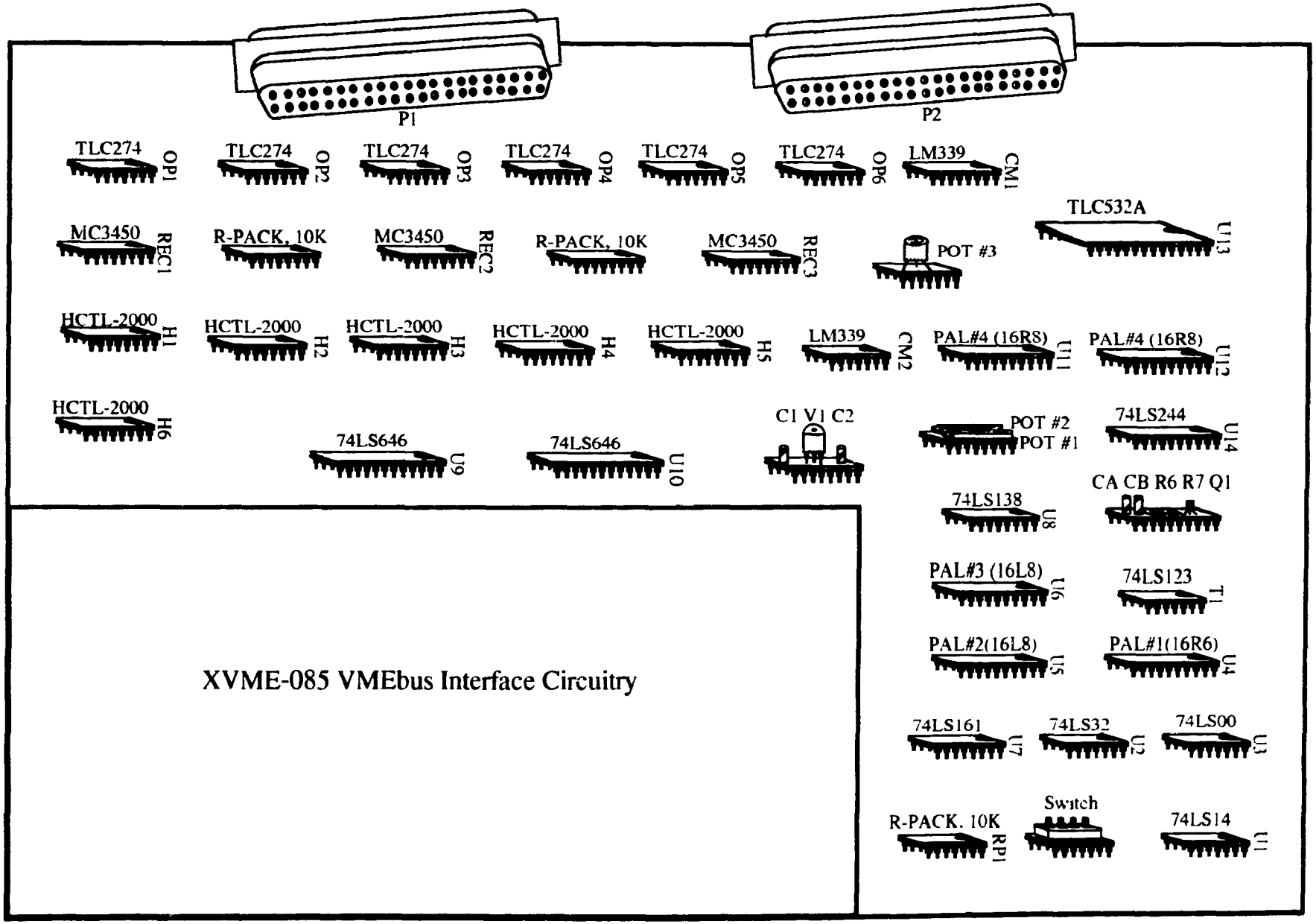
Schematic #4—Digital Encoder Interface Circuit



Schematic #5—Connector Pinout



Schematic #6—Emergency OFF switch Circuit



Circuit Side

Schematic #7 — Board layout

C.7 PAL Listings

```
Title     Robot Encoder & Pot Interface, CLOCK & WAIT CONTROL (PAL #1)
Pattern   robotb1.pds
Revision  B
Author    A. Topper
Company   McGill University
Date      4/23/89

CHIP ClockWait PAL16R6

CLK /SW1 /SW2 /SW3 /SW4 /WAIT_4_ME /ST0 /RESET /SS GND
/OC /SYSCLK /4MHZ /2MHZ /1MHZ /500KHZ /WAITS0 /WAITS /WAIT VCC

; DESCRIPTION
; Used for XYCOM, XYVME-85 prototyping board.
; Clock generator wait-state synchronizer. Generates system clock for
; board at user selected clock rate, from 500Khz to 4Mhz. Synchronizes
; WAIT to sysclock. WAIT is asserted and synchronized to the sysclock then
; WAIT is controlled by input /WAIT_4_ME.
;
; INPUT:
;  CLK (8Mhz)
;  /WAIT_4_ME is wait signal from encoder and pot read control pals (#2, #3)
;  /ST0   is start of XYVME-85 board select cycle
;  /RESET is VMEbus reset signal
;  /SS    is single step clock
;  clock select as follows (where L=CLOSED and H=OPEN for the switch bank)
;                    /SW1 /SW2 /SW3 /SW4
;              4MHz   L    H    H    H
;              2MHz   L    L    H    H
;              1MHz   L    L    L    H
;            500KHz   L    L    L    L
;       single step   H    H    H    H
;
; OUTPUT:
;  /SYSCLK is basic system clock for control pals (#2, #3, #4) and all
;          other synchronous components.
;  /WAIT  is wait state signal to XYVME-85 bus access (required for each
;         8Mhz clock cycle).
;
EQUATIONS

   4MHZ := /4MHZ * /RESET
   2MHZ := 4MHZ * 2MHZ * /RESET + /4MHZ * /2MHZ * /RESET
   1MHZ := /4MHZ * /2MHZ * /1MHZ * /RESET + 1MHZ * 2MHZ * /RESET +
           4MHZ * 1MHZ * /RESET
   500KHZ := /4MHZ * /2MHZ * /1MHZ * /500KHZ * /RESET +
             500KHZ * 1MHZ * /RESET + 500KHZ * 2MHZ * /RESET +
             500KHZ * 4MHZ * /RESET

   SYSCLK = 4MHZ * SW1 * /SW2 * /SW3 * /SW4 + 2MHZ * SW1 * SW2 * /SW3 * /SW4 +
            1MHZ * SW1 * SW2 * SW3 * /SW4 + 500KHZ * SW1 * SW2 * SW3 * SW4 +
            SS * /SW1 * /SW2 * /SW3 * /SW4
```

```
WAITS0 := ST0 * /4MHZ * SW1 * /SW2 * /SW3 * /SW4 * /RESET +
          ST0 * /4MHZ * /2MHZ * SW1 * SW2 * /SW3 * /SW4 * /RESET +
          ST0 * /4MHZ * /2MHZ * /1MHZ * SW1 * SW2 * SW3 * /SW4 * /RESET +
          ST0 * /4MHZ * /2MHZ * /1MHZ * /500KHZ * SW1 * SW2 * SW3 * SW4 *
           /RESET + ST0 * /500KHZ * /SS * /SW1 * /SW1 * /SW3 * /SW4 *
           /RESET + WAITS * ST0 * /RESET
        ; de-assert wait state when SYSCLK clock edge

WAITS  := WAITS0 * /RESET + WAITS * ST0 * /RESET
        ; delay de-assert of wait state by one 8 Mhz
        ; clock to allow slow pals to be used

WAIT   =  ST0 * WAIT_4_ME + ST0 * /WAITS

; Pinout for PAL
                Monolithic Memories PAL16R6

                    **********   **********
                    *          ***          *
                CLK ** 1                20 ** Vcc
                    *                       *
                /SW1 ** 2                19 ** /WAIT
                    *                       *
                /SW2 ** 3                18 ** /WAITS
                    *                       *
                /SW3 ** 4                17 ** /WAITS0
                    *                       *
                /SW4 ** 5                16 ** /500KHZ
                    *                       *
           /WAIT_4_ME ** 6               15 ** /1MHZ
                    *                       *
                 /ST0 ** 7               14 ** /2MHZ
                    *                       *
              /RESET ** 8                13 ** /4MHZ
                    *                       *
                  /SS ** 9               12 ** /SYSCLK
                    *                       *
                 Gnd ** 10               11 ** /OC
                    *                       *
                    ************************
```

```
Title      Robot Encoder & Pot Interface (PAL #2)
Pattern    robotd2.pds
Revision   D
Author     A. Topper
Company    McGill University
Date       23/4/89

CHIP EncoderControl PAL16L8

NC /ST0 /WRSTB A4 /CS3 /CS2 A1 C0 C1 GND
C2 /WAIT /S_COUNT /SEL /CS /RS /RW /ENIDX /OE VCC
;
; DESCRIPTION
; Encoder & Pot read control pal. Controls the reading of the HCTL-2000 chip
; in two 8 bit fetches, reading the pots via the TLC532A in two 8-bit fetches,
; and reading the encoder index.
;
; Addressing:
;  read encoder:                /CS2, A1, A2, A3 (use 74LS138 to enable one
;                                                 of eight encoders)
;                               16-bit, 0C0h - 0CEh (0C0h, 0C2h, ...)
;  read encoder index:          /CS3, /A4, 8-bit, 100h
;  read/write TLC532A regs:     /CS3, A1, A4, 16-bit, 110h - 11Eh (110h, 112h)
;                               write control reg: 110h
;                               read digital reg: 110h
;                               read analog reg: 112h
;
; INPUT:
;  /CS2, /CS3, A1, A4     chip select functions
;   C0, C1, C2            state machine counter
;  /WRSTB                 write stobe from VME I/F
;  /ST0                   start of VME cycle.
;
; OUTPUT:
;  /ENIDX                 enable reading of the encoder index pals (#4)
;  /OE                    enable HCTL-2000
;  /SEL                   select byte for HCTL-2000
;  /CS                    select TLC532A
;   RW                    Read/Write TLC532A
;   RS                    Register select TLC532A
;  /WAIT                  tell VME I/F to wait (to pal #1)
;  /S_COUNT               start state machine counter
;
; STATE MACHINE:
;              IDX      HCTL-2000            TLC532A
;              ------------------------------------------------------
;    C2 C1 C0 WAIT| WAIT  OE    SEL | WAIT     CS      RW       RS
;              ----+----------------+-----------------------------
; 0. 0  0  0   L  |  L     L     L  |  L       L       L        L
; 1. 0  0  1   H  |  H     H     H  |  H       H       WRSTB    A1
; 2. 0  1  0   L  |  H     H     H  |  H       H       WRSTB    A1
; 3. 0  1  1   L  |  H     H     L  |  H       H       WRSTB    A1
; 4. 1  0  0   L  |  H     H     L  |  L       L       L        L
; 5. 1  0  1   L  |  L     L     L  |  L       L       L        L
; 6. 1  1  0   L  |  L     L     L  |  L       L       L        L
; 7. 1  1  1   L  |  L     L     L  |  L       L       L        L
;              ------------------------------------------------------
; Standard binary count can be used instead of Grey code count since the
; settling time between states is much longer than the logic delay (applies to
; other PALs also).
```

```
EQUATIONS

   ENIDX = ST0 * CS3 * /A4

   OE =    ST0 * CS2 * C0 * /C1 * /C2 +
           ST0 * CS2 * /C0 * C1 * /C2 +
           ST0 * CS2 * C0 * C1 * /C2  +
           ST0 * CS2 * /C0 * /C1 * C2

   SEL =    ST0 * CS2 * C0 * /C1 * /C2 +
            ST0 * CS2 * /C0 * C1 * /C2

   CS =  ST0 * CS3 * A4 * C0 * /C1 * /C2 +
         ST0 * CS3 * A4 * /C0 * C1 * /C2 +
         ST0 * CS3 * A4 * C0 * C1 * /C2

   RS =  ST0 * CS3 * A4 * A1 * C0 * /C1 * /C2 +
         ST0 * CS3 * A4 * A1 * /C0 * C1 * /C2 +
         ST0 * CS3 * A4 * A1 * C0 * C1 * /C2

   RW =  ST0 * CS3 * A4 * WRSTB * C0 * /C1 * /C2 +
         ST0 * CS3 * A4 * WRSTB * /C0 * C1 * /C2 +
         ST0 * CS3 * A4 * WRSTB * C0 * C1 * /C2

   WAIT = ST0 * CS2 * C0 * /C1 * /C2 +
          ST0 * CS2 * C1 * /C2  +
          ST0 * CS2 * /C0 * /C1 * C2 +
          ST0 * CS3 * /A4 * C0 * /C1 * /C2 +
          ST0 * CS3 * A4 * C0 * /C1 * /C2 +
          ST0 * CS3 * A4 * C1 * /C2

   S_COUNT =  ST0 * CS2 + ST0 * CS3 * A4

; Pinout for PAL
                Monolithic Memories PAL16L8

                    **********   **********
                    *          ***         *
                 NC ** 1                 20 ** Vcc
                    *                     *
               /ST0 ** 2                 19 ** /OE
                    *                     *
              /WRSTB ** 3                 18 ** /ENIDX
                    *                     *
                 A4 ** 4                 17 ** RW
                    *                     *
               /CS3 ** 5                 16 ** RS
                    *                     *
               /CS2 ** 6                 15 ** /CS
                    *                     *
                 A1 ** 7                 14 ** SEL
                    *                     *
                 C0 ** 8                 13 ** /S_COUNT
                    *                     *
                 C1 ** 9                 12 ** /WAIT
                    *                     *
                Gnd ** 10                 11 ** C2
                    *                     *
                    ***********************
```

```
Title     Robot Encoder & Pot Interface (PAL #3)
Pattern   robotd3.pds
Revision  D
Author    A. Topper
Company   McGill University
Date      23/4/89

CHIP LatchControl PAL16L8

NC /ST0 /WRSTB A4 /CS3 /CS2 A1 C0 C1 GND
C2 CAB2 CBA2 CAB1 CBA1 /G1 DIR /G2 NC VCC
;
; DESCRIPTION
; Encoder & Pot latch control pal. Controls the latches interfacing VME-bus
; to the HCTL-2000 and TLC532A using two 8-bit fetches.
;
; Addressing:
;  read encoder:                /CS2, A1, A2, A3 (use 74LS138 to enable one
;                                                 of eight encoders)
;                               16-bit, 0C0h - 0CEh (0C0h, 0C2h, ...)
;  read/write TLC532A regs:     /CS3, A1, A4, 16-bit, 110h - 11Eh (110h, 112h)
;                               write control reg: 110h
;                               read digital reg: 110h
;                               read analog reg: 112h
;
; INPUT:
;  /CS2, /CS3, A1, A4     chip select functions
;   C0, C1, C2            state machine counter
;  /WRSTB                 write stobe from VME I/F
;  /ST0                   start of VME cycle.
;
; OUTPUT:
;  CBA1        capture bus B (latch MSB)
;  CAB1        capture bus A (latch MSB)
;  CBA2        capture bus B (latch LSB)
;  CAB2        capture bus B (latch LSB)
;  /G2         enable output bus B
;  /G1         enable output bus A
;  DIR         direction control
;
;
; STATE MACHINE:
;                        encoder read:
;
;    C2 C1 C0 CBA1  CAB1   CBA2  CAB2  DIR   G1   G2
;             ------------------------------------------
; 0. 0  0  0   L     L      L     L     H     L    L
; 1. 0  0  1   L     L      L     L     H     L    L
; 2. 0  1  0   H     L      L     L     H     L    L
; 3. 0  1  1   H     L      H     L     H     L    L
; 4. 1  0  0   H     L      H     L     H     L    L
; 5. 1  0  1   H     L      H     L     L     H    H
; 6. 1  1  0   H     L      H     L     L     H    H
; 7. 1  1  1   H     L      H     L     L     H    H
;
```

```
;                          TLC532 read
;
;     C2  C1  C0  CBA1   CAB1    CBA2   CAB2   DIR    G1   G2
;                  ------------------------  -      -
; 0.  0   0   0    L      L       L      L      H      L    L
; 1.  0   0   1    L      L       L      L      H      L    L
; 2.  0   1   0    H      L       L      L      H      L    L
; 3.  0   1   1    H      L       H      L      H      L    L
; 4.  1   0   0    H      L       H      L      L      H    H
; 5.  1   0   1    H      L       H      L      L      H    H
; 6.  1   1   0    H      L       H      L      L      H    H
; 7.  1   1   1    H      L       H      L      L      H    H
;
;                          TLC532 write
;
;     C2  C1  C0  CBA1   CAB1    CBA2   CAB2   DIR    G1   G2
;                  ------------------------------------  -
; 0.  0   0   0    L      L       L      L      H      L    L
; 1.  0   0   1    L      H       L      L      H      H    L
; 2.  0   1   0    L      H       L      H      H      L    H
; 3.  0   1   1    L      H       L      H      H      L    L
; 4.  1   0   0    L      H       L      H      H      L    L
; 5.  1   0   1    L      H       L      H      H      L    L
; 6.  1   1   0    L      H       L      H      H      L    L
; 7.  1   1   1    L      H       L      H      H      L    L
;
EQUATIONS

   /CBA1  = /C1 * /C2 +
            ST0 * CS2 * /WRSTB * /C0 * C1 * /C2 +
            ST0 * CS3 * A4 * WRSTB * /C0 * C1 * /C2 +
            ST0 * CS3 * A4 * WRSTB + /ST0

   /CAB1 = /C0 * /C1 * /C2 +
           ST0 * CS2 * /WRSTB +
           ST0 * CS3 * A4 * /WRSTB +
           ST0 * CS3 * A4 * WRSTB * /C0 * /C1 * /C2 +
           ST0 * CS3 * /A4 + /ST0

  /CBA2  = /C1 * /C2 +
           /C0 * C1 * /C2 +
            ST0 * CS2 * /WRSTB * C0 * C1 * /C2 +
            ST0 * CS3 * A4 * /WRSTB * /C0 * C1 * /C2 +
            ST0 * CS3 * A4 * /WRSTB * C0 * /C1 * /C2 +
            ST0 * CS3 * A4 * WRSTB + /ST0

  /CAB2  = /C0 * /C1 * /C2 +
           ST0 * CS2 * /WRSTB +
           ST0 * CS3 * A4 * /WRSTB +
           ST0 * CS3 * A4 * WRSTB * /C0 * /C1 * /C2 +
           ST0 * CS3 * A4 * WRSTB * C0 * /C1 * /C2 +
           ST0 * CS3 * /A4 + /ST0

      G1 =  ST0 * CS2 * C0 * /C1 * C2 +
            ST0 * CS2 * /C0 * C1 * C2 +
            ST0 * CS2 * C0 * C1 * C2 +
            ST0 * CS3 * /WRSTB * A4 * C2 +
            ST0 * CS3 * WRSTB * A4 * /C2 * /C1 * C0
```

```
      G2 =  ST0 * CS2 * C0 * /C1 * C2 +
            ST0 * CS2 * /C0 * C1 * C2 +
            ST0 * CS2 * C0 * C1 * C2 +
            ST0 * CS3 * /WRSTB * A4 * C2 +
            ST0 * CS3 * WRSTB * A4 * /C2 * C1 * /C0

    /DIR  =  ST0 * CS2 * /WRSTB * C0 * /C1 * C2 +
            ST0 * CS2 * /WRSTB * /C0 * C1 * C2 +
            ST0 * CS2 * /WRSTB * C0 * C1 * C2 +
            ST0 * CS3 * /WRSTB * A4 * C2

; Pinout
                Monolithic Memories PAL16L8

                   **********   **********
                   *          ***         *
                CLK **  1                20 ** Vcc
                   *                     *
               /ST0 **  2                19 ** NC
                   *                     *
             /WRSTB **  3                18 ** /G2
                   *                     *
                A4 **  4                17 ** DIR
                   *                     *
               /CS3 **  5                16 ** /G1
                   *                     *
               /CS2 **  6                15 ** CBA1
                   *                     *
                A1 **  7                14 ** CAB1
                   *                     *
                C0 **  8                13 ** CBA2
                   *                     *
                C1 **  9                12 ** CAB2
                   *                     *
               Gnd ** 10                11 ** C2
                   *                     *
                   ************************
```

```
Title     Robot Encoder & Pot interface board, ENCODER INDEX (PAL #4)
Pattern   robotc4.pds
Revision  A
Author    A. Topper
Company   McGill University
Date      24/4/88

CHIP EncoderIndex PAL16R8

CLK IRAW0 IRAW1 IRAW2 IRAW3 /EN NC NC NC GND
/OC I0 I1 I2 I3 /RST0 /RST1 /RST2 /RST3 VCC

; DESCRIPTION
; Encoder index pal. Captures index and holds it when enabled (used for
; 4-channels).

; INPUT:
;   /IRAW0, ... /IRAW3  raw input of encoder index pulse
;   /EN enable index pulses to be captured
; OUTPUT:
;   I0, ... I3 encoder index pulse
;   /RST0, ... /RST3 reset pulse for IRAW width when it is received (used
;                    to reset HCTL-2000)
EQUATIONS

        /I0 := /IRAW0 * /I0 + /IRAW0 * /EN + /I0 * /EN + /EN
        /I1 := /IRAW1 * /I1 + /IRAW1 * /EN + /I1 * /EN + /EN
        /I2 := /IRAW2 * /I2 + /IRAW2 * /EN + /I2 * /EN + /EN
        /I3 := /IRAW3 * /I3 + /IRAW3 * /EN + /I3 * /EN + /EN

        RST0 := IRAW0 * I0 * EN
        RST1 := IRAW1 * I1 * EN
        RST2 := IRAW2 * I2 * EN
        RST3 := IRAW3 * I3 * EN

; Pinout
                Monolithic Memories PAL16R8

                    **********   **********
                    *         ***          *
                CLK **  1                20 ** Vcc
                    *                    *
              IRAW0 **  2                19 ** /RST3
                    *                    *
              IRAW1 **  3                18 ** /RST2
                    *                    *
              IRAW2 **  4                17 ** /RST1
                    *                    *
              IRAW3 **  5                16 ** /RST0
                    *                    *
                /EN **  6                15 ** I3
                    *                    *
                 NC **  7                14 ** I2
                    *                    *
                 NC **  8                13 ** I1
                    *                    *
                 NC **  9                12 ** I0
                    *                    *
                Gnd ** 10                11 ** /OC
                    *                    *
                    ************************
```

C.8 IC Data Sheet References

The references for data sheets of all IC components used.

Part	Reference	Page
74LS00	The TTL Data Book Vol. 2, 1985, Texas Intruments	3-3
74LS14	The TTL Data Book Vol. 2, 1985, Texas Intruments	3-85
74LS32	The TTL Data Book Vol. 2, 1985, Texas Intruments	3-151
74LS123	The TTL Data Book Vol. 2, 1985, Texas Intruments	3-477
74LS138	The TTL Data Book Vol. 2, 1985, Texas Intruments	3-527
74LS161	The TTL Data Book Vol. 2, 1985, Texas Intruments	3-599
74LS244	The TTL Data Book Vol. 2, 1985, Texas Intruments	3-817
74LS646	The TTL Data Book Vol. 2, 1985, Texas Intruments	3-1241
HCTL-2000	Opto-Electronics Designers Guide., 1988, Hew-Pack.	4-67
PALs	PAL/PLE Device Prog. Logic Array Handbk., MMI.	–
TLC532A	Interface Circuits Data Book, 1987, Texas Instruments	2-139
TLC274	Linear Circuits Data Book, 1984, Texas Instruments	3-187
MC3450	Interface Circuits Data Book, 1987, Texas Instruments	4-35
LM339	Linear Circuits Data Book, 1984, Texas Instruments	4-25
79M05	Linear Circuits Data Book, 1984, Texas Instruments	6-207

References

Adept90, "Adept A-Series Controllers and AdeptOne Robot," Adept Inc., San Jose, California, June 1990.

Agapakis90 Agapakis, J. E., Katz, J. M., and Pieper, D. L., Automatix Inc., "Programming and Control of Multiple Robotic Devices in Coordinated Motion," *Proc. IEEE Int. Conf. on Robotics and Automation 1990*, pp. 362—367.

Ahmad88 Ahmad, S., "Issues In The Design of Multiprocessor-Based Robot Control Hardware," *IEEE Workshop on Special Computer Architectures for Robotics*, April 1988, pp. 127—144.

Ahmad88 (2) Ahmad, S., Guo, H., "Dynamic Coordination or Dual-Arm Robotic Systems With Joint Flexibility," *Proc. IEEE Int. Conf. on Robotics and Automation 1988*, pp. 332—337.

Allworth87 Allworth, S. T., Zobel, N., Introduction to Real-Time Software Design, 2nd ed., Springer-Verlag, New York, New York, 1987.

AMD89 ,"29k Family 1990 Data Book," Advanced Micro Devices, 1989.

AMD90, "29050 Data Sheet (preliminary)," Advanced Micro Devices, 1990.

Amdahl67 Amdahl, G. M., "Validity of the single processor approach to achieving large scale computing capabilities," *Proc. AFIPS.*, vol. 30, Thompson, Washington D C., 1967, pp. 483—485.

Andersson89 Andersson, R. L., "Computer Architectures for Robot Control: A Comparison and A New Processor Delivering 20 Real MFLOPS," *Proc. IEEE Int. Conf. on Robotics and Automation 1989*, pp. 1162—1167.

Andrews89 Andrews, W., "32-bit buses Contend for Designer's Attention," *Computer Design*, November 1989, pp. 78—96.

Andrews89(2) Andrews, W., "ORKID: The Standard No One Wants?," *Computer Design*, August 1989, pp. 44—46.

Andrews90 Andrews, W., "Designers Pack Intelligence, memory, speed into new DSPs," *Computer Design*, April 1990, pp. 82—97.

Andrews90(2) Andrews, W., "RISC vs. CISC Debate Moves into Real-Time Turf," *Computer Design*, January, 1990, pp. 44—48.

ANSI/RIA89, "Point-to-Point and Static Performance Characteristics for Industrial Robots," *ANSI/RIA R15.05-1-1990*, American National Standards Institute, September, 1990.

Asada86 Asada, H., Slotine, J. E., Robot Analysis and Control, John Wiley & Sons, New York, New York, 1986.

AT&T87, UNIX System V (v.2) Programmer's Reference Manual, Prentice-Hall, Englewood Cliffs, New Jersey, 1987 (also available on-line on most UNIX systems under heading shmget(2)).

Backes89 Backes, P., Hyati, S., Hayward, V., and Tso, K., "The KALI Multi-Arm Robot Programming and Control Environment," *Proc. of NASA Conf. on Space Telerobotics*, 1989, pp. 173—182.

Bejczy74 Bejczy, A. K., "Robot Arm Dynamics and Control," *Jet Propulsion Laboratory Technical Memo 33-669*, Pasadena, California, February 1974.

Bejczy87 Bejczy, A. K., Szakaly, Z., "Universal computer control system (UCCS) for space telerobots," *Proc. IEEE Int. Conf. on Robotics and Automation 1987*, pp. 318—324.

Bihn88 Bihn, D. G, Hsia, T. C., "Universal Six-Joint Robot Controller," *IEEE Transactions on Computers*, February 1988, pp. 31—35.

Bihari89 Bihari, T., Gopinath, P., Schwan, K., "Object-Oriented Design of Real-Time Software," *IEEE Real-Time Systems Symposium 1989*, pp. 194—201.

Birman90 Birman, M., *et. al.*, "Developing the WTL3170/3171 Sparc Floating-Point Coprocessors," *Micro*, February 1990, pp. 55—64.

Borrill85 Borrill, P. L., "Micro Standards Special Feature: A Comparison of 32-bit Buses," *IEEE Mirco*, December 1985, pp. 71—79.

Brooks75 Brooks, F. P., The Mythical Man-Month, Addison-Wesley, Reading, Massachusetts, 1975.

Brooks87 Brooks, F. P., "No Silver Bullet: Essence and Accidents of Software Engineering," *Computer*, April 1987, pp. 10—18.

Butner88 Butner, S., Wang Y., Mangaser, A., and Jordan, S., "Design and Simulation of RIPS, An Advanced Robot Control System," *Proc. IEEE Int. Conf. on Robotics and Automation 1988*, pp. 470—474.

Campbell90 Campbell, J. E., Helmes, G. D., Adept Technology Inc., "The Integrated Controls Platform," *Proc. International Robots & Vision Automation Conf. 1990*, pp. 13-28—13-43.

Chang88 Chang, P. R., Lee, C. S. G., "Residue Arithmetic VLSI Array Architecture for Manipulator Pseudo-Inverse Jacobian Computation," *Proc. IEEE Int. Conf. on Robotics and Automation 1988*, pp. 297—302.

Chen86 Chen, J. B., Fearing, R. S., Armstrong, B. S., and Burdick, J. W., "NYMPH: A Multiprocessor for Manipulation Applications," *Proc. IEEE Int. Conf. on Robotics and Automation 1986*, pp. 1721—1736.

Child91 Child, J., "High-performance microprocessors push limits of VMEbus," *Computer Design*, March 1991, pp. 105—114.

Clark89 Clark, D., "HIC: An Operating System for Hierarchies of Servo Loops," *Proc. IEEE Int. Conf. on Robotics and Automation 1989*, pp. 1004—1009.

Cole90 Cole, C. T., "Real-Time Unix: Fact or Fantasy," *Unix Review*, October 1990, pp. 40—44.

Coonan80 Coonan, J. T., "An Implementation Guide to a Proposed Standard for Floating-Point Arithmetic," *Computer*, January 1980.

Copley88, "PWM Servo Amplifiers Model 220/230 Specifications," Copley Controls Corp., Newton, Massachusetts, 1988.

Craig88 Craig, J. J., Adaptive Control of Mechanical Manipulators, Addison-Wesley, Reading, Massachusetts, 1988.

Craig88(2) Craig, J. J., Silma Inc., "Issues in the Design of Off-line Programming Systems," *Robotics Research*, 4th International Symposium, pp. 379—389.

Craig89 Craig, J. J. , Introduction to Robotics Mechanics and Control, 2nd ed., Addison-Wesley, Reading, Massachusetts, 1989.

Darley90 Darley, M., *et. al.*, "The TMS39C602A Floating-Point Coprocessor for Sparc Systems," *Micro*, June 1990, pp. 36—47.

Dinning89 Dinning A., "A Survey of Synchronization Methods for Parallel Computers," *Computer*, July 1989, pp. 66—77.

Dote90 Dote, Y., Servo Motor and Motion Control Using Digital Signal Processors, Prentice Hall, Englewood Cliffs, New Jersey, 1990.

Dyer88 Dyer, S. A., Morris, L. R., "Floating-Point Digital Signal Processing Chips, A New ERA for DSP Systems Design," *Micro*, December 1988, pp. 10—13.

Edenfield90 Edenfield, *et. al.*, "The 68040 Processors: Part 1, Design and Implementation," *Mirco*, February, 1990, pp. 66—78.

Feitelson90 Feitelson, D. G., Rudolph, L.,"Distributed Hierarchical Control for Parallel Processing," *Computer*, May 1990, pp. 65—77.

Fijany91 Fijany, A., Bejczy, A. K., "Parallel Algorithms and Architecture fo Computation of Manipulator Forward Dynamics," *Proc. IEEE Int. Conf. on Robotics and Automation 1991*, pp. 1156—1161.

Fijany91 (2) Fijany, A., Bejczy, "Fast Parallel Computation of Manipulator Forward Dynamics or Two-Dimensional Processor Array," Submitted to *IEEE Trans. Robotics and Automation.*

Fleischer88 Fleischer, W. A., "How to Select DC Motors," *Machine Design*, November 1988, pp. 99—103.

Franklin86 Franklin, G., Powell, J. D., and Emami-Naeini, A., Feedback Control of Dynamic Systems, Addison-Wesley, Reading, Massachusetts, 1986.

Fuccio88 Fuccio, M. L., *et. al.*, "The DSP32C: AT&T's Second-Generation Floating-Point Digital Signal Processor," *Micro*, December 1988, pp. 30—48.

Gentleman89 Gentleman, W.M., MacKay, S.A., Stewart, D.A., and Wein, M., "Using the Harmony Operating System: Release 3.0," *National Research Council Report ERA-377*, February 1989.

Glass91 Glass, B., "Under the Hood: SPARC Revealed," *Byte*, April 1991, pp. 295—302.

Gimarc87 Gimarc, C. E., Milutinovnic, V. M., "A Survey of RISC processors and computers of the mid-1980s," *Computer*, September 1987, pp. 59—69.

Golwasser84 Goldwasser, S. M., "Computer architecture for grasping," *Proc. IEEE Int. Conf. on Robotics and Automation 1984*, pp. 320—325.

Gopinath89 Gopinath, P., Schwan, K., "CHAOS. Why one cannot have only an operating system for real-time applications," *ACM Operating Systems Review*, July, 1989, pp. 106—125.

Graunke90 Graunke, G., Thakkar, S., "Synchronization Algorithms for Shared-Memory Multiprocessors," *Computer*, June 1990, pp. 60—69.

Hackett90 Hackett, J. K., Mubarak, S., "Multi-Sensor Fusion: A perspective," *Proc. IEEE Int. Conf. on Robotics and Automation 1990*, pp. 1324—1330.

Han89 Han, J.Y., Wang, C.Y., "Modeling and Performance Evaluation of Multiprocessor Systems for Real-Time Nonlinear Robot Control," *Proc. IEEE Int. Conf. on Robotics and Automation 1989*, pp. 1016—1021.

Harber88 Harber, R.G., Hu, X., Li, J., and Bass, S.C., "The Application of Bit-Serial CORDIC Computational Units to the Design of Inverse Kinematics Processors," *Proc. IEEE Int. Conf. on Robotics and Automation 1988*, pp. 1152—1163.

Hayward84 Hayward, V., Paul, R., "Introduction to RCCL: A Robot Control C Library," *Proc. IEEE Int. Conf. on Robotics and Automation 1984*, pp. 293—297.

Hayward86 Hayward, V., Paul, R., "Robot Manipulator Control Under UNIX RCCL: A Robot Control 'C' Library," *International Journal of Robotics Research*, vol. 5, no. 4., 1986, pp. 94—111.

Hayward88 Hayward, V., Hayati, S., "Kali: An Environment for the Programming and Control of Cooperative Manipulators," *American Control Conference, 1988*, pp. 473—478.

Hayward88(2) Hayward,V., Daneshmend, L., Nilakantan, A., "Model Based Trajectory Planning Using Preview," *SPIE Conf., Space Automation IV*, 1988, pp. 186—193.

Hayward89 Hayward, V., Daneshmend, L. K., Hayati, S., "An Overview of Kali: A System to Program and Control Cooperative Manipulators," *Fourth International Conference on Advanced Robotics 1989*, pp. 547—558.

Hayward91 Hayward, V., Nilakantan, A., Daneshemnd, L. K., "Trajectory Generation and Control for Automatic Manipulation," Submitted to *Robotica*.

Hayati90 Hayati, S., Lee, T., Tso, K., Backes, P., and Lloyd, J., "A Testbed for a Unified Teleoperated-Autonomous Dual-Arm Robotic System," *Proc. IEEE Int. Conf. on Robotics and Automation 1990*, pp. 1090—1095.

Herndon89 Herndon, J. N., et al., "Telerobotic Manipulator Developments for Ground-Based Space Research," *Proc. of ANS 3rd Topical Meeting on Robotics and Remote Systems*, March 1989.

Hiroshi90 Hiroshi, S., "History and Future Development Trends of Robot Controllers," *Proc. International Robots & Vision Automation Conf. 1990*, pp. 13-56—13-67.

Hogan87 Hogan, N., "Stable execution of contact tasks using impedence control," *Proc. International Robots & Vision Automation Conf. 1987.*

Hollerbach82 Hollerbach, J. M., "A recursive formulation of manipulator dynamics and comparative study of dynamics formulation and complexity," *Robot Motion*, Brady *et al.*(Eds.), MIT press, Cambridge, Massachusetts, 1982, pp. 73—87.

Horning91 Horning, R. J., Forsyth, M., Yetter, J., and Thayer, L. J., "How ICs impact workstations," *IEEE Spectrum*, April 1991, pp. 58—68.

Hsu89 Hsu, P., "Control of Multi-manipulator Systems—Trajectory Tracking, Load Distribution, Internal Force Control, and Decentralized Architecture," *Proc. IEEE Int. Conf. on Robotics and Automation 1989*, pp. 1234—1239.

Iacobovici88 Iacobovici, A., "A Pipelined Interface for High Performance Floating Point Performance with Precise Exceptions," *Micro*, June 1988, pp. 77—87.

IEEE87, "Standard 1296-Standard for a Full-Feature 32-Bit Backplane Bus," *IEEE CS Press*, 1987.

IEEE88, "Standard 1196-Standard for a Simple 32-Bit Backplane Bus," *IEEE CS Press*, 1988.

IEEE/ANSI87, "Standard 1014-Versatile Backplane Bus: VMEbus," *IEEE CS Press*, 1987.

IEEE/ANSI89, "Draft P896.1–Standard for FutureBus+," *IEEE CS Press*, 1989.

Intel87, "80386/80387 Hardware Reference Manual," Intel, 1987.

Intel88, "80960KB Hardware Designer's Reference Manual," Intel, 1988.

Intel90, "i486 Programmer's Reference Manual," Osborne McGraw-Hill, 1990.

Intelledex90, "Specifications Model 2400 series III SCARA Robot," Intelledex, Corvalis, Oregon, July 1990.

Izaguirre91 Izaguirre, A., Hashimoto, M., Paul, R. P., Hayward, V., "A New Computational Structure for Real-time Dynamics," *International Journal of Robotics Research*, In Press.

Javaheri87 Amin-Javaheri, M., Orin, D. E., "A systolic architecture for computation of the manipulator invertia matrix," *Proc. IEEE Int. Conf. on Robotics and Automation 1987*, pp. 647—653.

Johnson91 Johnson, M., Superscalar Microprocessor Design, Prentice Hall, Englewood Cliffs, New Jersey, 1991.

Kalbfleisch91 Kalbfleisch, C. W., "Overview of Real-Time Kernels at the Superconducting Super collider Laboratory," Superconducting Super collider Laboratory, Dallas, Texas, 1991.

Kanade84 Kanade, T., Khosla, P., and Tanaka, N., "Real-time control of CMU Direct-Drive Arm II Using Customized Inverse Dynamics," *Proc. 23rd IEEE CDC*, December 1984, pp 1345—1352.

Kane88 Kane, G., MIPS RISC Architecture (R2000/R3000), Prentice Hall, Englewood Cliffs, New Jersey, 1988.

Kasahara85 Kasahara, H., Narita, S., "Parallel processing of robot-arm control computation on a multi-microprocessor system," *IEEE Journal of Robotics and Automation*, vol. RA-1, no. 2, 1985, pp. 104—113.

Kazanzides86 Kazanzides, P., Wasti, H., and Wolovich, W. A., "A Multiprocessor System for Real-Time Robotic Control: Design and Applications," *Proc. IEEE Int. Conf. on Robotics and Automation 1987*, pp. 1731—1736.

Kazerooni88 Kazerooni, H., Kim, S., "A New Architecture for Direct Drive Robots," *Proc. IEEE Int. Conf. on Robotics and Automation 1987*, pp. 442—445.

Khatib86 Khatib, O., "The Operational Space Formulation in the Analysis, Design, and Control of Robot Manipulators," *Robotics Research Third International Symposium*, MIT Press, Cambridge, Massachusetts, 1986, pp. 263—270.

Klein82 Klein, C. A., Wahawisan, W., "Use of a Multiprocessor for Control of a Robotic System", *The International Journal of Robotics Research*, vol. 1, no. 2, summer 1982.

Killough86 Killough, S. M., Martin, H.L., and Hamel, W. R., "Conversion of a Servomanipulator from Analog to Digital Control," *Pro.c IEEE Int. Conf. on Robotics and Automation 1986*, pp. 734—739.

Kircanski86 Kircanski, M., Vukobratovic, M., and Timcenko, A., "An Approach to Development of Real-Time Robot Models," *IFToMM Symposium. ROMANCY*, Krakow 1986.

Kircanski88 Kircanski, M., Vukobratovic, M., and Kircanski, N., "A new program package for the generation of efficient manipulator kinematic and dynamic equations in symbolic form," *Robotica, vol 6.*, 1988, pp. 311—318.

Kircanski89 Kircanski, N., Timcenko, A., Jovanovic, Z., "Computation of Customized Symbolic Robot Models on Peripheral Array Processors," *Proc. IEEE Int. Conf. on Robotics and Automation 1989*, pp. 1180—1185.

Kohn89 Kohn, L., Margulis, N., "Introducing the Intel i860 64-bit Microprocessor," *Micro*, August 1989, pp. 15—30.

Korein78 Korein, J. U., Bollinger, J. G., "Design Parameters for Sampled-Data Drives fo CNC Machine Tools," *IEEE Trans. Industry Applications*, May/June 1978, pp. 255—263.

Korein86 Korein, J. U., Maier, G. E., Taylor, R. H., and Durfee, L. F., "A Configurable System for Automation and Programming Control," *Proc. IEEE Int. Conf. on Robotics and Automation 1986*, pp. 734—739.

Krick91 Krick, R. F., Dollas, A., "The Evolution of Instruction Sequencing," *Computer*, April 1991, pp. 5—15.

Lathrop85 Lathrop, R. H., "Parallelism in Manipulator Dynamics," *Proc. IEEE Int. Conf. on Robotics and Automation 1985*, pp. 772—777.

Lawson90 Lawson, W. H., "Philosophies for Engineering Computer-Based Systems," *Computer*, December 1990, pp. 52—63.

Leahy86 Leahy, M. B., Saridis, G. N., "The RAL Hierarchical Control System," *Proc. IEEE Int. Conf. on Robotics and Automation 1986*, pp. 407—411.

Lee86 Lee, C. G. S., Chang, P. R., "A maximum pipelined CORDIC architecture for robot inverse kinematics computation," TR-EE-86-5, School of E.E., Purdue University, January 1986.

Lee89 Lee, I., King, R. B., and Paul, R. P., "A Predictable Real-Time Kernel for Distributed Multisensor Systems," *Computer*, June 1989, pp. 78—83

Lee90 Lee, E. A., "Programmable DSPs: A Brief Overview," *Micro*, October 1990, pp 14.—16.

Leibowitz90 Leibowitz, M. R., "UNIX Workstations Arrive!" *Datamation*, June 1, 1990, pp 24—30.

Leung86 Leung, S. S., Shanblatt, M. A., "Real-Time Direct Kinematics on a VLSI Chip," *Proc. IEEE Real-Time Systems Symposium 1986*, pp. 257—263.

Leung88 Leung, S. S., Shanblatt, M. A., "Computer Architecture Design for Robotic," *Proc. IEEE Int. Conf. on Robotics and Automation 1988*, pp. 453—456.

Leung88(2) Leung, S. S., Shanblatt, M. A., "A Conceptual Framework for Designing Robotic Computational Hardware with ASIC Technology," *Proc. IEEE Int. Conf. on Robotics and Automation 1988*, pp. 461—464

Levin87 Levin, F., Bühler, M., and Koditschek, D. E., "A Prototype Processing Cell for Distributed Real Time Control," RP-8701, Center for Systems Science, Dept. E.E., Yale University, March 1987.

Li88 Li, X., Malek, M., "Analysis of Speedup and Communication/Computation ratio in multiprocessor Systems," *Proc. IEEE Real-Time Systems Symposium 1988*, pp. 282—288

Ling88 Ling, Y. L. C., Sadayappan, P., Olson, K. W., and Orin, D. E , "A VLSI Robotics Vector Processor for Real-Time Control," *Proc. IEEE Int. Conf. on Robotics and Automation 1988*, pp. 303—308.

Lloyd88 Lloyd, J., Parker, M., and McClain, R., "Extending RCCL Programming Environment to Multiple Robots and Processors," *Proc. IEEE Int. Conf. on Robotics and Automation 1988*, pp. 465—469.

Lloyd91 Lloyd, J. E, Hayward, V., "Real-time Trajectory Generation Using Blend Functions," *Proc. IEEE Int. Conf. on Robotics and Automation 1991*.

Lozano-Pérez88 Lozano-Pérez, T., *et al.*, "Handey: A Task-Level Robot System," *Robotics Research Symposium 1988*, pp. 29—36.

LSI88, "LSI Logic—LR3010 Floating Point Accelerator (preliminary)," LSI Logic 1988.

Luh80 Luh, J. Y. S., Walker, M. W., and Paul, R. P., "On-line computation scheme for mechnical manipulators," *Trans. ASME Journal of Dyn. Syst. Meas. Cont.*, June 1984, pp. 134—142.

Luh83 Luh, J.Y.S., "Conventional Controller design for industrial robots—A tutorial," *IEEE Trans. Systems, Man, and Cybernetics*, May/June, 1983, pp. 298—316.

Mangaser89 Mangaser, A. A., Wang, Y., and Butner, S.E., "Concurrent Programming Support for a Multi-Manipulator Experiment on RIPS," *Proc. IEEE Int. Conf. on Robotics and Automation 1989*, pp. 853—859.

Manuel88 Manuel, T., "DSP Gets a Real-Time Operating System," *Electronics* (now *Electronics Week*), October 1988, pp. 137.

Marsan83 Marsan, M. A., Balbo, G., "Modeling Bus Contention and Memory Interference in a Multi-Processor System," *IEEE Trans. on Computing*, January,1983, pp. 60—72.

McMillan91 McMillan, S., Orin, D. E., and Sadayappan, P., "Real-Time Robot Dynamic Simulation on a Vector/Parallel Supercomputer," *Proc. IEEE Int. Conf. on Robotics and Automation 1991*, pp. 1836—1841.

Mendelsohn91 Mendelsohn, A., "Will Monolithic or Multichip Processors Win the Performance Race?" *Computer Design*, May 1991, pp. 100—121.

Miller90 Miller, D. J., Lennox, C. R., "An Object-Oriented Enviroment for Robot System Architectures," *Proc. IEEE Int. Conf. on Robotics and Automation 1990*, pp. 352—361.

Motorola87, "MC68881/MC68882 Floating Point Coprocessor User's Manual," Motorola Inc., November 1987.

Motorola88, "MC88100 Technical Summary," Motorola Inc., October 1988.

Motorola90, "MC68040 Reference Manual," Motorola Inc., June 1990.

Narasimhan86 Narasimham, S., Siegel, D. M., Hollerbach, J. M., Biggers, K., and Gerpheide, G., "Implementation of control methodologies on the computational architecture for the Utah/MIT hand," *Proc. IEEE Int. Conf. on Robotics and Automation 1986*, pp. 1884—1889.

Narasimhan88 Narasimham, S., Siegel, D. M., and Hollerbach, J. M., "Condor: A Revised Architecture for Controlling the Utah-MIT hand," *Proc. IEEE Int. Conf. on Robotics and Automation 1988*, pp. 446—449.

Nash85 Nash, J. G., "A systolic/cellular computer architecture for linear algebraic operations," *Proc. IEEE Int. Conf. on Robotics and Automation 1985*, pp. 779—784.

Newton89 Newton, K., "VSB Breaks Through VME Peformance Roadblocks," *Computer Design*, November 1989, pp. 96—105.

Nigam85 Nigam, R., Lee, C. S. G., "A multiprocessor-based controller for the control of mechanical manipulators," *IEEE J. Robotics Automation*, December 1985, pp. 173—182.

Nilakantan88 Nilkantan, A., Hayward, V., "Synchronizing Multiple Manipulators," *Second Int. Symposium on Robotics and Manfacturing, Research, Education, and Applications, 1988*

Nilakantan89 Nilakantan, A., Hayward, V., "The Synchronization of Multiple Manipulators in Kali," *Robotics and Autonomous Systems (Special Issue on Multi-Arm Robots)*, North Holland, vol 5., 1989.

Oehler90 Oehler, R., "RISC System/6000 Architecture and Performance," *Proc. Hot Chips Symposium*, IEEE CS Press, 1990.

Oehler91 Oehler, R., Blasgen, M. W., "IBM RISC/6000 Architecture and Performance," *Micro*, June 1991, pp. 14—17, 56—61.

Olsen89 Olsen, W. R., Dimitri, D. S., and Berg, D., "Control for Eight Axes by DSP," *Power Conversion and Intelligent Motion (Europe)*, Nov. 1989, pp. 236—240.

Orin85 Orin, D. E., Chao, H. H., Olson, K. W., and Schrader, W. W., "Pipeline/parallel algorithms for the Jacobian and inverse dynamic computations," *Proc. IEEE Int. Conf. on Robotics and Automation 1985*, pp. 785—789.

Orin86 Orin, D. E., Tsai, Y. T., "A real-time computer architecture for inverse kinematics," *Proc. IEEE Int. Conf. on Robotics and Automation 1986*, pp. 843—850.

Ostroff87 Ostroff, J.S., Wonham, W.M., "Modelling, Specifying and Verifying Real-Time Embedded Computer Systems," *Proc. IEEE Real-Time Systems Symposium 1987*, pp. 124—132.

Papamichalis88 Papamichalis, P., Simar, R., "The TMS320C30 Floating-Point Digital Signal Processor," *Micro*, December 1988, pp. 13—29.

Paul72 Paul, R. P., "Modeling, Trajectory Calculation and Servoing of a Computer Controlled Arm," AIM 177, Standford Artificial Intelligence Laboratory, Stanford University, 1972

Paul81 Paul, R. P., Robot Manipulators: Mathematics, Programming and Control, MIT Press, Cambridge, Massachusetts, 1981.

Paul85 Paul, R. P., Zhang, H., "Robot Motion trajectory specification and generation," *Robotics Research: The Second International Symposium*, H. Hanafusa and H. Inoue (Eds.), MIT Press, Cambridge, Massachusetts, 1985, pp. 373-380.

Paul86 Paul, R. P., Zhang, H., "Design and Implementation of a Robot Force/Motion Server," *Proc. IEEE Conf. on Robotics and Automation 1986*, pp. 1878—1883.

Piepho89 Piepho, R. S., Wu, W. S., "A Comparison of RISC Architectures," *Micro*, August 1989, pp. 51—61.

Przytula88 Przytula, K. W., Nash, J. G., "A Special Purpose Coprocessor for Robotics and Signal Processing," *IEEE Workshop on Special Computer Architectures for Robotics*, April 1988, pp. 74—82.

Raibert81 Raibert, M. H., Craig, J. J., "Hybrid Position/Force Control of Manipulators," *Trans. ASME J. of Dyn., Sys., Meas., and Control*, June 1981, pp. 102:126—133.

Ramaritham88 Ramamritham, K., "Real-Time System Support for Robotics," *IEEE Workshop on Special Computer Architectures for Robotics*, April 1988, pp. 149—160.

Randell69 Randell, B., "A Note on Storage Fragmentation and Program Segmentation," *Communications of the ACM*, July 1969, pp. 365—372.

Rosing90 Rosing, W., "Technology Trends of the 1990's," *UNIX Review*, February 1990, pp. 82—89.

Salkind89 Salkind, L., "The SAGE Operating System," *Proc. IEEE Int. Conf. on Robotics and Automation 1989*, pp. 860—865.

Salisbury80 Salisbury, J. K., "Active Stiffness Control of a Robot Manipulator in Cartesian Coordinates," *Proc. IEEE Inter. Conf. on Decision and Control*, 1980.

Scheinman88 Scheinman,V., "Robotworld: A Multiple Robot Vision Guided Assembly System," *Robotics Research*, 4th ed., R. Bolles and B. Roth (Eds.), MIT Press, Cambridge, Massachusetts, 1988, pp. 23—28.

Schmitz89 Schmitz, D., Khosla, P., Hoffman, R., and Kande, T., "CHIMERA: A Real-time Programming Environment," *Proc. IEEE Int. Conf. on Robotics and Automation 1989*, pp. 846—852.

Schwan85 Schwan, K., Bihari, T., Weide, B. W., and Taulbee, G., "GEM· Operating System Primitives for Robots and Real-Time Control," *Proc. IEEE Int. Conf. on Robotics and Automation 1985*, pp. 807—813.

Schwan86 Schwan, K., Bo, W., and Gopinath, P., "A High-Performance, Object-Based Operating System for Real-time, Robotics Applications," *Proc. IEEE Real-Time Systems Symposium 1986*, pp. 147—156.

Seiko90, "RT 3200 High Performance Precision Robot," Seiko Inc., Torrence, California, June 1990.

Sha91 Sha, L, Rajkumar, R., and Lehoczky, J. P., "Real-Time Computing with IEEE Futurebus+," *Micro*, June 1991, pp. 30—33, 95—100.

Shalom88 Ish-Shalom, J., Kazanzides, P., "SPARTA: Multiple Signal Processors for High-Performance Robotic Control," *Proc. IEEE Int. Conf. on Robotics and Automation 1988*, pp. 284—290.

Singh91 Singh, I., "Inder Singh on: Posix," *Computer Design*, March, 1991, pp. 23—25.

Sohie88 Sohie, G. R., Kloker, K. L., "A Digital Signal Processor with IEEE Floating-Point Arithmetic," *Micro*, December, 1988, pp. 49—67.

Stewart89 Stewart, D. B., Schmitz, D. E., and Khosla, P. K., "CHIMERA II: A Real-Time UNIX-Compatible Multiprocessor Operating System for Sensor-based Control Application," CMU-RI-TR-89-24, Advanced Manipulators Lab, The Robotics Institute Carnegie Mellon University, September 1989.

Stone87 Stone, H. S., High-Performance Computer Architecture, Addison-Wesley Co., Reading, Massachusetts, 1987.

Stroustrup87 Stroustrup, B., The C++ Programming Language, Addison-Wesley Co., Reading, Massachusetts, 1987.

SUN87 ,"The SPARC Architecture Manual Rev. 5," Sun Microsystems Inc., February, 1987.

SUN89, "The SBUS Specification Rev A.2," Sun Microsystems Inc., November, 1989.

Takanashi89 Takanashi, N., Ikeda, T., and Tagawa, N., "A High-Sample-Rate Robot Control System Using a DSP Based Numerical Calculation Engine," *Proc. IEEE Int. Conf. on Robotics and Automation 1989*, pp. 1168—1173.

TI89, "Third-Generation TMS320 User's Guide," Texas Instruments, October 1989.

TI91, "TI32C40 Data Sheet," Texas Instruments, March 1991.

Unimation80 *Unimate Puma Robot Volume I* — Technical Manual 398H1, Unimation Inc., Danbury, Connecticut, April 1980.

Vischer88 Vischer, D., and Khatib, O., "Design and Development of Torque-Controlled Joints," *Experimental Robotics 1*, V. Hayward and O. Khatib (Eds.), *Lecture Notes in Control and Information Science 139*, Springer Verlag, 1988, pp. 271—286.

Volder59 Volder, J. E., "The CORDIC Trigonometric Computing Technique," *IRE Transactions on Electronic Computers*, September 1959, pp. 330—334.

Volz84 Volz, R.N., Mudge, T.N., "Robots Are (Nothing More than) Abstract Data Types," *Robotics Research: The Next Five Years and Beyond*, August, 1984.

Wang87 Wang, Y., Butner, S. E., "A New Architecture for Robot Control," *Proc. IEEE Int. Conf. on Robotics and Automation 1987*, pp. 664—670.

Wang89 Wang, W. S., Chen, K. K., Lai, Y. S., and Liu, C. H., "Implementation of a Multiprocessor system for Real-Time Inverse Dynamics Computation," *Proc. IEEE Int. Conf. on Robotics and Automation 1989*, pp. 1174—1179.

Weicker90 Weicker, R. P., "An Overview of Common Benchmarks," *Computer*, December 1990, pp. 65—75.

Whang90 Whang M., Kua, J., "Join the EISA Evolution," *Byte*, May 1990, pp. 241—247.

Whittaker90 Whittaker, W. L., Kanade, T., "Japan robotics for unmanned space exploration," *IEEE Spectrum*, December 1990, pp. 64—67.

Williams90 Williams, T., "Real-time operating systems struggle with multiple tasks," *Computer Design*, October 1990, pp. 92—108.

Williams90(2) Williams, T., "BIOS Standard Would Link Real-Time Kernels To New Hardware," *Computer Design*, May 1990, pp. 24—26.

Williams91 Williams, T., "RISC Instruction benchmarks Spark Performance Debate," *Computer Design*, May 1991, pp. 69—72.

Williams91(2) Williams, T., "Transputer Performance boosted 10x that of its Predecessor," *Computer Design*, May 1991, pp. 36—38.

Wind87 , "VxWorks Reference Manual & Documentation version 3.2," Wind River Systems, October 1987.

Zhang88 Zhang, H., Paul, R.P., "Robot Manipulator Control and Computational Cost," *IEEE Workshop on Special Computer Architectures for Robotics*, April 1988, pp. 28—65.