Dynamic Reconfigurable Machine Tool Controller

Wei Li
Brigham Young University - Provo

DYNAMIC RECONFIGURABLE MACHINE TOOL CONTROLLER

by

Wei Li

A dissertation submitted to the faculty of

Brigham Young University

In partial fulfillment of the requirements for the degree of

Doctor of Philosophy

Department of Mechanical Engineering

Brigham Young University

April 2005

BRIGHAM YOUNG UNIVERSITY

GRADUATE COMMITTEE APPROVAL

of a dissertation submitted by

Wei Li

This dissertation has been read by each member of the following graduate committee and by majority vote has been found to be satisfactory.

Date	W. Edward Red, Committee Chair
Date	C. Greg Jensen
Date	Carl D. Sorensen
Date	Robert H. Todd
Date	Timothy W. McLain

BRIGHAM YOUNG UNIVERSITY

As chair of the candidate's graduate committee, I have read the dissertation of Wei Li in its final form and have found that (1) its format, citations, and bibliographical style are consistent and acceptable and fulfill university and department style requirements; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the graduate committee and is ready for sub-mission to the university library.

Date	W. Edward Red Chair, Graduate Committee

Accepted for the Department

Matthew R. Jones
Graduate Coordinator

Accepted for the College

Douglas M. Chabries
Dean, Ira A. Fulton College of Engineering and Technology

ABSTRACT

DYNAMIC RECONFIGURABLE MACHINE TOOL CONTROLLER

Wei Li

Department of Mechanical Engineering

Doctor of Philosophy

This dissertation presents a dynamic reconfigurable control strategy based on the Direct Machining And Control (DMAC) research at Brigham Young University. A reconfigurable framework is proposed which will allow a machine tool to be controlled by a variety of applications and control laws. This Reconfigurable Mechanism for Application Control (**RMAC**) paradigm uses a hierarchical architecture to configure a mechanism into a device driver for direct control by an application like CAD/CAM. The RMAC paradigm is one of a mechanism device driver assigned to each mechanism class or model, and uses only the master model to control the mechanism. The traditional M&G code language is no longer necessary since motion entities are passed directly to the mechanism.

The design strategy of using dynamic-link libraries (DLL) to form a mechanism ***device driver*** permits a mechanism to assume different operating configurations,

depending on the number of axes and machine resolution. For example, the machine can perform as a material removal machine in one instant, and then, by loading a new device driver, act as a Coordinate Measuring Machine (CMM). This strategy is possible because RMAC is a software and networked-based control architecture. Both the CAD/CAM planning software and the real-time control software reside on the same PC. The CAM process plan can thus directly control the machine without need for process plan decomposition into the forms supported by the controller.

The architectural framework is explained in detail and the methodology for control software reconfiguration into a device driver is presented. For demonstration purposes two device drivers are implemented on a prototype machine to demonstrate feasibility and usefulness.

ACKNOWLEDGEMENTS

I would like to thank my graduate advisor, Dr. Red, for all his support and patience through the past four and half years. I would like to thank the other members of my graduate committee, Dr. Jensen, Dr. McLain, Dr. Sorensen, and Dr. Todd, for their excellent support and encouragement. I would also like to thank the members of the Direct Machining and Control research group at Brigham Young University.

I owe a special debt of gratitude to my parents and family. They have always been there to offer their support and encouragement. I would like to express appreciation to all of my friends who have given help and support during this endeavor.

I owe a special thanks to the English editors, Lynn Holm and Brooke Barker, for their great effort to edit and refine my English writing.

Finally, I would like to thank Brigham Young University, especially the department of Mechanical Engineering. I have learned greatly in the past four and half years to be a better person. With the education I received from here, I have laid a good foundation for my future professional career.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1 INTRODUCTION

1.1 Statement of the Problem

Historically, manufacturing systems have passed certain distinct phases. In each phase, machine tools and their controllers are used by manufacturing enterprises quite differently. The differences between machine tools and their controllers during these manufacturing phases have been caused by differences in the available technologies and the variation in customer demands. We are now at the embryonic stage of a revolutionary new phase. Dedicated manufacturing systems are behind us and flexible manufacturing systems show more and more limitations; manufacturing systems of the future will be reconfigurable.

As technologies and customers demand greater efficiency and sophistication, machine tools and their controllers used in current manufacturing systems must keep pace. This dissertation will propose and develop a new reconfigurable direct machine tool controller paradigm to address the problems existing in today's machine tools and controllers.

1.1.1 Dedicated manufacturing systems (DMS)

Dedicated machine tools and controllers were widely used among manufacturing enterprises before the first Numerically Controlled (NC) machine was invented. During that time, most machine tools and controllers were purely mechanical or electromechanical systems. The major disadvantage of these systems was that each machine tool and controller was tailored for a special product. As a result, the function of a dedicated machine tool controller could not be changed or upgraded without great difficulty. As customer demands for different products changed over time, manufacturing enterprises often had to replace the dedicated machine tools and controllers to accommodate this demand.

1.1.2 Flexible manufacturing systems (FMS)

The invention of Numerically Controlled [NC] machines and their subsequent evolution (i.e., Computer Numerical Control [CNC], Distributed Numerical Control [DNC]) dramatically changed manufacturing. CNC, together with Computer Aided Design (CAD) and Computer Aided Manufacturing (CAM), have become core technologies in flexible manufacturing systems (FMS). These technologies have drastically changed the way parts are designed and manufactured. What was once a manual process in dedicated manufacturing systems has largely been transformed into a paperless digital process.

CNC machines and controllers have brought many benefits into manufacturing systems by improving production rates, product quality, product accuracy, and machine control accuracy. Meanwhile, the manufacturing flexibility has been increased over the dedicated machines.

Despite the advantages of CNC systems, there are two distinct drawbacks in current CNC machines and their controllers that limit the implementation of new technologies.

1. Even though the first CNC machine tools were developed about fifty years ago, CNC machine tools are still programmed today using the decades-old instruction code called M&G code. M&G code is a collection of ASCII code generated from a post-processor running independently from CAD/CAM software. It is formatted specifically for a machine controller and different M&G variations are often not interchangeable. To operate a CNC machine tool today, part geometries and their process instructions contained within CAD/CAM systems must be decomposed into the forms required for each machine's controller. There is no direct link between CAD/CAM software and machine tool controllers. The process of generating M&G codes and feeding them into machine tool controllers is tedious, inefficient, and error-prone. More importantly, this old process is a bottleneck to further improving the CNC machining production rate, quality and flexibility.
2. Over the past half century, many machine tool companies have attempted to build an ideal machine tool. But most machine tool controllers are proprietary and their architecture is closed. Vendors may add different dialects and vendor-specific syntax into the M&G codes; thus making their machining codes incompatible with other controllers. Under this old paradigm, a single vendor would provide the entire controller. Once these controllers were built and delivered to end

customers, it was extremely difficult for the customer or third party developers to upgrade the machine tool with customized functionalities. The machine tool controllers function as a *black box*; thus, end users have limited or no access to their internal control algorithms or hardware.

1.1.3 Reconfigurable manufacturing systems (RMS)

Because of these problems, there has been a worldwide effort in the past decade, from industry as well as from many research institutions, to propose developing a new architecture for open control. This new wave of research is aimed at developing open-architecture control systems that will enable modular and reconfigurable manufacturing systems.

Koren [1] proposed a reconfigurable manufacturing system (RMS) in 1999. He noted the deficiencies of existing CNC machine tools and controllers, which include lack of interchangeability, modularity, extensibility, and reconfigurabililty. He predicted that a new generation of reconfigurable machine tools, based on an open-architecture controller with adjustable modular structure, will come into existence in the next decade and will be the cornerstone of the RMS.

During the last few years, two enablers for reconfigurable machine tools have emerged: in machine hardware, modular machine tools that offer end customers more machine options [2]; and, in control software, modular, open-architecture controllers that use reconfigurable control software. These emerging technologies will stimulate the design of control systems with reconfigurable hardware and software.

1.2 Direct Machining And Control (DMAC)

Beginning in 1998, the Direct Machining And Control (DMAC) research group at Brigham Young University has been developing an open-architecture controller [39-47] that directly interfaces to application software like CAD/CAM (see Fig. 1.1).

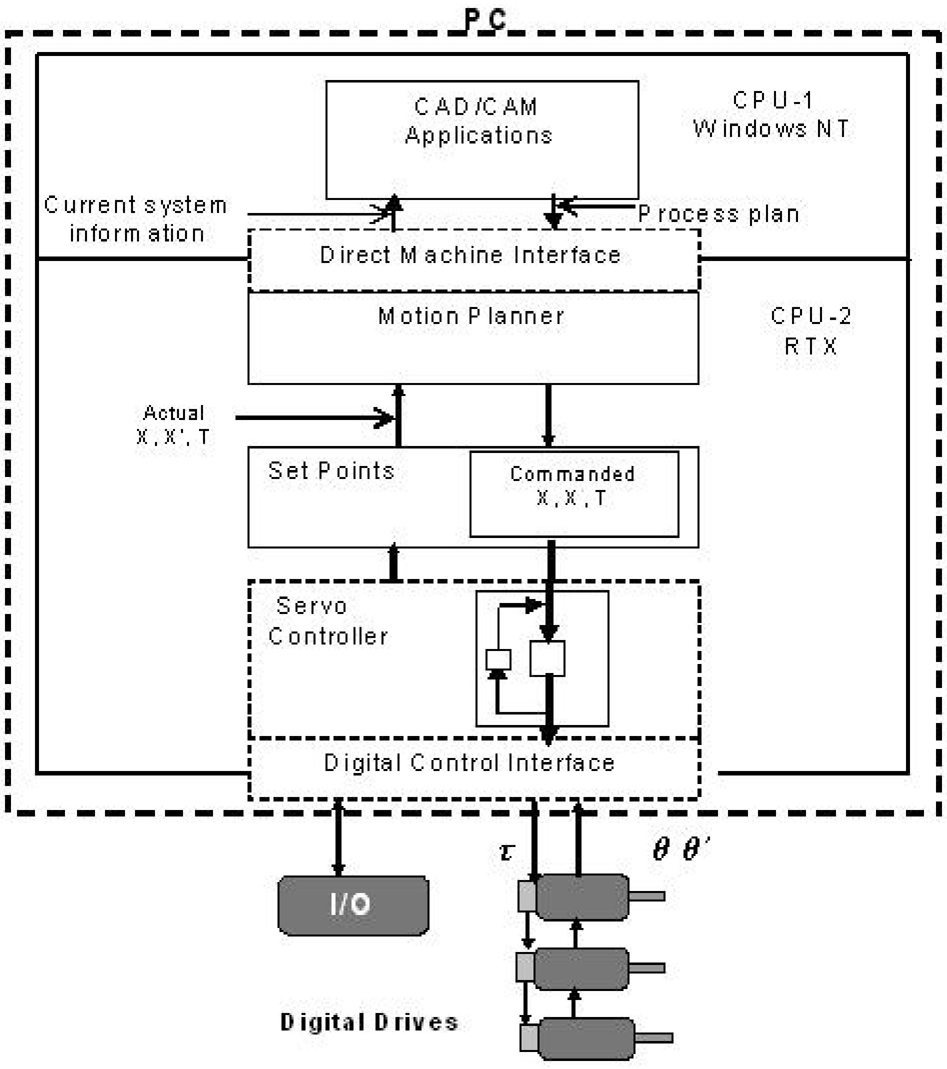


Fig. 1.1 Current DMAC architecture

The DMAC controller is a truly software-based controller and all control components, such as motion and servo control, are defined and developed in object oriented C++ code. The DMAC architecture is configured on a dual-processor platform. One processor runs non real-time Windows applications, such as CAD/CAM and Human Machine Interface (HMI). The second processor runs real-time control applications, such as motion planning control and servo-loop control. A direct machining interface is developed to allow communication between the real-time and non real-time applications. The DMAC architecture is designed to be independent of the interface to the control hardware and thus can control both machine tools and robots.

1.3 Reconfigurable Mechanism for Application Control (RMAC)

With this advanced DMAC control system in place at Brigham Young University, this dissertation proposes a more flexible and reconfigurable control architecture. The Reconfigurable Mechanism for Application Control (RMAC) architecture in Fig. 1.2 is developed to allow for machine tools to be controlled like *part printing* devices. This reconfigurable control architecture is a hierarchical and modular software structure that can be dynamically reconfigured for direct control, with each software module designed and built independently. The collection of modules necessary to enable a CAD/CAM process plan to directly control a machine is called a mechanism device driver. A mechanism can be reconfigured to perform differently by simply loading a different device driver for the mechanism.

All control software modules and their interfaces are specified in a well-defined manner. A set of interface APIs (Application Programming Interface) are provided for each software module, thus allowing for control and feedback information flow among these various modules. Under the RMAC paradigm, various mechanism devices are connected directly to CAD/CAM systems through different device drivers. Each device driver is designed as separate software module and is able to map the mechanism's configurations and capabilities to the manufacturing process intent of a CAD/CAM process plan, thus allowing a CAD/CAM process planner to make run-time decisions to choose optimal machines to fulfill different manufacturing process requirements. The static DMAC open-architecture controller is thus replaced with a more flexible and reconfigurable RMAC controller that can be dynamically reconfigured for different machine tools or control applications.

The current DMAC architecture is insufficient due to the following limitations:

1. DMAC was built with one software control solution and connected to a CAD/CAM system.
2. The current DMAC implementation can not dynamically reconfigure a single machine to operate differently. Each DMAC-compliant machine has one behavior. For instance, a milling machine cannot be operated as a CMM.
3. The DMAC controller is not generic. Each DMAC controller is tailored for a specific machine tool or control application; thus, lacks the flexibility to dynamically vary its functionality for different machine tools or control applications. For instance, a three-axis mill controller cannot be used to control a five-axis machining center.
4. The part printer paradigm requires a device driver architecture that does not exist in the current DMAC architecture.

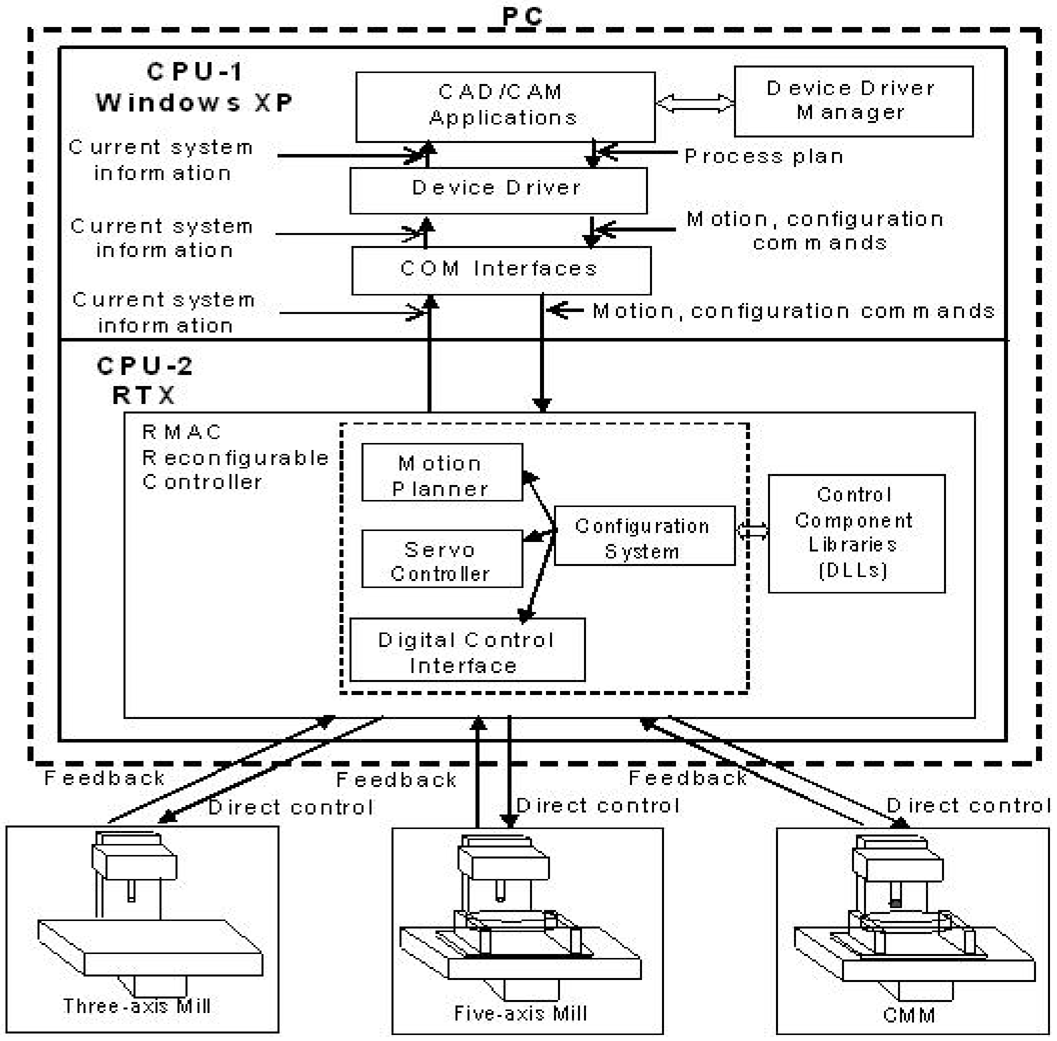


Fig. 1.2 RMAC architecture

RMAC overcomes these limitations in the current DMAC architecture with the following architectural improvements:

1. RMAC is designed for more generic software solutions; thus, it is reconfigurable for different machines, control solutions, and CAD/CAM systems.

2. The RMAC architecture contains a device driver manager that allows CAD/CAM users to select an optimal machine tool. A built-in database search engine allows

users to easily and quickly narrow down their machine selections and then locate a relevant mechanism device driver.

3. The RMAC architecture contains a generic device driver architecture that allows for part printer paradigm. Any machine-specific configurations and capabilities, such as machine limits, maximum federate, etc, are built into a mechanism device database and are directly accessible to the device driver software. The device driver has standard driver interface and APIs to communicate with CAD/CAM systems and the machine open-architecture controller. As a result, various CAD/CAM systems and machine tools can be connected to the device drivers through the same driver interface and APIs. By loading relevant device drivers, RMAC allows for reconfiguring a single machine to operate differently.
4. The RMAC architecture contains a generic and reconfigurable open-architecture controller. This RMAC reconfigurable controller contains the generic control codes that are applicable to various machine tools and control applications. Any mechanism-specific control codes are designed and built as separate *dynamic-link libraries* (DLLs). Thus, RMAC open-architecture controller can be reconfigured to apply on different machine tools.
5. Under the RMAC architecture, a configuration system is developed to allow the run-time mapping of any mechanism-specific control codes from the relevant DLLs into the RMAC reconfigurable controller for the selected machine tool.

The RMAC paradigm provides new opportunities for manufacturing organizations and machine tool end users. Manufacturing enterprises can introduce greater flexibilities into their manufacturing systems. Fig. 1.2 shows how one machine tool can be operated differently. If the manufacturing operations need a three-axis mill, the user loads a three-axis mill device driver prior to machining. But if products or customer demands change over time, such that manufacturing operations require a five-axis mill, it may be necessary to add a two-axis rotary table. RMAC provides the user a relevant five-axis mill device driver, so the same machine can be commanded as a five-axis mill. Once all the parts are made, by adding a measurement probe and loading a relevant CMM driver, the machine can be commanded as a CMM machine to inspect the parts during their manufacturing. These flexibilities cannot be realized with any conventional machine tools or even with the current DMAC controller.

1.4 Research Objectives

The objectives of this research are then to propose, develop, and demonstrate an architecture for a dynamic reconfigurable machine tool controller using the direct control and device driver paradigms. Specifically, the research objectives are to (1) develop a generic and reconfigurable control architecture that would allow direct control to be easily reconfigured for different machines, control applications, and CAD/CAM systems; (2) develop a configurable device driver architecture so that a CAD/CAM process would be mapped into an appropriate machine, thus allowing for the mathematical CAD model to drive the connected machine tool directly without tessellating into thousands of line and arc segments ; (3) develop standardized device driver interface and a set of interface APIs so that All CAD/CAM packages would connect to a standard driver software interface, and all machine tools accept that driver interface through the RMAC reconfigurable controller; and (4) demonstrate the reconfigurable control architecture on a prototype mill.

Achieving these design objectives requires a reconfigurable controller to posses the following general characteristics:

1. **Modularity**: In a reconfigurable controller, all machine-specific software components should be modular (e.g., kinematics, machine actuator mapping, servo control, I/O interface, etc). These software modules should be designed independently into separate *dynamic-link libraries* (DLLs) so that they can be easily added to the controller, removed from the controller, or replaced by other modules during system reconfiguration.
2. **Portability**: A reconfigurable controller should be vendor-neutral so that end users can easily integrate new machine hardware or software from any third party vendors.
3. **Customization**: A reconfigurable should be flexible enough to allow end users to integrate customized control modules with the aid of open-architecture technology, providing the exact control functions that end users need.
4. **Run-time reconfigurability**: A desired dynamic reconfigurable machine should be reconfigurable at run-time without shutting down the machine tool.
5. **Verifiability**: A reconfigurable controller should enable end users to verify its functionality upon system reconfiguration.

1.6 Outline of Dissertation

I. **Introduction**

Chapter one introduces the objective and contribution of this dissertation and defines the research scope.

II. **Literature review**

Chapter two reviews the past and present research on open-architecture controllers and some more recent research projects on reconfigurable control systems. The research of Direct Machining And Control (DMAC), which is the foundation platform for the proposed reconfigurable controller, is also reviewed.

III. **RMAC software architecture**

Chapter three presents an overall architecture for a dynamic reconfigurable machine tool controller.

IV. **Methodology**

Chapter four describes in greater details for each software module and interface defined within the RMAC architecture. It then presents the general methodology for reconfiguring the RMAC controller.

V. **Prototype implementation**

Chapter five first shows the implementation of a dynamic reconfigurable controller on a three-axis tabletop mill by developing a machine device driver specific to this mill. It then shows the proposed implementation of this research on a Tarus five-axis full-size mill, and a Coordinate Measuring Machine (CMM). Finally, it presents the simulation of the RMAC controller on a commercial CAD/CAM program.

VI. **Results**

Chapter six presents the experimental results of the prototype developed as explained in chapter four.

VII. **Summary and Recommendations**

Chapter seven summarizes this research and gives some recommendations for future research.

CHAPTER 2 LITERATURE REVIEW

This chapter reviews research related to this dissertation, including research in the field of open-architecture control systems, reconfigurable control systems, and reconfigurable robot systems. A modernized machining code standard, called STEP-NC, is also reviewed. Finally, the Direct Machining And Control (DMAC) architecture, the foundation architecture for this dissertation, is also reviewed.

2.1 Related Research

2.1.1 Open-architecture control (OAC) system

In the past decade, there has been a growing demand from machine tool end users, as well as from machine tool manufacturers, to open the current proprietary control systems. A new concept of open-architecture control was proposed and introduced in both industry and academia. This new type of open-architecture control is a necessary enabler for integrated CAD/CAM and sensor-based control.

Three active industrial consortiums, the OSE (Open System Environment for controller) [3] of Japan, the OSACA (Open System Architecture for Controls within Automation systems) [5, 6] of Europe, and the OMAC (Open Modular Architecture Controllers) [4] consortium of the U.S., define and promote the use of open-architecture controllers to replace the older, closed CNC systems. Their objectives consist of defining and developing a set of APIs that enable control vendors to supply standard components. These components are then delivered to the machine tool suppliers to be integrated into different control systems, and the integrated control systems and machines are finally delivered to end users to satisfy their specific needs.

In academia, several research projects were undertaken to open CNC control. One of the earliest research projects in open-architecture control was the Next Generation workstation/machine Controller (NGC) [7] in 1989, sponsored by the US Air Force. The goals for the NGC program were to provide a commercial version of an expanded machine tool environment that would integrate CAD/CAM and sensor-based machining.

One of the first large-scale research initiatives was done by Wright et al. [8, 9] in 1988. They proposed the MOSAIC (Machine Tool Open System Advanced Intelligent Controller) architecture, in which a real-time version of UNIX is chosen as the operating platform and VME bus is used as the de-facto communication bus that can communicate the machining information to the controller. This group of researchers coined the term "open-architecture controller". In a parallel effort to the open system in the PC industry, these researchers envisioned that by using industrial PC as the control hardware basis, machine tool industry can open the current closed controller architecture so that different hardware and software vendors could work on different elements of the control system and integrate their products into a seamless robust controller.

Koren et al. [10], in the Engineering Research Center for Reconfigurable Machining System at the University of Michigan, proposed an open CNC system, named UMOAC, which allows interchanging motion control tasks as a feature of reconfigurability. The UMOAC architecture is designed in a distributed platform: the HMI and motion control runs in the main controller, while the servo control runs on a DSP board that communicates with main computer via VME bus or any other network protocol such as TCP/IP. A common Windows-based HMI API is defined for different CNC systems. The UMOAC is also designed to be used on their reconfigurable machine tool [11].

Yellowley et al. [12] at the University of British Columbia proposed and developed a UBC open-architecture controller. The National Institute of Standards and Technology (NIST) [13] applied the NGC open-architecture framework into its Enhanced Machine Controller (EMC) project. The EMC offered real-time, open-architecture control based on open source and community software development, and was suitable for a variety of machines, including machine tools, robots, and Coordinate Measuring Machine (CMM). There are a number of other researchers [14-17] who applied the principle of open-architecture control to different control applications.

2.1.2 Reconfigurable control system

More recently, with open-architecture control as a basis, some researchers have gone one step further, proposing to develop reconfigurable machine tool controllers in which the same machine tool controller can be reconfigured to control different types of machine tools. This in turn will allow end users to have even more flexible control systems on their factory floors.

One of the first large-scale initiatives was launched by the European Union (EU) in the early 1990s. In a European Union-sponsored report [18] a strategy was outlined to ensure the long-term survivalability of the European machine tool industry. This report stressed the need for machine tools to be designed and built modularly, allowing machine tool manufacturers to specialize in particular modules instead of complete systems. System integrators could then build complete systems from the modules according to end users' specific needs. This strategy requires splitting a machine tool into a set of autonomous functional units that can be "plug-and-play" interfaced to form complete systems for particular customers' needs.

Several European projects are currently under development to achieve this design goal. The European MOSYN (Modular Synthesis of Advanced Machine Tools) project [19], lead by the Hannover University, looks at customer-specific configurations of modular machine tools. The Reconfigurable Machining Systems [20] of the Special Research Program (SRP) 467, sponsored by the German Research Foundation, are aimed at developing models for structuring and configuring reconfigurable manufacturing systems (RMS). To be reconfigurable, well-defined interface layers and concepts for functional units as modules of RMS are introduced and under development.

In the U.S., the Engineering Research Center of Reconfigurable Machining Systems (ERC/RMS) was founded at the University of Michigan in 1996. Koren et al., from ERC/RMS [11], presented "Reconfigurable Machine Tools", in which they developed new machine tools whose mechanical configurations and software-based open controllers [10] can both be reconfigured at run-time.

Altintas et al. [18] presented an open and reconfigurable modular tool kit as a design tool for future machine tools and machining monitoring systems. In their system, they used a real-time preemptive operating system (ORTS) for machine-level real-time tasks and an enhanced Windows-NT-based environment, running on a PC, for applications such as HMI. The motion control boards were off-the-shelf DSP boards, running under ORTS, and had built-in algorithms that cannot be interchanged or modified externally by end users. This limited the implementation of any new advanced motion control algorithms. Rather than using a graphic tool to reconfigure their controller, the reconfiguration of their machine tools and controller was accomplished by running a series of script commands. Due to the nature of the script language, reconfiguring their controller at run-time is not easy or user-friendly.

Birla [21] presented a reconfigurable machine tool controller in his Ph.D. dissertation "Software Modeling for Reconfigurable Machine Tool Controller" in 1997. He used two well-known computer science paradigms to define all controller components, object-oriented programming (OOP) and finite state machine (FSM). All these components were designed to be reusable, scalable, and portable. A component library was developed from which the control components could be selected and reconfigured into a control system. Similar research was also undertaken by S. Wang and K.G. Shin [22], who proposed a reconfigurable software architecture for machine control systems. One limitation with Birla's work was that he did not fully implement his work with a graphic configuration tool and there was no simulation tool available to validate the control system upon the controller reconfiguration.

Similar works on reconfigurable control systems can also be found from S. Kolia et al. [24], S. Birla et al. [25], and D. Kalita et al. [27].

2.1.3 Reconfigurable robot system

Another research area that is related to this dissertation is in the field of reconfigurable robot systems. A pioneer research project in reconfigurable robot systems is the *Chimera RTOS Project* [26] in the Advanced Manipulators Laboratory at Carnegie Mellon University (CMU). The Chimera architecture is based on port-based objects (PBO), which are similar to component-based objects. The objectives of the Chimera project are to develop a control architecture that will support reconfigurable robots, integrated sensor control, dynamic controller reconfiguration, and collaboration through code sharing. In the Chimera architecture, the entire control system is viewed as an interconnection of components forming a system configuration that will provide an exact system response. Each component is defined as a port-based object with some input and output ports. A graphic software assembly tool is used to configure the robotic manipulator system at run-time and a PBO library is developed and is available to the system integrators for run-time control system reconfiguration.

Zhang et al. [28], at Xerox Palo Alto Research Center, developed software architecture for Modular Self-Reconfigurable Robots. Their software architecture is a multi-master/multi-slave structure running in a multi-threaded environment. The architecture is implemented on a Motorola PowerPC under the real-time operating system vxWorks. The master controllers are responsible for motion planning, synchronizing slave controllers, and reconfiguring slave controllers. Based on the different motion and configuration requirements, the slave controllers can reconfigure their software modules at run-time. The communications between master and slave controllers are through a CANBus.

In the past few years, I.M. Chen [29], K. Feldmann and M. Wenk [30], and W.J. Schonlau [31] have conducted similar research on reconfigurable robot software architectures.

2.1.4 Summary of the past research

The current state of open-architecture and reconfigurable machine controllers has evolved from a number of diverse development efforts. The design goal of these research projects is to develop a vendor-neutral, tool-neutral, and controller-neutral architecture. The resulting architectures represent a wide range of design strategies and solutions. However, despite their differences, there are some commonalities and prevailing trends that are shared by all of these previous development efforts.

Most of the proposed open-architecture and reconfigurable machine controllers use Windows as the operating platform. As Windows has become the de facto operating system (OS) in the PC industry, more and more control vendors choose Windows as their control software OS platform.

A prevailing trend that can be found in these control architectures is that the control systems are becoming more software-based. All of these control architectures use either object-oriented codes or component-based languages to define their control modules. Software-based control architecture has made the entire control system very flexible, highly modular, and easy to upgrade.

More control architectures use a dual-processor platform, where one processor runs non real-time Windows application program and the other processor runs a real-time operating system such as VenturCom RTX, VxWorks, QNX, or a real-time extension of Windows to do real-time motion, servo, and I/O control. There are a growing number of design strategies that have adopted distributed control solutions, where the server side controller runs application programs such as CAD/CAM and HMI while the client side controller runs motion, servo and I/O control. Until a hard real-time network protocol is developed, this distributed control solution will have difficulties satisfying the hard real-time constraints of machine tool controllers.

Even though significant research has been made into open-architecture and reconfigurable control systems, and a wide variety of design strategies and solutions that have been proposed, these developed architectures are still insufficient because of several major limitations.

First, even though these development efforts apply open-architecture principles to enable machine end users to gain greater access to proprietary internal control algorithms, these control architectures still rely on machine-dependent M&G codes. Thus, these so-called open control and reconfigurable control systems are still not truly interchangeable, reconfigurable, or open to end users or any third party developers. Currently, a CAD/CAM vendor must develop a postprocessor to generate a machine-specific M&G code for each machine tool controller. This represents a tremendous burden on any

CAD/CAM organization. Only by completely eliminating the machine-dependent M&G codes will a truly open and reconfigurable control system be feasible.

Second, these developed control architectures are still dependent upon some customized hardware. The control architectures in [8], [9], [10], [12], and [14] use DSP boards in motion control and many motion control algorithms are embedded inside these motion control boards. They cannot be interchanged or modified externally by end users or third party developers, which limit the interchangeabilities and the reconfigurability of these proposed control systems.

Third, these control architectures do not maintain associativity between the CAD model, CAM system, and the CNC machine. As a result, this is a great deterrent to fully integrated CAD/CAM and sensor-based control.

2.1.5 STEP-NC

With the limitations seen in those past machine control systems and the problems existed in the current standard (ISO 6983) of machining instruction code, namely M&G code, a modernized machining code standard (ISO 14649), called STEP-NC, is being developed. With the development and introduction of this new ISO standard 14649, STEP-NC extends the STEP geometric data exchange standard (ISO 10303), a neutral data exchange format, into the manufacturing domain by defining a two-way interface between CAM process planning systems and NC control systems. STEP-NC is a neutral data description language designed to be CAM independent and NC machine-tool independent; thus, the post-processing of process plans into M&G codes specific to each machine is no longer necessary.

Currently, under the IMS project [32-36] called STEP-NC in Europe and Asia, and Super Model in USA, industrialists and academics are collaborating to deliver a new data model as an ISO 14649 standard for CNC machines and to develop STEP-NC controllers. Parallel to these development efforts, researchers [37, 38] are developing a new generation of CAM systems that are designed to be completely STEP-NC compatible and independent of NC machine tools.

Even though STEP-NC provides a better link between CAM systems and CNC machine tools, it has not taken the integration process far enough. There is still no direct associativity between the parametric CAD model and the STEP-NC file. Because of this, many disadvantages can still be found that are commonly found in the M&G code (ISO 6983). For instance, if the original CAD model from which the STEP-NC file was created is modified, those changes were not reflected on the STEP-NC file that already left the system. The STEP-NC file, which is loaded into a STEP-NC compliant controller, is not parametric, meaning that any change in the geometry on the machine tool controller cannot be done. But even if it could be done, those changes would not be reflected back to the original CAD model from which the STEP-NC file was created.

2.2 Direct Machining And Control (DMAC)

The proposed dynamically reconfigurable machine tool controller in this dissertation is based on the Direct Machining And Control research at Brigham Young University. In the past six years, the DMAC research group has developed a direct machining architecture that allows CAD/CAM applications to run machining process directly on a DMAC controller.

Prior to this dissertation, research work [39-48] has connected the DMAC controller directly to ParaSolids, Unigraphics, Alias, GibbsCAM, CATIA and PC-DMIS, a popular part dimensional inspection application. Fig. 2.1 shows the DMAC flexible software structure to connect to these CAD/CAM systems. The idea is to take full advantage of the 3D modeling and tool path planning capabilities of CAD/CAM packages and to utilize a DMAC open-architecture controller to run the derived machining processes directly. This approach completely eliminates the machine-dependent M&G codes and establishes a direct link between CAD model, CAM system, and CNC machine. The design strategy of the DMAC architecture is the foundation from which integrated CAD/CAM and sensor-based control can be truly realized.

The DMAC architecture is configured on a dual-processor platform with CAD/CAM applications running on the first processor and all the real-time control applications running on the second processor.

The tool paths and process plans generated from CAD/CAM applications are passed down directly to the motion planner [39, 44] through a Direct Machine Interface [41]. The motion planner is composed of a trajectory generator and a kinematics object. The motion planner will generate all motion setpoints, position, speed, and acceleration, for each independent joint at each trajectory step. These joint setpoints are first mapped into the actuator setpoints and are then fed to the Servo Controller.

The Servo Controller [40, 45] receives actuator position, speed, and acceleration setpoints from the motion planner. Then based on certain control laws, such as Proportional-Integral-Derivative (PID) and feed forward control, the control effort, in the form of torque commands, is calculated and sent down to each motor through a hardware interface.

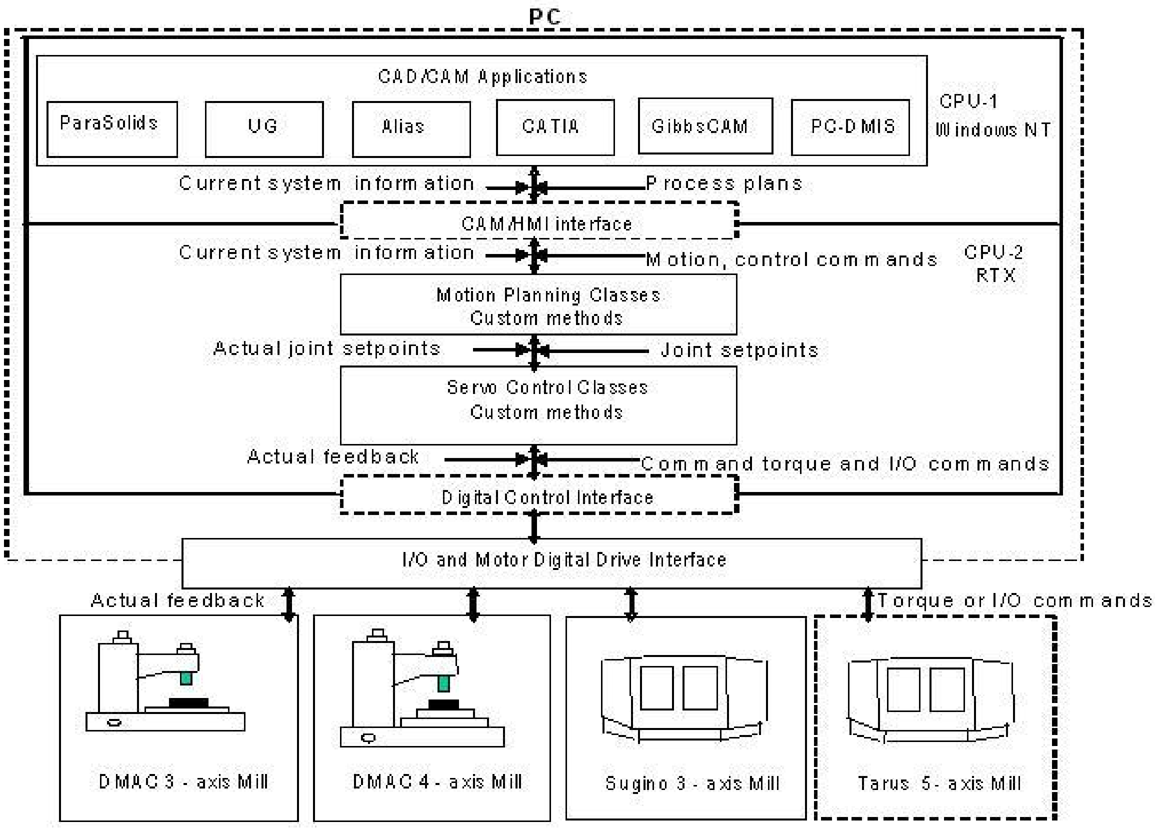


Fig. 2.1 Illustration of the flexible DMAC software structure

The DMAC architecture is fully software-based and can be configured to communicate directly with any CAD/CAM system, given the right interface functionality. Presently, the DMAC controller supports linear, circular, and Nurbs-based motion, which are the general motions required for a machine tool controller. This general architecture will be the basis from which a newer reconfigurable controller (RMAC) will be developed, and will be explained in the rest of this dissertation.

CHAPTER 3 RMAC SOFTWARE ARCHITECTURE

This chapter proposes and develops a new software architecture for a dynamically reconfigurable machine tool controller. It then presents the necessary software modules and interfaces defined within the RMAC architecture.

3.1 Traditional CNC Paradigm vs. RMAC Paradigm

In a traditional CNC paradigm, one machine tool controller is dedicated to a particular CNC machine tool. The functionality of that controller cannot be changed by end users for controlling different machines. For example, a CNC controller designed for a three-axis mill cannot be used to control a five-axis machining center.

Fig. 3.1 shows the standard steps used to plan a process and conduct it on a machine tool:

- Model a part using a CAD system.
- Create tool paths using a CAM system.

- Output a CL or APT file that contains tool path geometry data.
- Post-process the CL or APT file to obtain an M&G-code file, which then is delivered to the machine
- Operate the machine until the part (or batch of parts) is made.

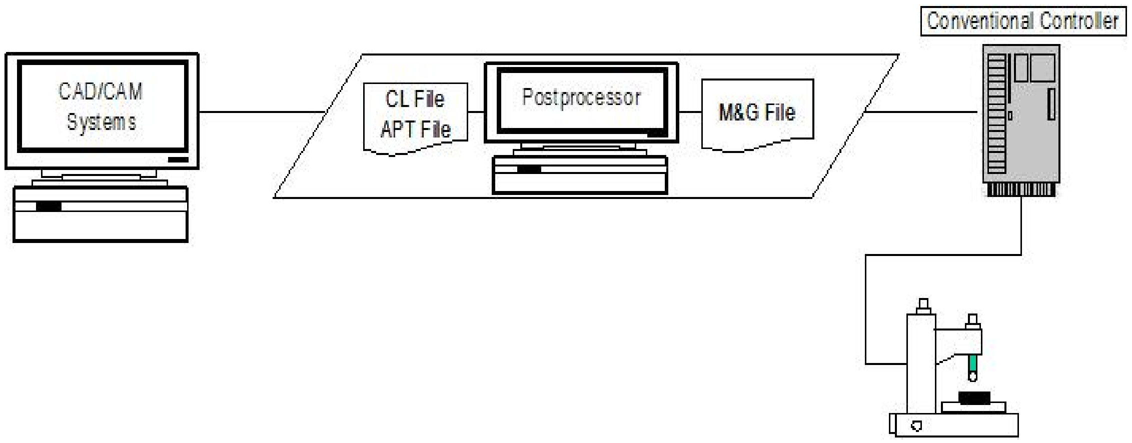


Fig. 3.1 Traditional CNC paradigm

CL and APT files are independent of machine tool controllers, but the M&G file is machine-specific. This conventional data flow from CAD to CAM systems and to a CNC machine tool creates the disassociativity between the original CAD model and the driving machining codes, namely M&G codes. The CAD description is not used directly on the machine; instead it must go through a machine-specific post-processor (of which there are estimated to be about 5,000 in existence). Due to many different dialects and vendor-specific additions to the language, M&G codes are not always interchangeable between different controllers and machines. This obsolete standard assumes that information flows from the CAD to the shop floor, and does not enable feedback of experience from the shop floor back to the designer.

As a result, there is a growing demand from machine tool end users to develop a new generation of machine tool controllers that are both highly flexible and dynamically reconfigurable based on newer manufacturing process requirements. For example, end users of a three-axis mill may require the addition of new sensor-assisted application-specific modules to efficiently and cost-effectively convert the mill into an inspection system. Also, a machine tool controller designed for a milling operation may be required to support a turning operation as well. Moreover, with the ever growing number of parametric CAD models widely used in product design, end users of CAD/CAM and machine tools expect the information flow between CAD/CAM and machine tools to be bi-directional, which would promote feedback from the shop floor back to the CAD designer. Therefore, these new requirements from manufacturing companies and machine tool end users pose new challenges for designing future machine tool controllers.

Fig. 3.2 presents the RMAC paradigm. Under this new paradigm, machine tools are controlled similar to the way printers are controlled by a personal computer. All machine tools are directly connected to CAD/CAM applications through different device drivers. This driver software acts as an interface between CAD/CAM systems and the control software. CAD/CAM users can select different machines to execute the process plans based on manufacturing process requirements. By calling a specific device driver, the tool paths and process plans generated from the CAD/CAM applications can be sent directly through the driver's interface. The software driver can then enable the same reconfigurable controller for controlling the machine that is connected through this driver. Under the RMAC paradigm, CAD/CAM software, device driver software, and control software all reside in the same PC, thus allowing the CAD description to be used directly on the machine. Doing so makes the information flow bi-directional; the CAD master model is sent to the controller through a device driver interface, and the modifications made on the shop floor can be fed back to update the original CAD model through the same interface.

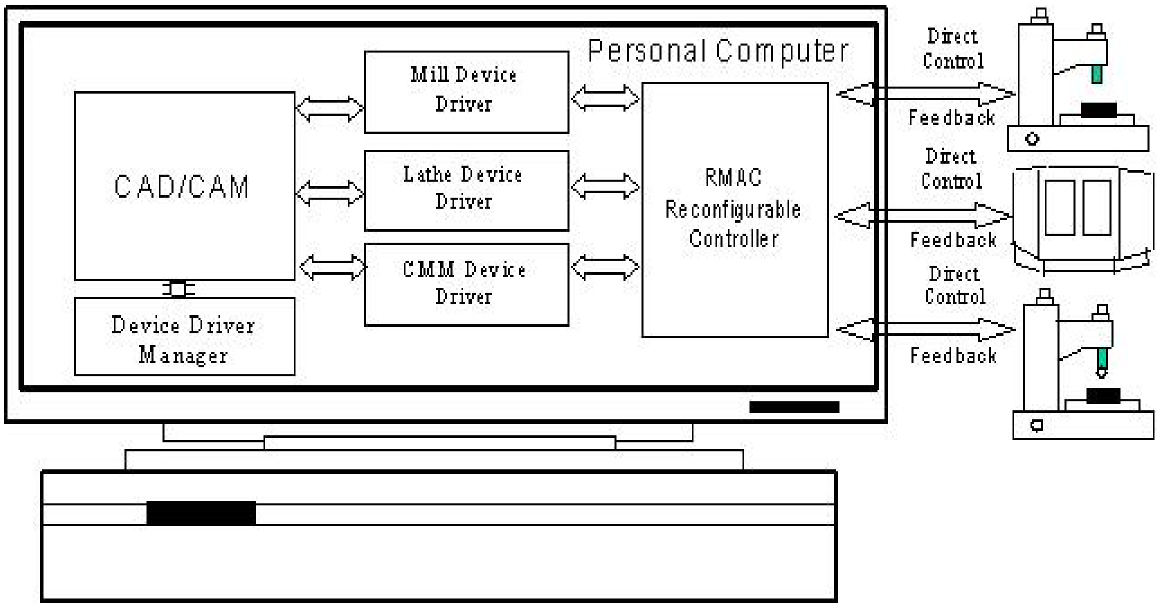


Fig. 3.2 RMAC paradigm

3.2 RMAC Control Schemes

The RMAC architecture developed in this research is generic, and therefore applicable to various control applications, such as machining, welding, robotics, etc. For these different applications, the control software must be flexible enough to accommodate different control schemes.

3.2.1 Position and velocity control

For most modern machine tools or robots, position and velocity control is the most widely used method.

To control a machine tool or robot's position and velocity in Cartesian space, a CAD/CAM application needs to generate a series of tool paths along which the mechanism's tool must follow. The process plan may also specify path following speeds and a spindle rpm. Fig. 3.3 shows how RMAC controls such a mechanism.

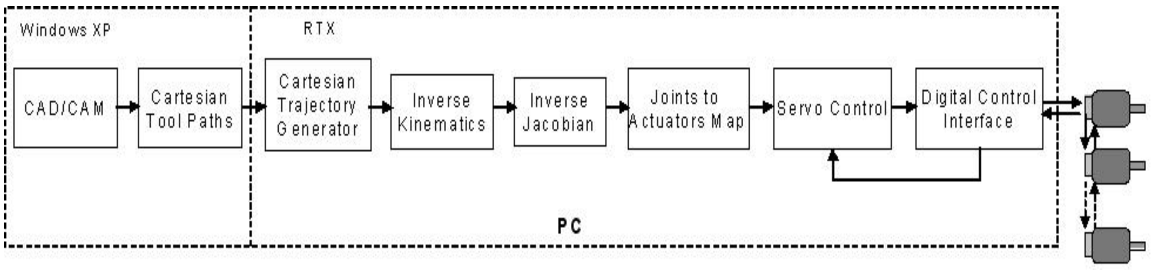


Fig. 3.3 RMAC controlling steps on position and velocity control

From the diagram, the first step consists of generating Cartesian tool paths inside a CAD/CAM package. Since these paths are associated with the master CAD model and are used directly to drive the RMAC compliant mechanism, whenever the CAD model is changed, the associated tool paths will be updated and automatically reflected in the machined part.

The generated Cartesian tool paths are then sent down to the RMAC controller to produce the mechanism tool's desired motion. A Cartesian trajectory generator is used to interpolate the tool paths to generate the tool position and orientation that the tool can follow.

To follow the desired Cartesian tool path, position, velocity, and acceleration setpoints must be found for each individual joint. This requires a mapping between a mechanism's Cartesian space and its joint space.

The transformation between Cartesian space and joint space requires an understanding of the mechanism kinematics. For instance, forward kinematics consists of calculating the position in Cartesian space, given a set of joint position. Inverse kinematics is the reverse of the forward kinematics: it involves calculating the joint positions necessary to position the tool at a given point in Cartesian space. The forward Jacobian consists of calculating the velocity of the tool in Cartesian space, given a set of joint speed, and, the inverse Jacobian consists of calculating the joint speeds necessary to generate the desired tool velocity in Cartesian space. Reference [50] contains three chapters that cover, in detail, the kinematics computations.

To find joint position, velocity, and acceleration given a desired Cartesian tool path, Inverse kinematics is used to map a mechanism's Cartesian state to its joint state. The inverse Jacobian is used to map a mechanism's Cartesian velocity to its joint velocity. Equation $\Theta' = J^{1}(\Theta)\upsilon$ relates the joint speed vector to the corresponding tool speed vector, where Θ' denotes the joint speed vector and υ, the tool speed vector. $J^{1}(\Theta)$ is the inverse Jacobian matrix and is mechanism-specific. The joint accelerations can usually be derived by differentiating the joint velocities at two consecutive trajectory steps. However, because inverse kinematics and the inverse Jacobian are machine-dependent, each different RMAC-complaint mechanism requires a specific inverse kinematics and inverse Jacobian algorithm.

To drive a physical motor, position, velocity, and acceleration setpoints must be found for each individual actuator. This requires a mapping between a mechanism's joint space and its actuator space.

Typically, a mechanism's axes are not directly actuated by motors. Instead, they are connected and actuated by intermediate mechanisms, such as ball screws, gears, or pistons. The manner in which actuators may be connected to move a kinematic joint varies among different mechanisms. For instance, some mechanisms use a ball screw to enable an angular motor to drive a linear kinematic joint. Sometimes, two actuators work together in a differential pair to move a single joint. At other times, a linear actuator rotates a revolute joint through the use of a four-bar linkage. In all, there are many other ways in which actuators can be connected to drive kinematic joints.

To find actuators' position, velocity, and acceleration given a setpoint of joint position, velocity, and acceleration, a machine actuator map object needs to be developed. The machine actuator map object contains a set of functions to determine the mappings between actuator space and joint space. These mappings are mechanism-specific and must be designed and implemented for each RMAC-complaint mechanism. Once the actuator's setpoints of position, velocity, and acceleration have been determined, they are passed down to the servo controller [30].

Servo control deals with establishing mathematical models to compute the control effort—in the form of a torque value—necessary to move control system variables to some desired value, or "reference" value. Depending on what control methods are utilized, these control system variables may be position, velocity, or contact force.

In the field of feedback control of dynamic systems [51, 52], control researchers have developed several different control laws based on different control criteria. The concept of these control laws is to create different mathematics models, which can represent the dynamic system. Such models allow for computing the servo control effort necessary to move the actuator system to follow the commanded position or velocity within the designed tolerances.

Consequently, the servo control algorithms may be machine-specific as different mechanisms require different servo algorithms based on the machine tolerance or customer requirements. Therefore, for each RMAC-complaint mechanism, a specific servo control algorithm needs to be developed and implemented for each kinematics joint.

Once the servo controller calculates the necessary torque value for each actuator, this torque value needs to be sent to each digital motor drive through a digital control interface.

For any RMAC-compliant mechanism, a digital interface is necessary to connect the digital control devices with the controller software. With the increasing digitization of control applications, and with the evolution of computer communication hardware, there are many possible communication standards, such as IEEE 1394, USB2, and proprietary fiber optic communication protocols, that can be chosen to enable communication between the digital motor drive and the controller software. Therefore, each RMAC-complaint mechanism may require a specific digital control interface.

Finally, to connect any external I/O sensor, such as limit switches or coolant on/off switches, to the RMAC controller, a digital I/O interface needs to be developed. For each RMAC compliant mechanism, a different I/O board may be chosen to handle the I/O connections. Therefore, a mechanism-specific digital I/O interface needs to be designed and implemented for each mechanism to be controlled.

The above outline shows that even though the kinematic structure of different RMAC-compliant mechanisms may vary, these mechanisms are still similar in how they are controlled. In the end, the ability to allow top-level CAD/CAM applications to switch controlling from one machine to another, or from one control application to another at run-time, has become a great challenge for control software designers.

3.2.2 Force or hybrid force/position control

While position and velocity control are widely used in machine tools and robots, there are other occasions when position control alone may not suffice. For instance, for robotics welding, assembling, and friction stir welding operations, the position of the tool is not specified as the control variable. Instead, the contact force or the combination of force and position are the system variables that need to be controlled.

Force control, or hybrid force/position control schemes, are quite different from position control. Fig. 3.4 shows a hybrid force/position control scheme applied on a three-axis kinematic structure. This kinematic structure has three prismatic joints moving individually along X, Y, and Z directions. The X and Y prismatic joints are free to move, while the Z axis is constrained so that the tool cannot move in the Z direction. The tool is currently normal to the XY plane and is in contact with a surface parallel to the XY plane.

The solution to this hybrid force/position control problem is to control joints X and Y with a position controller while simultaneously controlling the contact force along the Z axis with a force controller. Here, X_d and Y_d are the desired positions which feed into the position controller. $\dot{X}_d, \ddot{X}_d, \dot{Y}_d$, and $\ddot{Y}_d$ are the desired velocity and acceleration points for joints X and Y, and are generated from the motion planner described in section 3.2.1. These motion setpoints need to be fed into the position controller to compute a necessary torque value. *X* and *Y* are the actual positions, which are fed back from the digital motor drives.

The Z axis is out of the motion planning loop. F_d is the desired contact force that needs to be controlled. The actual force *(F)* is measured by a force sensor, which is attached to the Z axis. This value is fed back to the force controller for computing the necessary control effort for joint Z.

As illustrated, these control schemes are quite different in terms of the control characteristics and the control methods utilized. To take advantage of these control methods and to integrate them into RMAC, a flexible software architecture must be developed, allowing for easy reconfiguration of these different control methods.

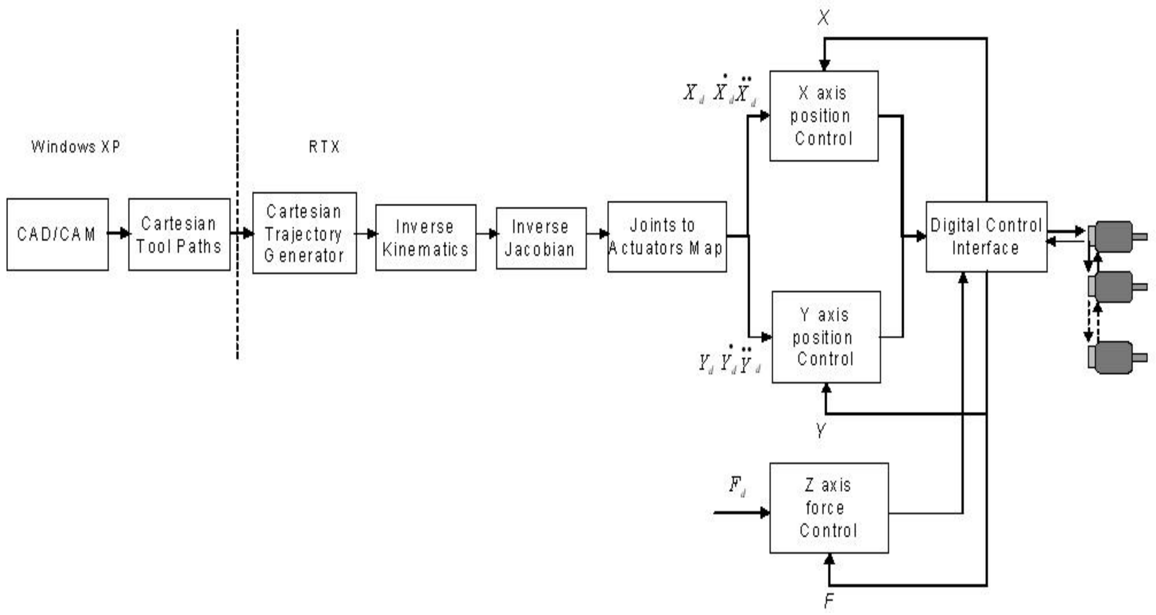


Fig. 3.4 Hybrid force/position control

3.3 RMAC Software Architecture

The overall software architecture for the RMAC control system is shown in Fig. 1.2. As can be seen from this figure, the RMAC control system is decomposed into separate hierarchically organized software modules, with CAD/CAM applications and the device driver manager sitting at the top and the RMAC reconfigurable controller at the bottom. Residing between the CAD/CAM systems and the RMAC reconfigurable controller are the device driver software and the COM interfaces. Motion control and configuration commands flow from CAD/CAM to the RMAC reconfigurable controller through the device driver and the COM interfaces. The machining feedback information flow from the RMAC reconfigurable controller back to CAD/CAM through the same device driver and COM interfaces. To allow for these control and feedback information flows, three interfaces and their interface APIs are developed.

3.3.1 RMAC software modules and interfaces

Fig. 3.5 shows the necessary software modules and the interfaces defined within the RMAC architecture. The software system is composed of the following six different programs:

- **CAD/CAM system** creates 3D representations of physical models and generates the manufacturing process plans.
- **Device driver manager** maintains a device driver database relevant to a collection of different mechanism devices and their driver DLLs (see Fig. 3.7). Meanwhile it provides interface APIs for CAD/CAM users to query for a proper machine and then locates a driver DLL for that machine.

- **Device driver** maintains a device database relevant to the details of a mechanism (see Fig. 3.8). By accessing this database, the device driver software knows exactly how to properly operate this mechanism. It then connects this physical mechanism directly to a CAD/CAM application and processes the CAD/CAM function calls to enable easy reconfiguration of the RMAC reconfigurable controller necessary for direct control.
- **RMAC_Config interface** directs the configuration commands from a device driver to the RMAC controller to allow the reconfiguration of the motion planner, servo controller, and the underlying digital control interface.
- **RMAC_CAM interface** directs the motion and control commands to the RMAC controller, receives the machining feedback information, and sends it to a device driver software.
- **RMAC** open-architecture reconfigurable controller (see Fig. 3.6) receives the configuration commands from the device driver. It uses a configuration system to map any mechanism-specific or application-specific control codes from the relevant DLL libraries. It then interpolates motion and control commands to generate the necessary torque values to drive each individual actuator.

Control and feedback information flows among these software modules through the following three interfaces:

- **Device driver manager interfaces to CAD/CAM:** The device driver manager exposes interface APIs to CAD/CAM to allow the CAD/CAM applications to access the device driver database for obtaining the necessary machine information.

- **Device driver interfaces to CAD/CAM:** The device driver exposes interface APIs to CAD/CAM allowing the CAD/CAM applications to access the device database for obtaining the detailed machine information. This machine information is then used to reconfigure the RMAC controller necessary for executing manufacturing process plans on the selected machine tool.
- **Device driver interfaces to the RMAC reconfigurable controller:** The device driver software communicates with the RMAC reconfigurable controller through two COM interfaces and they are RMAC_Config and RMAC_CAM. The device driver software contains an instance of the RMAC_Config and RMAC_CAM, thus, all the interface APIs defined within these two COM interfaces are directly accessible to the device driver software.

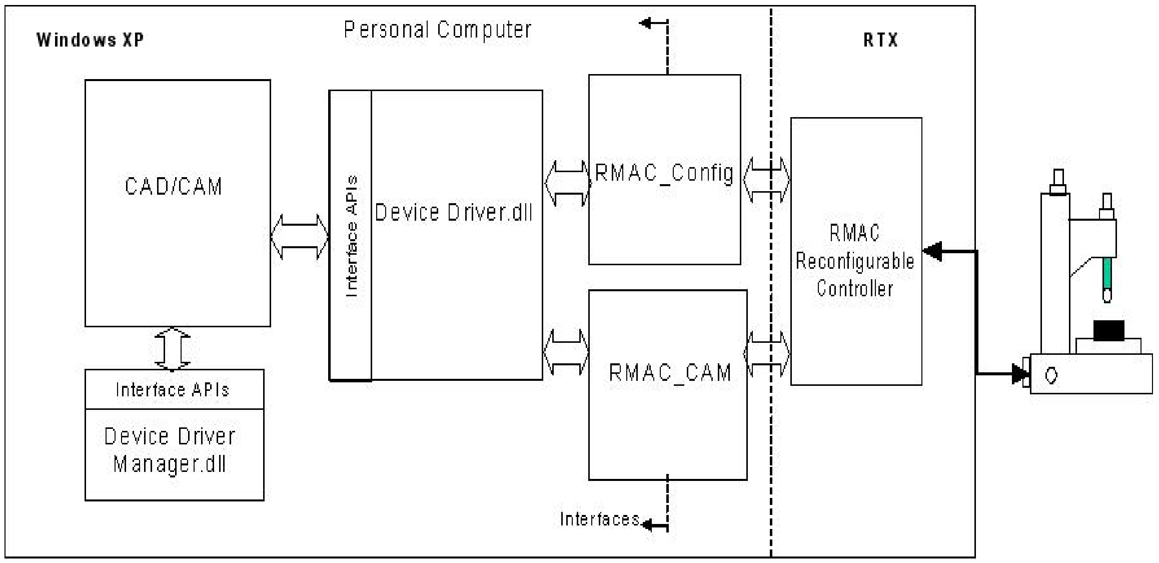


Fig. 3.5 Software modules and interfaces in RMAC architecture

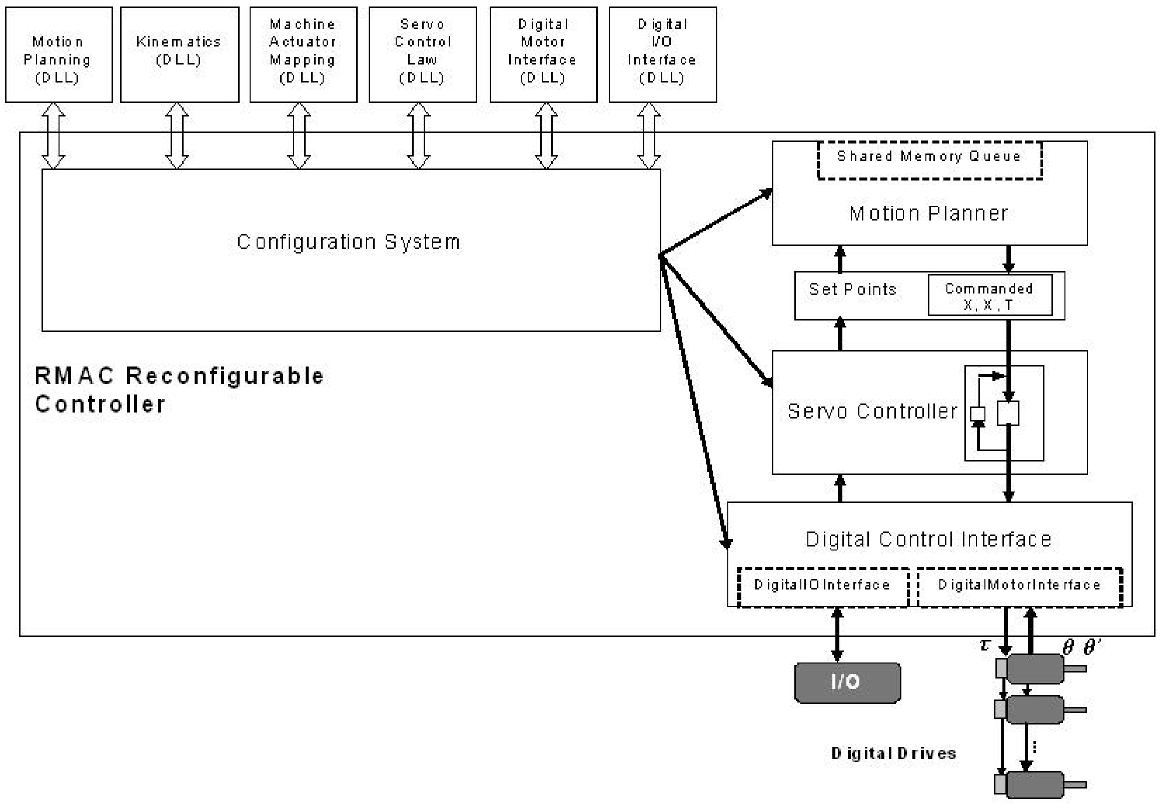


Fig. 3.6 RMAC reconfigurable controller architecture

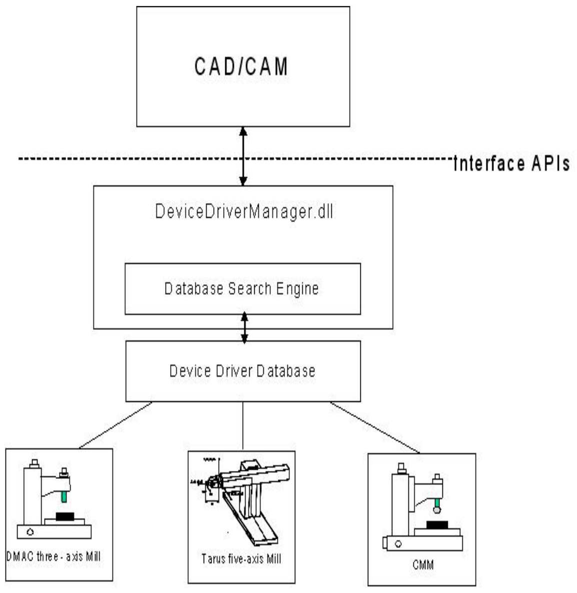


Fig. 3.7 Device driver manager

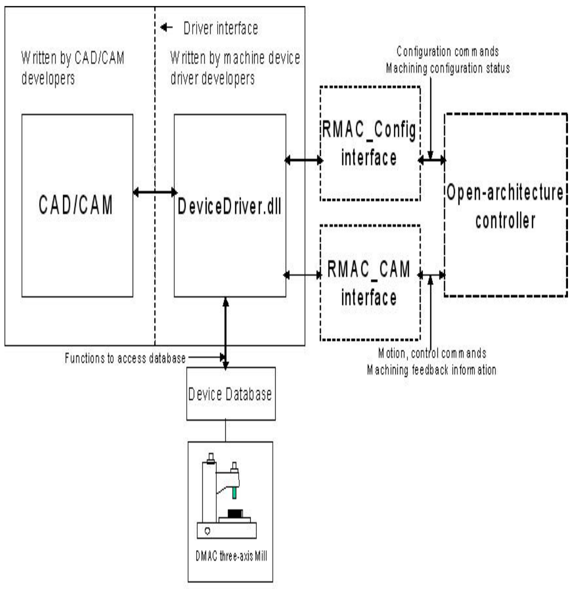

Driver interface
Written by CAD/CAM developers
Written by machine device driver developers
CAD/CAM
DeviceDriver.dll
Configuration commands
Machining configuration status
RMAC_Config interface
Open-architecture controller
RMAC_CAM interface
Motion, control commands
Machining feedback information
Functions to access database
Device Database
DMAC three-axis Mill

Fig. 3.8 Device driver

3.3.2 Control information flow in RMAC

To better understand how control and feedback information flows among these different software systems, or, more specifically, how motion command flows from CAD/CAM applications to the RMAC reconfigurable controller and the feedback information flows from the RMAC controller to CAD/CAM, an example is given as shown in Fig. 3.9.

From Fig. 3.9, it assumes that CAD/CAM users have selected a machine and a device driver DLL has been loaded into memory. Here, a CAD/CAM application generates a Nurbs tool path and seeks to send this tool path to a RMAC-compliant mechanism for direct machining. It makes a driver service function call named *Machine_MoveInNurbs*. Upon receiving this interface function call, the device driver interprets this CAD/CAM function and makes a COM interface call named *MoveAlongNurbs*. The RMAC_CAM interface is used to direct this service routine to the RMAC controller. *MoveAlongNurbs* is the final function expected by the RMAC controller. Once the RMAC controller receives this service call, the motion planner interpolates the Nurbs tool path and generates the necessary motion setpoints. These motion setpoints are then sent to the servo controller to calculate the torque values. The torque values are sent to each individual motor drive at each trajectory step, and consequently, the tool is commanded to move along the Nurbs tool path.

During this operation, the digital motors actual position and speed are fed back to the servo controller through the digital control interface. These joints setpoints are then sent to the motion planner. The motion planner calculates the mechanism speed using forward kinematics algorithm. Once the device driver software makes a COM interface call *GetFeedrate*, the motion planner will send the actual machine federate value to the device driver. The device driver will send this value to CAD/CAM upon receiving the driver service call *Machine_GetFeedrate*. At this point, the mechanism actual federate value is fed back to CAD/CAM application for either display or debugging purposes.

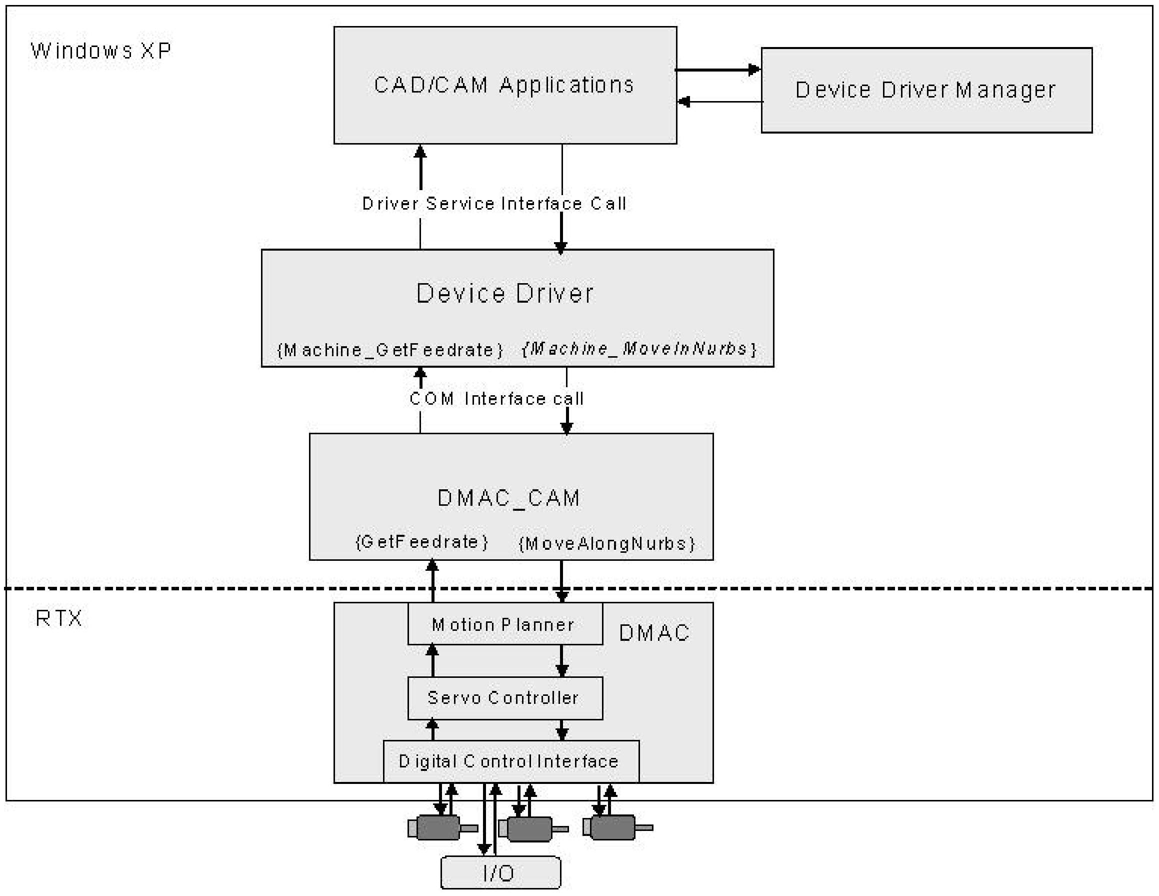


Fig. 3.9 Flow of information between CAD/CAM and the RMAC reconfigurable controller

The next chapter will describe each part of the software system involved in this information flow in details. The interface APIs enabling this information flow will also be discussed.

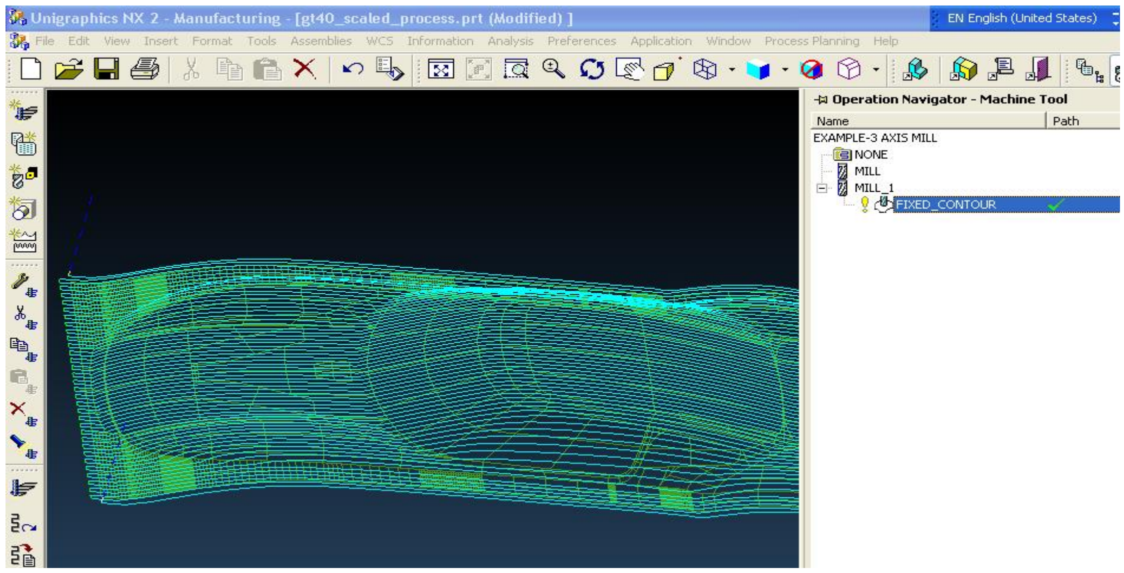
CHAPTER 4 METHODOLOGY

This chapter describes in greater details for each software module and interface defined within the RMAC architecture. It then presents the methodology for reconfiguring the RMAC reconfigurable controller necessary for controlling different machines.

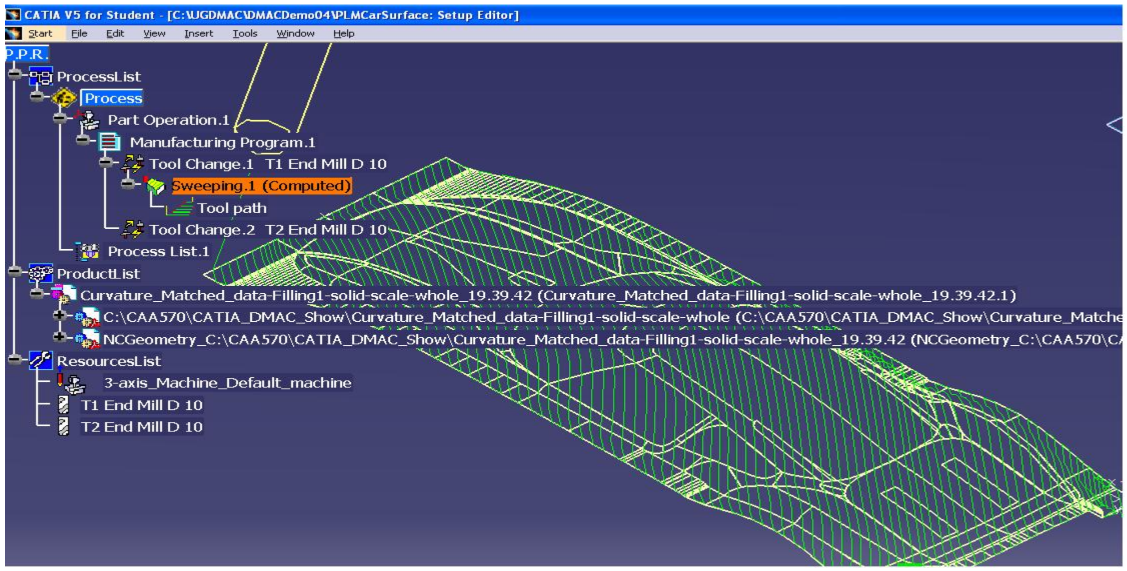
4.1 CAD/CAM

CAD/CAM systems are computer-aided engineering tools that are widely used to assist product design and manufacturing. Fig. 4.1 shows a Ford GT top surface being modeled and process planned in Unigraphics (UG) and CATIA. The manufacturing process plans generated from UG and CATIA are highlighted (see Fig. 4.1).

Traditionally, these manufacturing process plans must be post-processed into the ASCII APT and M&G files to be executed on a machine. To overcome this post-processing limitation, a device driver is developed for each individual machine to be


Unigraphics NX 2 - Manufacturing - [gt40_scaled_process.prt (Modified)]
EN English (United States)
File Edit View Insert Format Tools Assemblies WCS Information Analysis Preferences Application Window Process Planning Help
Operation Navigator - Machine Tool
Name
Path
EXAMPLE-3 AXIS MILL
NONE
MILL
MILL_1
FIXED_CONTOUR

a) UG


CATIA V5 for Student - [C:\UGDMAC\DMACDemo04\PLMCarSurface: Setup Editor]
Start File Edit View Insert Tools Window Help
P.P.R.
ProcessList
Process
Part Operation.1
Manufacturing Program.1
Tool Change.1 T1 End Mill D 10
Sweeping.1 (Computed)
Tool path
Tool Change.2 T2 End Mill D 10
Process List.1
ProductList
Curvature_Matched_data-Filling1-solid-scale-whole_19.39.42 (Curvature_Matched_data-Filling1-solid-scale-whole_19.39.42.1)
C:\CAA570\CATIA_DMAC_Show\Curvature_Matched_data-Filling1-solid-scale-whole (C:\CAA570\CATIA_DMAC_Show\Curvature_Matche
NCGeometry_C:\CAA570\CATIA_DMAC_Show\Curvature_Matched_data-Filling1-solid-scale-whole_19.39.42 (NCGeometry_C:\CAA570\C
ResourcesList
3-axis_Machine_Default_machine
T1 End Mill D 10
T2 End Mill D 10

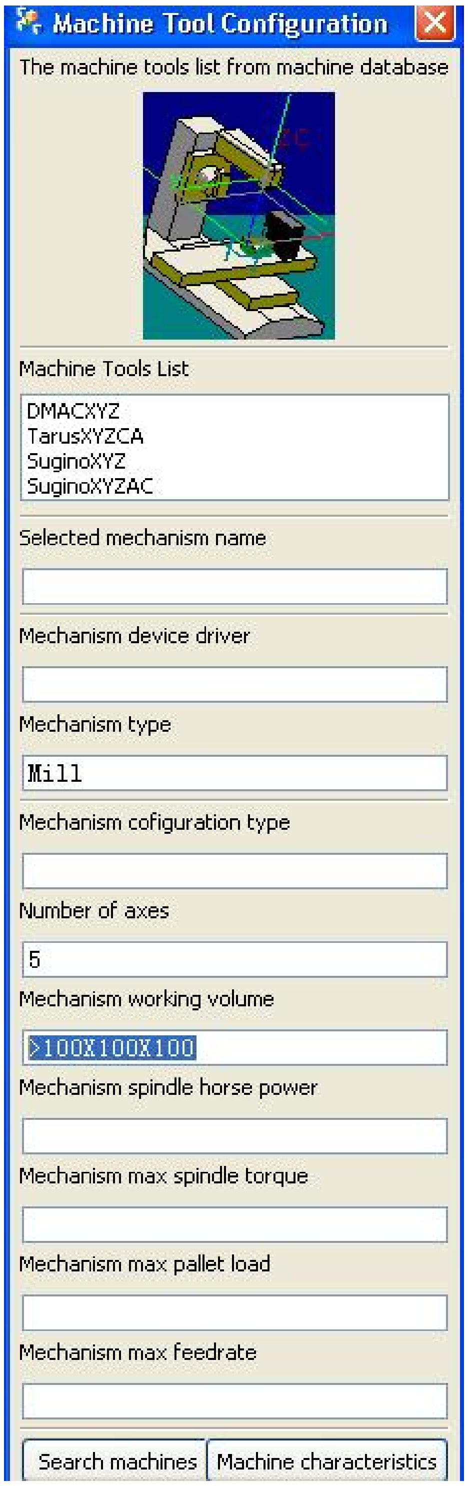
b) CATIA

Fig. 4.1 UG and CATIA process plans

connected directly with CAD/CAM systems. Whenever CAD/CAM users generate the manufacturing process plans and are ready to execute them on a machine, they will first select a machine to perform the process. CAD/CAM software will automatically load a relevant device driver and then pass the process plans directly to that machine through the device driver. This parallels the way printers work in Windows. For instance, whenever a Microsoft Word user wants to print a document, the document is sent through a printer driver directly to the printer. There is no need to store and maintain intermediate process files. Instead, a unique printer driver establishes a direct link between the computer and the printer. The printer driver knows exactly how to operate the printer as desired by end users.

Because many different machines exist that are feasible for executing a manufacturing process plan, a customized graphic user interface is embedded inside CAD/CAM systems to assist users in selecting the best machine tool. This is shown in Fig. 4.2.

In Fig. 4.2, the configuration dialog boxes are designed as a plug-in user interface to UG and CATIA. These dialog boxes allow CAD/CAM users to see the different machines, and provide users with enough information to select the proper machine to perform the process. To better assist CAD/CAM users, the device driver manager also has a built-in search engine, enabling them to narrow down their selection to a few machines, based on various machine filter schemes. Fig. 4.3 shows two selected machine tools classified as five-axis mills, with a working volume greater than 100x100x100 mm.



Machine Tool Configuration

The machine tools list from machine database

Machine Tools List

DMACXYZ
TarusXYZCA
SuginoXYZ
SuginoXYZAC

Selected mechanism name

Mechanism device driver

Mechanism type

Mill

Mechanism cofiguration type

Number of axes

5

Mechanism working volume

>100X100X100

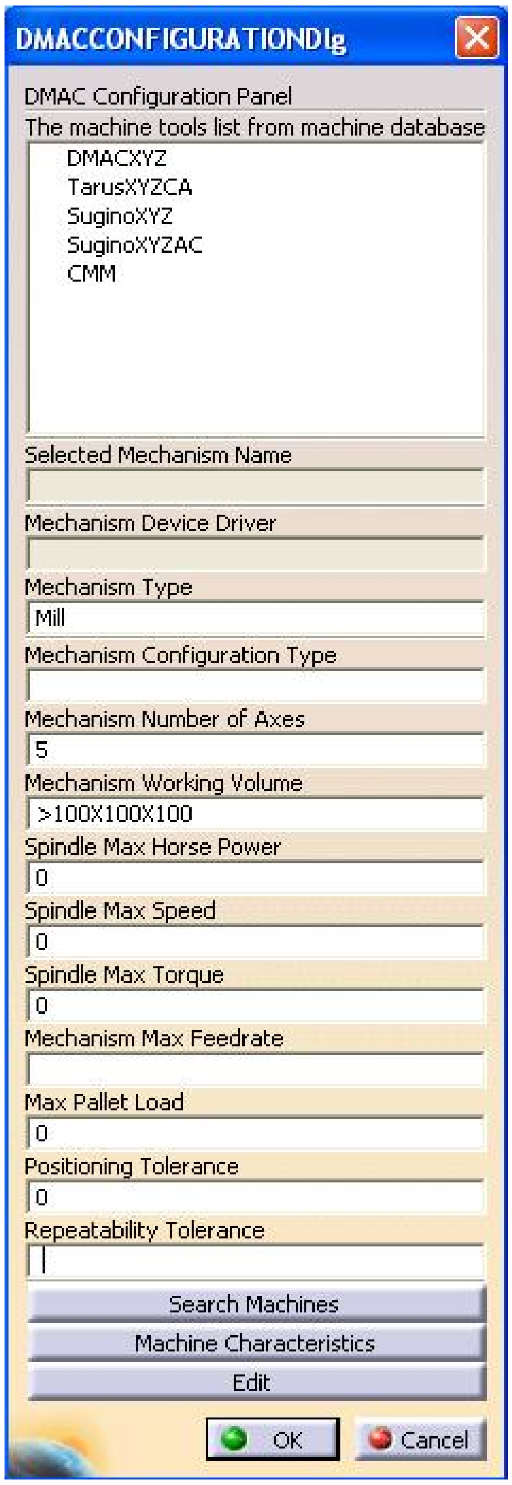
Mechanism spindle horse power

Mechanism max spindle torque

Mechanism max pallet load

Mechanism max feedrate

a) UG



DMACCONFIGURATIONDlg

DMAC Configuration Panel

The machine tools list from machine database

DMACXYZ
TarusXYZCA
SuginoXYZ
SuginoXYZAC
CMM

Selected Mechanism Name

Mechanism Device Driver

Mechanism Type

Mill

Mechanism Configuration Type

Mechanism Number of Axes

5

Mechanism Working Volume

>100X100X100

Spindle Max Horse Power

0

Spindle Max Speed

0

Spindle Max Torque

0

Mechanism Max Feedrate

Max Pallet Load

0

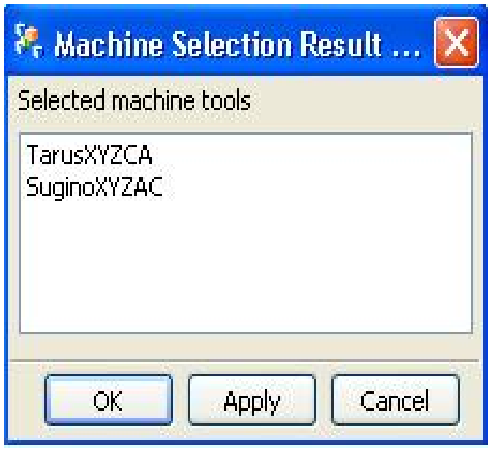
Positioning Tolerance

0

Repeatability Tolerance

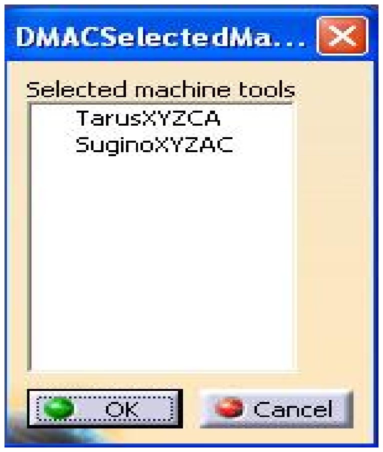
b) CATIA

Fig. 4.2 Machine configuration user interface under UG and CATIA



TarusXYZCA
SuginoXYZAC

a) UG



TarusXYZCA
SuginoXYZAC

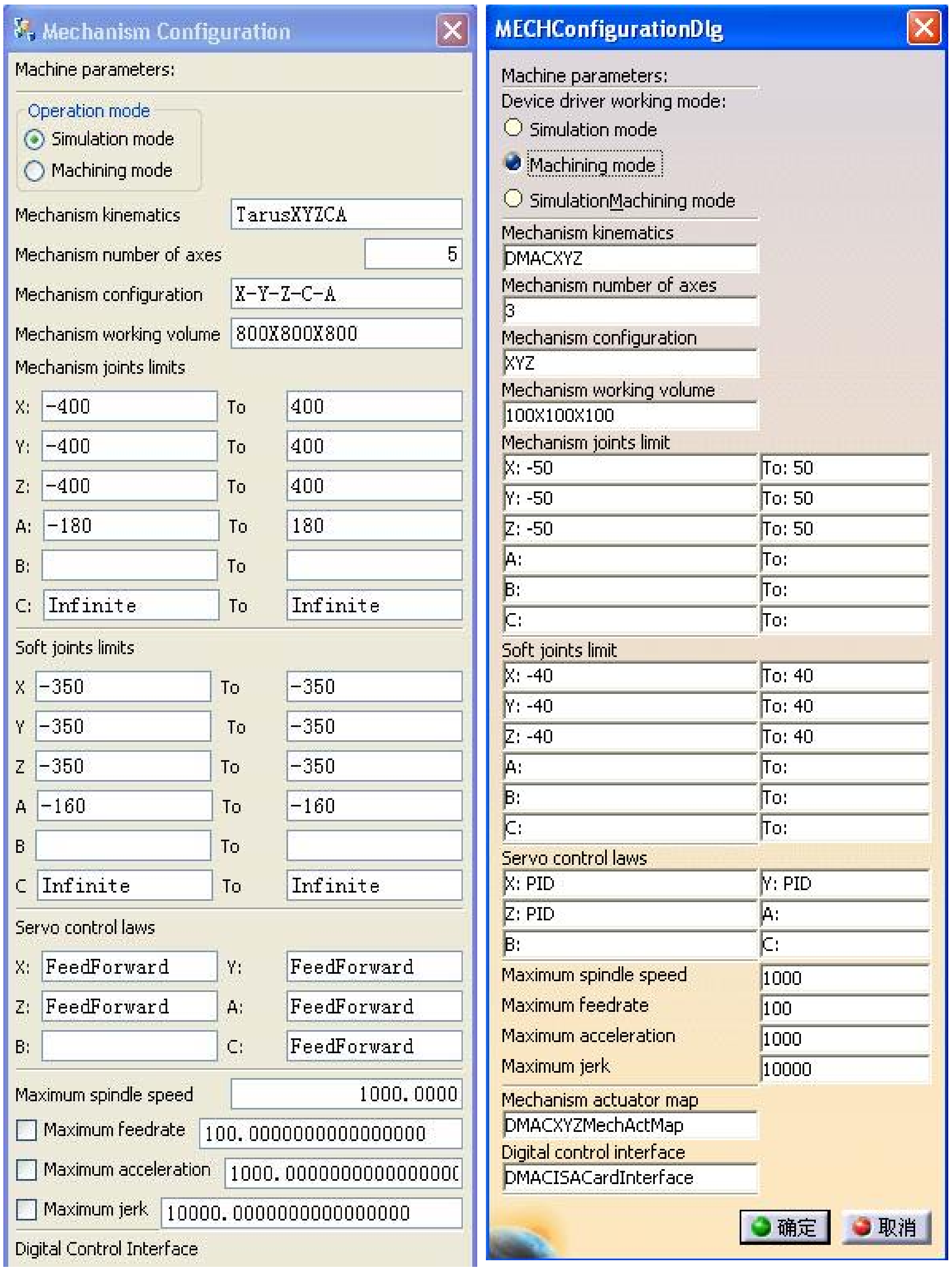
b) CATIA

Fig. 4.3 Selected machine tools under UG and CATIA

If CAD/CAM users want more information about a particular machine's characteristics to make a better decision, they can open a new dialog box by clicking the machine characteristics button (as shown in Fig. 4.2 and Fig. 4.4).

This dialog box contains detailed information about the machine, such as machine configuration, working volume, machine limits, maximum feederate, maximum spindle speed, etc. Such information assists CAD/CAM users in making a more informed decision about whether to use this machine to perform the process.

CAD/CAM applications use a device driver manager to obtain all machine characteristics information. The next section will describe in detail how the device driver manager obtains this information and what functions are exposed to CAD/CAM software for obtaining the necessary machine information to assist in the machine selection process.



Mechanism Configuration

Machine parameters:

Operation mode

[✓ Simulation mode] Machining mode

Mechanism kinematics	TarusXYZCA		
Mechanism number of axes	5		
Mechanism configuration	X-Y-Z-C-A		
Mechanism working volume	800X800X800		

Mechanism joints limits

X:	-400	To	400
Y:	-400	To	400
Z:	-400	To	400
A:	-180	To	180
B:		To	
C:	Infinite	To	Infinite

Soft joints limits

X	-350	To	-350
Y	-350	To	-350
Z	-350	To	-350
A	-160	To	-160
B		To	
C	Infinite	To	Infinite

Servo control laws

X:	FeedForward	Y:	FeedForward
Z:	FeedForward	A:	FeedForward
B:		C:	FeedForward

Maximum spindle speed	1000.0000
☐ Maximum feedrate	100.0000000000000000
☐ Maximum acceleration	1000.000000000000000
☐ Maximum jerk	10000.0000000000000000

Digital Control Interface

MECHConfigurationDlg

Machine parameters:

Device driver working mode:

Simulation mode [✓ Machining mode] SimulationMachining mode

Mechanism kinematics

DMACXYZ

Mechanism number of axes

3

Mechanism configuration

XYZ

Mechanism working volume

100X100X100

Mechanism joints limit

X: -50	To: 50
Y: -50	To: 50
Z: -50	To: 50
A:	To:
B:	To:
C:	To:

Soft joints limit

X: -40	To: 40
Y: -40	To: 40
Z: -40	To: 40
A:	To:
B:	To:
C:	To:

Servo control laws

X: PID	Y: PID
Z: PID	A:
B:	C:

Maximum spindle speed	1000
Maximum feedrate	100
Maximum acceleration	1000
Maximum jerk	10000

Mechanism actuator map

DMACXYZMechActMap

Digital control interface

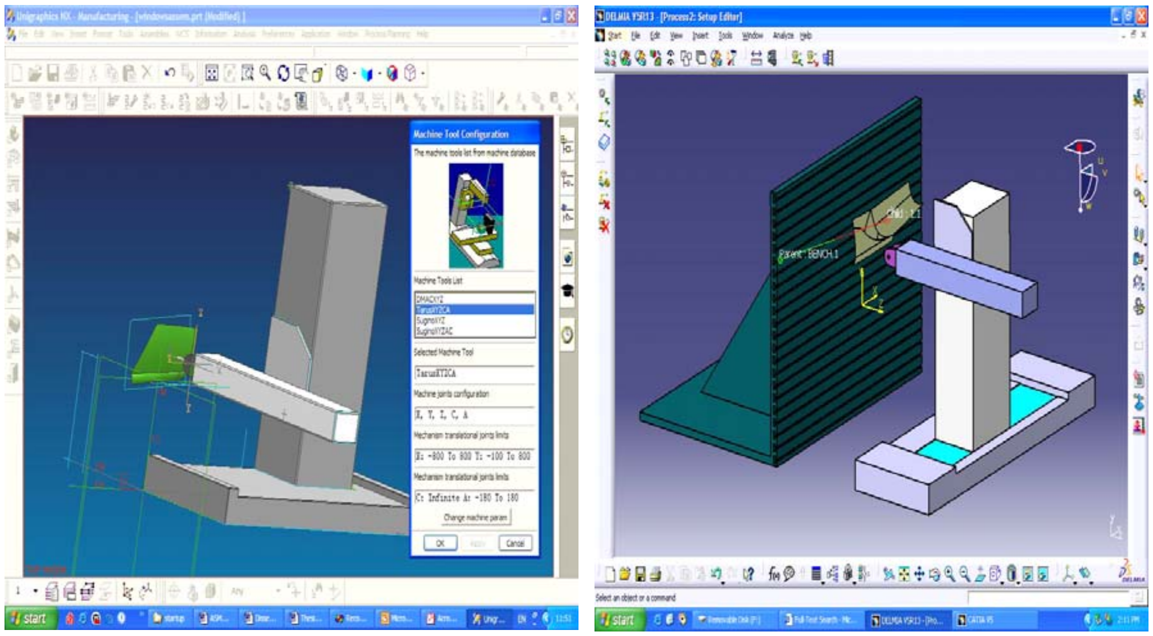
DMACISACardInterface

a) UG

b) CATIA

Fig. 4.4 Machine characteristics dialog box under UG and CATIA

Once CAD/CAM users select a proper machine, they can choose to perform the manufacturing process either by moving a connected machine to follow the desired tool paths and cutting a part, or by simulating the same operations on a virtual machine built inside the CAD/CAM applications. CAD/CAM users can set this virtual or machining option in the machine characteristic dialog box (see Fig. 4.4). Fig. 4.5 shows a process plan being simulated in UG and DELMIA, a simulation tool for CATIA users.



a) UG **b) DELMIA**

Fig. 4.5 Simulations under UG and DELMIA

4.2 Device Driver Manager

The device driver manager (see Fig. 3.7) is a DLL running independently from CAD/CAM systems. The functions of the device driver manager are as follows:

- Maintain a device driver database relevant to a collection of different machines and their driver DLLs.

- Uses a device tree structure to organize different device driver DLLs.
- Provide built-in database search engine to assist CAD/CAM users to narrow down their selected machines based on various searching schemes.
- Provide interface APIs to communicate with CAD/CAM applications.

4.2.1 Device tree

The device driver manager organizes a collection of device driver DLLs in a device tree structure (see Fig. 4.6). The base of this tree is called a root, and is represented by a root device driver folder. Under this root device driver folder, different mechanism device drivers are collected into sub folders according to their machine types (i.e. milling machine, robot, and CMM). At the second level of the device tree are nodes (branches) or end nodes (leafs). For example, as a branch, the milling machine node divides into three new nodes: the three-axis, four-axis, and five-axis node. Thus, all the three-axis mill device drivers are placed under the three-axis folder. Likewise, the four-axis and five-axis mill device drivers are placed under the four-axis and five-axis folders. As a result, the device tree's hierarchy reflects the structure and classification of the device drivers.

The purpose for a hierarchical structure is threefold: first, it enables easier management of different device drivers. Second, it provides an extensible foundation for adding new device drivers to the existing device driver database. Third, the tree structure allows the device driver manager to easily locate a device driver. By knowing the mechanism device type and the device driver name, the device driver manager can easily locate that device driver by traversing the tree from the root until it reaches a leaf.

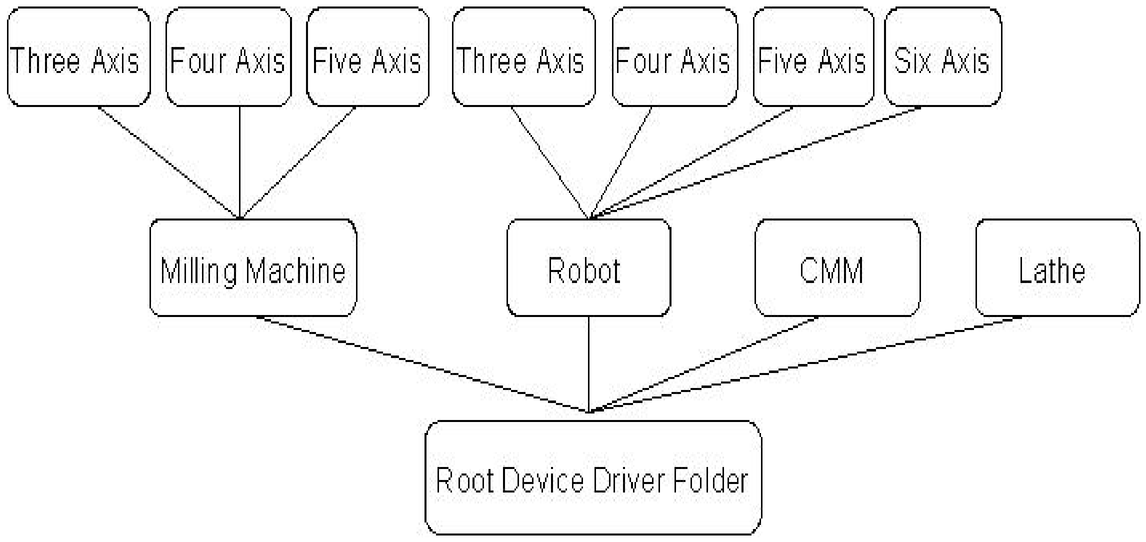


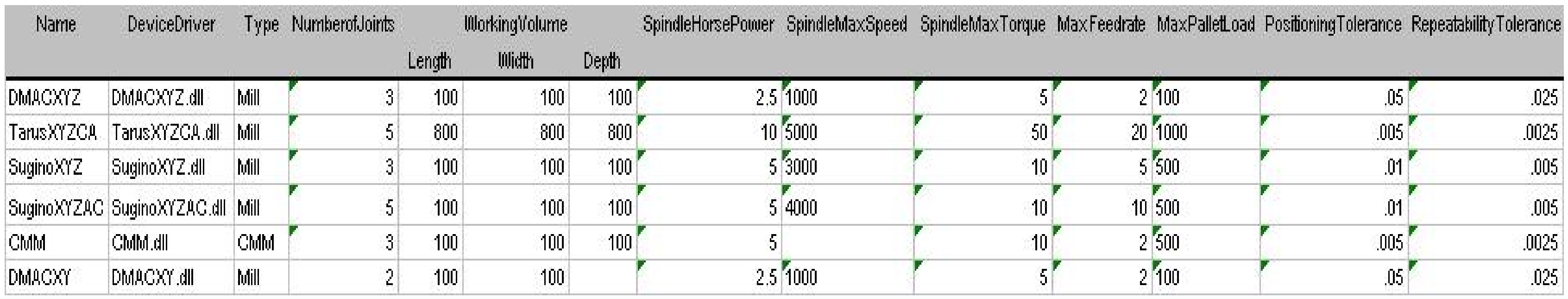
Fig. 4.6 Device tree structure

4.2.2 Device driver database

The device driver manager maintains a database called the *device driver database*. Table 4-1 displays the contents of this database. This database contains the minimum amount of information relevant to a mechanism and its device driver. Its purpose is to allow CAD/CAM users to inquire about a mechanism and its device driver to assist their evaluation and selection of a machine tool.

The design form for the database is Microsoft Access, a low-end relational database program widely used on small and medium sized databases. The standard user and application program interface to a relational database is the *structured query language* (SQL). SQL statements are used both for interactive queries for information from a relational database, and for gathering data for reports.

Table 4-1 Device driver database



Name	DeviceDriver	Type	NumberofJoints	WorkingVolume			SpindleHorsePower	SpindleMaxSpeed	SpindleMaxTorque	MaxFeedrate	MaxPalletLoad	PositioningTolerance	RepeatabilityTolerance
				Length	Width	Depth							
DMACXYZ	DMACXYZ.dll	Mill	3	100	100	100	2.5	1000	5	2	100	.05	.025
TarusXYZCA	TarusXYZCA.dll	Mill	5	800	800	800	10	5000	50	20	1000	.005	.0025
SuginoXYZ	SuginoXYZ.dll	Mill	3	100	100	100	5	3000	10	5	500	.01	.005
SuginoXYZAC	SuginoXYZAC.dll	Mill	5	100	100	100	5	4000	10	10	500	.01	.005
CMM	CMM.dll	CMM	3	100	100	100	5		10	2	500	.005	.0025
DMACXY	DMACXY.dll	Mill	2	100	100		2.5	1000	5	2	100	.05	.025

Since a manufacturing organization may have hundreds of different machines for CAD/CAM users to choose from, the device driver manager implements a built-in database search engine (in SQL syntax) to assist CAD/CAM users in their search for a particular machine. By using this search engine, CAD/CAM users can easily narrow down their selection to a few machines. For instance, if CAD/CAM users wish to find a five-axis mill classified machine, with a working volume greater than 150x150x150 mm, and a mechanism spindle greater than 5 hp, one SQL query in the database will narrow their selection to a single machine: TarusXYZCA, (see Table 4-1). This searching and selection process reduces the need for CAD/CAM users to review every available machine before finding one capable of performing the manufacturing process.

Once CAD/CAM users select a machine, the device driver manager is responsible for locating a relevant mechanism device driver. The combination of the second, third, and fourth columns are used to assist the device driver manager to track and locate a device driver. The mechanism type and mechanism number of joints columns are used by the device driver manager to traverse the device tree, while the mechanism device driver is the driver name that the device driver manager searches for. Once the device driver manager finds all of this information, it sends the information back to CAD/CAM systems.

Once the device driver manager obtains the minimum amount of information relevant to a specific mechanism, it can pass this machine and device driver information back to the CAD/CAM user interface (as discussed in section 4.1) for machine evaluation and selection purposes.

4.2.3 Device driver object

To assist the process of retrieving mechanism and device driver related information from the device driver database, a *device driver object* data structure is defined, as shown in Fig. 4.7.

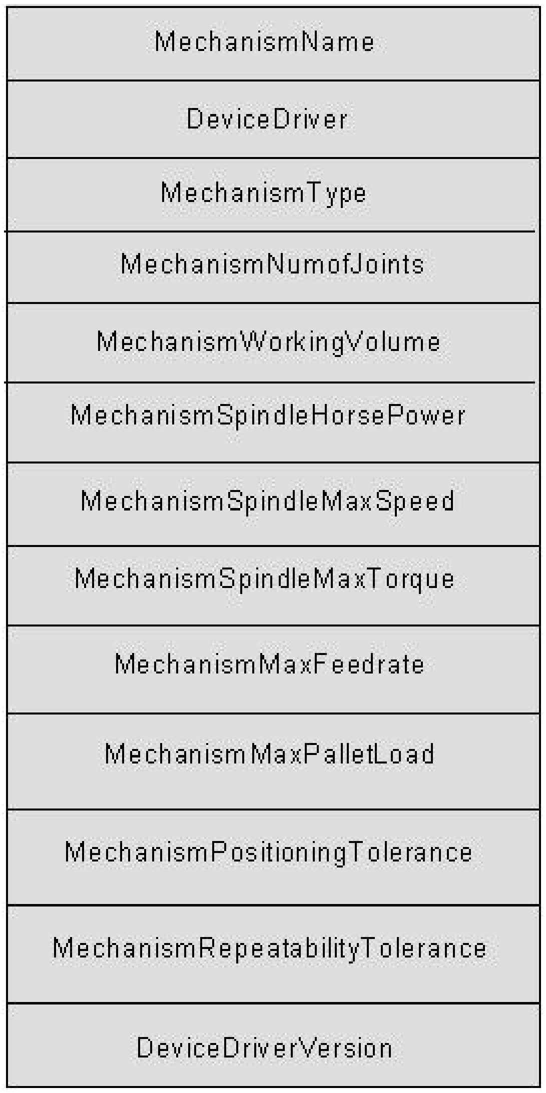


Fig. 4.7 The Device_driver_object data structure

This complete *device driver object* data structure is defined in the device driver manager's header file, as shown in Appendix I. The following excerpt is an example:

```
typedef struct _DEVICE_DRIVER_OBJECT{

char            MechanismName[100];
char            DeviceDriver[100];
…
char            DeviceDriverVersion[100];
}DEVICE_DRIVER_OBJECT, *PDEVICE_DRIVER_OBJECT;
```

The device driver manager uses the *device driver object* data structure to represent each device driver. According to this data structure, each field corresponds to one column in the device driver database.

The device driver manager will declare an ODBC (Open DataBase Connectivity) object. It then uses this ODBC object to access the device driver database and set up each field defined within the *device driver object*. ODBC is designed to be database-independent. The MFC (Microsoft Foundation Class) library contains well-defined function calls that allow the ODBC object to access any data source, local or remote. The ODBC object can use SQL statements to query the device driver database and assist CAD/CAM users in searching for correct machines.

4.2.4 Interface to CAD/CAM

The device driver manager is designed as a DLL and is a stand-alone program; thus, it must expose some interface APIs to enable communication with CAD/CAM systems.

The interface APIs are separated into two groups: functions that allow CAD/CAM systems to access the device driver database, and functions that return the selected machine information back to CAD/CAM.

CAD/CAM applications call the first group of interface APIs to operate on a device driver database. Three interface API examples—with no input parameters—are listed for demonstration.

- OpenDeviceDriverDatabase()
- CloseDeviceDriverDatabase()
- GetTotalNumberOfRecords()

Function *OpenDeviceDriverDatabase()* allows the CAD/CAM applications to connect to the device driver database. Function *CloseDeviceDriverDatabase()* disconnect the database from CAD/CAM. The last function *GetTotalNumberOfRecords()* returns the total number of machine records contained within the device driver database.

CAD/CAM applications call the second group of interface APIs to obtain information related to a selected machine. This machine information will then be displayed to CAD/CAM users, as described in section 4.1, for proper machine evaluation and selection. Two API examples are given below.

- Machine_GetDeviceDriver()
- Machine_GetMechanismMaxFeedrate()

The first function, *Machine_GetDeviceDriver(),* returns the selected device driver name to CAD/CAM applications. The second function, *Machine_GetMechanismMaxFeedrate(),* returns the mechanism maximum feedrate value to CAD/CAM.

Appendix II provides a more detailed list and description of these interface APIs.

4.3 Device Driver

A device driver (see Fig. 3.8) must be developed to connect a mechanism device directly to CAD/CAM. It may be useful to think of a complete mechanism device driver as a container for a collection of methods and classes. These methods and classes can be called by CAD/CAM systems to perform various operations on the connected mechanism device and to read back the mechanism operational parameters, such as current feedrate, spindle speed, joint value, current torque, etc. Each device driver must be able to entirely determine a particular mechanism's behavior and understand exactly how to make the mechanism work for the user. Specifically, the device driver should be designed with the following functions:

- Apply a self-contained device database to expose the details of a mechanism device.
- Expose functions required by CAD/CAM.
- Communicate directly with the RMAC reconfigurable controller.

The device driver is designed as a dynamic-link library (DLL). DLL is currently the de-facto library form for Windows device drivers [54, 55]. Because it is the only library that can be explicitly loaded by Windows at run-time, all of the device drivers running under Windows are designed as DLLs. Windows end users frequently need to install new devices, or to upgrade their old devices with a new functionality. And, because of the independent and run-time loadable nature of DLL, end users can easily upgrade their exiting devices with new functionality. Here, end users do not need to know all of the details about a device; rather, they simply download a device driver DLL from the device manufacturer and use the Windows device manager to install it. The Windows system will then load the device driver into memory and operate this device as the end user desires.

Similar to the Windows device driver concept, this research also uses DLL as the device driver form. Besides being run-time loadable by an application program, there are a few other advantages of using DLL to map mechanism-specific modules into a device driver. The following advantages comprise the rationale for choosing DLL as the device driver form.

First, DLLs are compiled and linked independently from the applications that use them. They are separate executable files containing functions or classes that can be called by application programs and other DLLs to perform certain functions or computations. Therefore, DLLs can be updated without requiring applications to be recompiled or relinked.

Second, DLLs are run-time modular while C++ classes are only build-time modular. This means that the loading of a DLL can be determined at run-time while the loading of a C++ class must be determined at link time. A C++ class can be designed and formed into a static library, but to use this library, an application program must first link this library into its executable file in order to run. Once the library is linked to an application, it becomes a permanent part of the application's executable file. All of the subsequent calls to the library functions or classes are resolved at link time, thus making the functionality of the application software no longer changeable at run-time.

Third, if several applications work together as a system, and they all share common DLLs, the *entire* system can be updated or improved by replacing the common

DLLs with enhanced versions. A bug fix for one of the DLLs fixes the bug in all applications that use it. Likewise, speed improvements or new functionality developments benefit all applications that use the DLLs.

Again, the need for a reconfigurable machine tool controller centers on the need to change controller functionality at run-time, depending upon which machine the CAD/CAM user chooses to perform a manufacturing process plan.

The functionality of a DLL makes it the perfect form for a mechanism device driver. By using DLLs, the mechanism-specific software modules can be designed, linked, and debugged independently. These DLLs are separate executable files and are completely independent of all other software. The reconfigurable controller paradigm is one of a mechanism device driver assigned to a mechanism class. The CAD/CAM applications can make a run-time decision to load a device driver DLL for a particular mechanism class or control application upon the user's request; thus making the mechanism assume different operating configurations depending on the number of axes, machine resolutions, and the relevant mechanism device driver functions. If the functionality of a mechanism device driver needs to be updated or enhanced, the driver developers only need to update this device driver DLL.

4.3.1 Device database

Each device driver has a self-contained device database, as shown in Table 4-2. The primary purpose for developing this device database is to allow CAD/CAM users to easily access any machine information before they select a particular machine to execute the manufacturing processes. The secondary purpose is to allow the device driver to correctly configure the RMAC controller based on information contained within the device database.

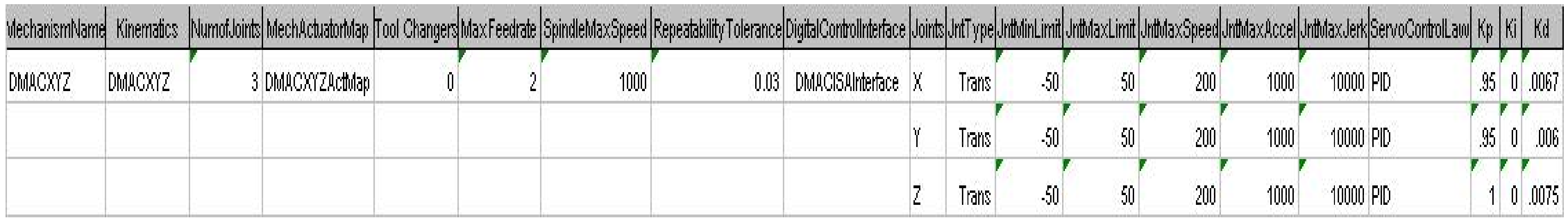
As shown in Table 4-2, the device database contains three categories of information that are relevant to a mechanism device: 1) primary machine characteristics, such as the tool changers and number and type of axes, etc; 2) machine operational parameters, such as mechanism maximum feedrate, spindle maximum RPM, mechanism positioning and repeatability tolerance, etc; 3) machine-specific motion planning and servo controlling capabilities, such as kinematics, servo control law and servo gains used on each axis, joint to actuator mapping, etc.

4.3.2 Device object

A *device object* data structure, shown in Fig. 4.8, serves two purposes. First, it assists CAD/CAM applications to easily access machine information contained within the device database. Second, it contains the exact information necessary for the device driver to set up a mechanism's operational parameters and reconfigure the RMAC controller for controlling this specific mechanism.

The *device object* data structure is declared in the device driver manager's header file and is shown in Appendix I. The following excerpt is an example of its structure:

Table 4-2 Mechanism device database



MechanismName	Kinematics	NumofJoints	MechActuatorMap	Tool Changers	MaxFeedrate	SpindleMaxSpeed	RepeatabilityTolerance	DigitalControlInterface	Joints	JntType	JntMinLimit	JntMaxLimit	JntMaxSpeed	JntMaxAccel	JntMaxJerk	ServoControlLaw	Kp	Ki	Kd
DMACXYZ	DMACXYZ	3	DMACXYZActMap	0	2	1000	0.03	DMACISAInterface	X	Trans	-50	50	200	1000	10000	PID	.95	0	.0067
									Y	Trans	-50	50	200	1000	10000	PID	.95	0	.006
									Z	Trans	-50	50	200	1000	10000	PID	1	0	.0075

```
typedef struct _DEVICE_OBJECT{

char            MechanismName[100];
int             NumOfJoints;
int             MechanismJointType[DMAC_MAX_JNT];
…
char            MechanismSpindleServoControlLaw[100];

}DEVICE_OBJECT, *PDEVICE_OBJECT;
```

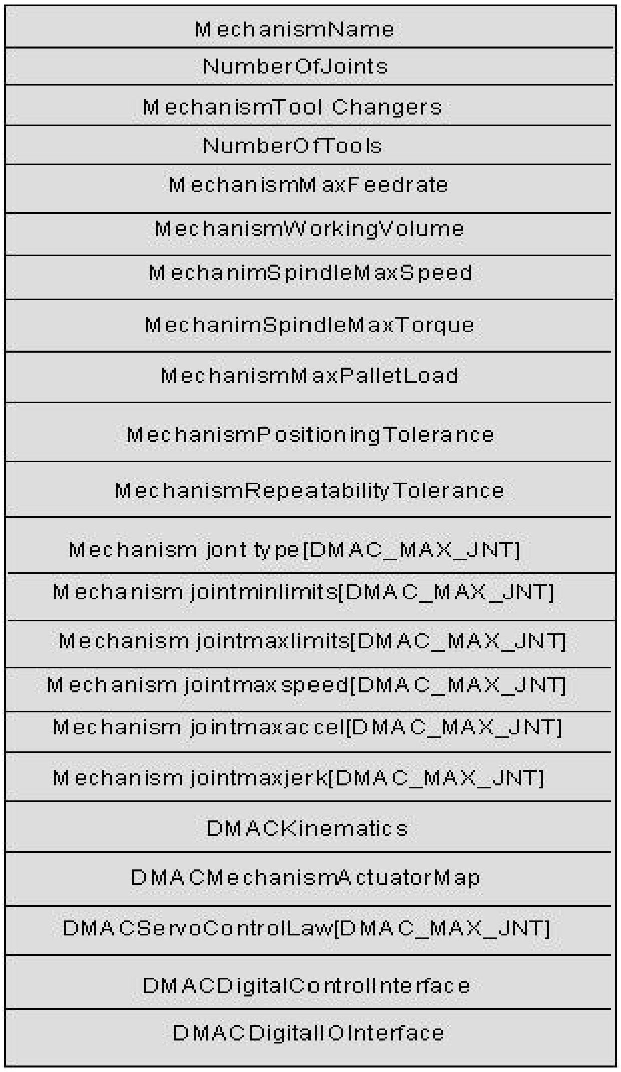

MechanismName
NumberOfJoints
MechanismTool Changers
NumberOfTools
MechanismMaxFeedrate
MechanismWorkingVolume
MechanimSpindleMaxSpeed
MechanimSpindleMaxTorque
MechanismMaxPalletLoad
MechanismPositioningTolerance
MechanismRepeatabilityTolerance
Mechanism jont type[DMAC_MAX_JNT]
Mechanism jointminlimits[DMAC_MAX_JNT]
Mechanism jointmaxlimits[DMAC_MAX_JNT]
Mechanism jointmax speed[DMAC_MAX_JNT]
Mechanism jointmaxaccel[DMAC_MAX_JNT]
Mechanism jointmaxjerk[DMAC_MAX_JNT]
DMACKinematics
DMACMechanismActuatorMap
DMACServoControlLaw[DMAC_MAX_JNT]
DMACDigitalControlInterface
DMACDigitalIOInterface

Fig. 4.8 Device_object data structure

The device driver uses the *device object* data structure to represent each mechanism device. In Fig. 4.8, each field corresponds to one column in the device database (see Table 4-2). Here, the device driver creates an instance of an ODBC object. It then uses this ODBC object to access any data defined within the device database. Upon obtaining this data, the device driver transmits it to the corresponding fields defined within the *device object.* After the *device object's* data structure is completely filled up, all of the mechanism-related information will be made available to CAD/CAM users upon their request. The first data field defined within this data structure is the mechanism name. The second to the seventeenth data fields all relate to the mechanism's characteristics. These sixteen fields of data serve two purposes. First, CAD/CAM users can access this information when they are evaluating a machine. Second, the device driver can use this data to correctly set up a machine before end users can operate that machine.

The last five fields all relate to the machine-specific motion planning, servo controlling capabilities, and the underlying communication hardware used by each machine.

For reconfiguring the RMAC controller to control different mechanisms, machine-specific software modules must be separated and encapsulated into DLLs, thus making the RMAC controller completely generic.

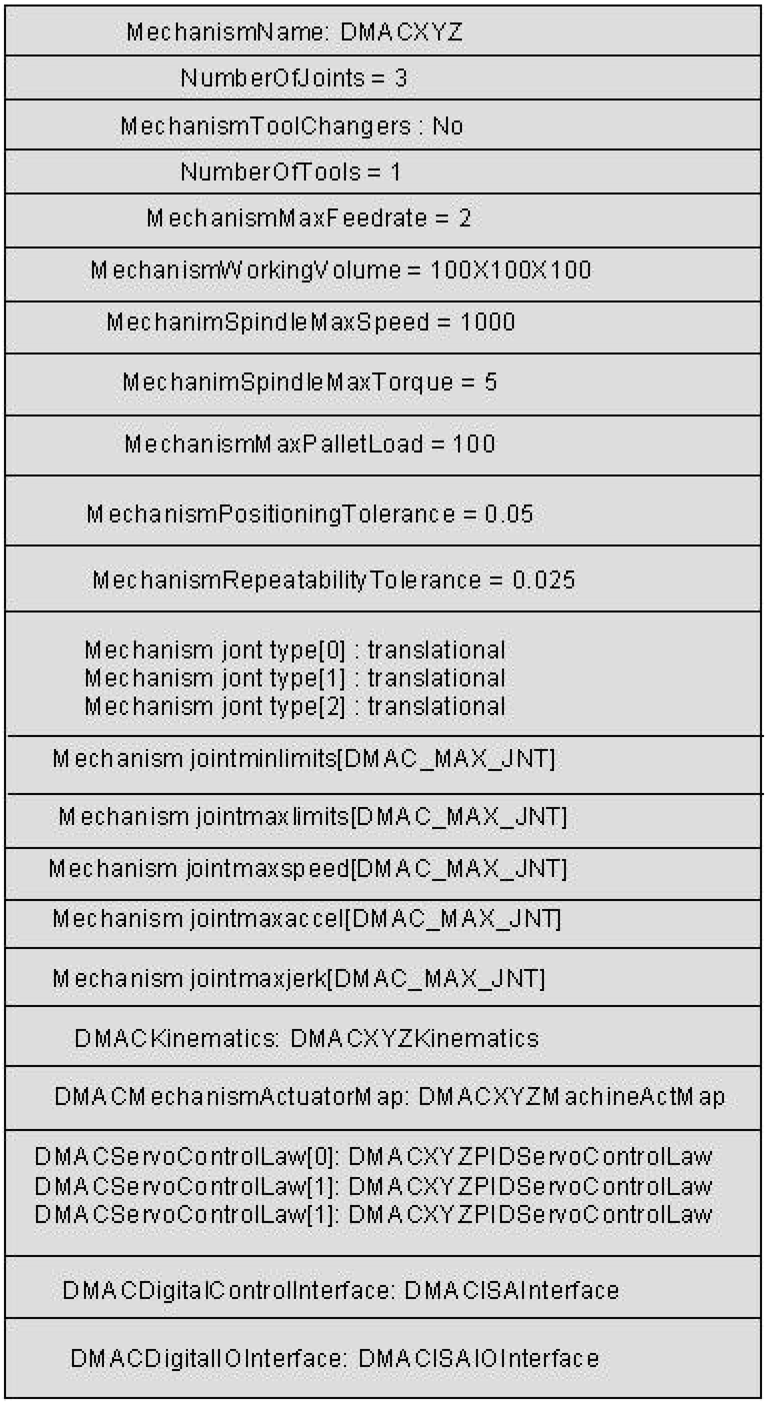
As described in section 3.2, to command a mechanism's tool to follow a desired tool path, a series of computations must be taken to generate the commanded torque values to send to each digital motor drive. As for different mechanisms or machine tools, some of these computations vary from one mechanism to another. Under the RMAC architecture, they are no longer designed within the RMAC controller, but are designed and built as separate DLLs that can be dynamically loaded and mapped into the RMAC controller upon a user's request. Since the RMAC controller no longer contains these mechanism-specific algorithms, it becomes critical for a device driver software to instruct the RMAC controller to map these algorithms from corresponding DLLs at run-time so that the RMAC controller's functionality is adaptable to a selected mechanism. The mapping method and the interface APIs that enable these mappings will be described in the subsequent sections.

4.3.3 Device object example for a three-axis mill

To better understand this *device object*, the following example is demonstrated. Once the three-axis mill device driver creates an instance of the *device object*, it searches the three-axis tabletop mill device database (see Table 4-2). By accessing the data defined within this database, the device driver correctly sets up the *device object* data structure, as shown in Fig. 4.9. The second to the seventeenth data fields are machine-specific operational parameters. For the mechanisms joint parameters, only joint type is shown in this figure.

The last five fields are machine-specific motion planning, servo controlling algorithms, and the underlying digital control interface—all necessary components for controlling this mechanism. As can be seen from this figure, the names of the DLLs corresponding to these machine-specific algorithms are mapped into this *device object* data structure. The device driver software will use the RMAC_Config interface APIs to pass the DLL names to the RMAC controller. The RMAC controller can then locate the

DLLs and import all the necessary motion planning and servo controlling algorithms into its controller software.



MechanismName: DMACXYZ

NumberOfJoints = 3

MechanismToolChangers : No

NumberOfTools = 1

MechanismMaxFeedrate = 2

MechanismWorkingVolume = 100X100X100

MechanimSpindleMaxSpeed = 1000

MechanimSpindleMaxTorque = 5

MechanismMaxPalletLoad = 100

MechanismPositioningTolerance = 0.05

MechanismRepeatabilityTolerance = 0.025

Mechanism jont type[0] : translational
Mechanism jont type[1] : translational
Mechanism jont type[2] : translational

Mechanism jointminlimits[DMAC_MAX_JNT]

Mechanism jointmaxlimits[DMAC_MAX_JNT]

Mechanism jointmaxspeed[DMAC_MAX_JNT]

Mechanism jointmaxaccel[DMAC_MAX_JNT]

Mechanism jointmaxjerk[DMAC_MAX_JNT]

DMACKinematics: DMACXYZKinematics

DMACMechanismActuatorMap: DMACXYZMachineActMap

DMACServoControlLaw[0]: DMACXYZPIDServoControlLaw
DMACServoControlLaw[1]: DMACXYZPIDServoControlLaw
DMACServoControlLaw[1]: DMACXYZPIDServoControlLaw

DMACDigitalControlInterface: DMACISAInterface

DMACDigitalIOInterface: DMACISAIOInterface

Fig. 4.9 DMACXYZ device object

Because this mill is a three-axis mechanism, each of the kinematic axis needs a specific servo control algorithm. For this case, the X, Y, and Z axes all use the same PID servo control algorithm. Thus, if the Z axis uses a different servo control algorithm, such as a force control algorithm, the device driver maps a different servo control DLL name into this data structure, resulting in a flexible architecture that allows for easy reconfiguration of different servo control methods into the RMAC controller.

4.3.4 Interface to CAD/CAM

A device driver is run-time loaded by CAD/CAM applications upon a user's request. The communication between the device driver and CAD/CAM systems are through a set of device driver interface APIs.

The interface APIs are divided into three groups: 1) functions that allow CAD/CAM systems to access the device database, 2) functions that return this specific machine information back to CAD/CAM, and 3) functions that instruct the RMAC open-architecture controller to set up correct operational parameters and configure the motion planning and servo controlling specific to this mechanism.

The first two groups of interface APIs share similarities with the interface APIs between the device driver manager and CAD/CAM systems. CAD/CAM applications can call the first group of interface APIs to operate on a device database. Two similar interface API examples (with no input parameters) are given below.

- OpenDeviceDatabase()
- CloseDeviceDatabase()

Function *OpenDeviceDatabase* allows the CAD/CAM applications to connect to the device database. Function *CloseDeviceDatabase* disconnects the database from CAD/CAM.

The second group of interface APIs are used to obtain information related to a specific mechanism. One example, *Machine_GetMechanismNumOfJoints(),* returns the number of joints to CAD/CAM applications.

CAD/CAM applications use the third group of interface APIs to instruct the RMAC controller to set up correct machine parameters and reconfigure its controller software prior to executing a manufacturing process plan. Four examples, with no input parameters, are listed as follows:

- Machine_SetMechanismMaxFeedrate()
- Machine_ConfigureMotionPlanner()
- Machine_ConfigureServoController()
- Machine_ConfigureDigitalControlInterface()

CAD/CAM applications use function *Machine_SetMechanismMaxFeedrate()* to instruct the RMAC controller to set up the mechanism's maximum federate. The last three APIs are generic functions that CAD/CAM applications can call to instruct the RMAC controller to reconfigure its controller software modules. Under the RMAC architecture, various device drivers can be connected to a CAD/CAM application. Therefore, CAD/CAM applications must use generic APIs to communicate with different device drivers. The actual interpretation of these functions is done internally in the driver software. Based on the information contained within a device object (see Fig. 4.8), the device driver then calls the relevant RMAC_Config interface APIs to correctly configure the RMAC controller.

Appendix III provides a list and description of these interface APIs.

4.3.5 Interface to the RMAC reconfigurable controller

A device driver connects CAD/CAM applications to the RMAC reconfigurable controller. It receives process instructions from CAD/CAM and then passes them to the RMAC controller.

The device driver software uses two COM interfaces, RMAC_Config and RMAC_CAM, to communicate with the RMAC controller. It uses the RMAC_Config interface APIs to instruct the RMAC reconfigurable controller to correctly set up machine parameters and to map mechanism-specific motion planning and servo controlling algorithms into its controller software. Once the device driver software finishes reconfiguring the RMAC controller, it then uses the RMAC_CAM interface APIs to pass process instructions to the RMAC controller. The device driver software contains an instance of the RMAC_Config and RMAC_CAM objects. As a result, the interface APIs contained within these two COM interfaces are directly available to the device driver.

4.4 RMAC_Config Interface

To connect any control input to the RMAC reconfigurable controller, multiple COM-based control plug-ins have been developed. This COM-based control plug-in, developed by Direct Controls, Inc., acts as the interface between the RMAC reconfigurable controller and an external control source. Fig. 4.10 shows a few existing

COM interfaces that have been implemented. It also shows a newly developed COM interface that allows for reconfigurable control.

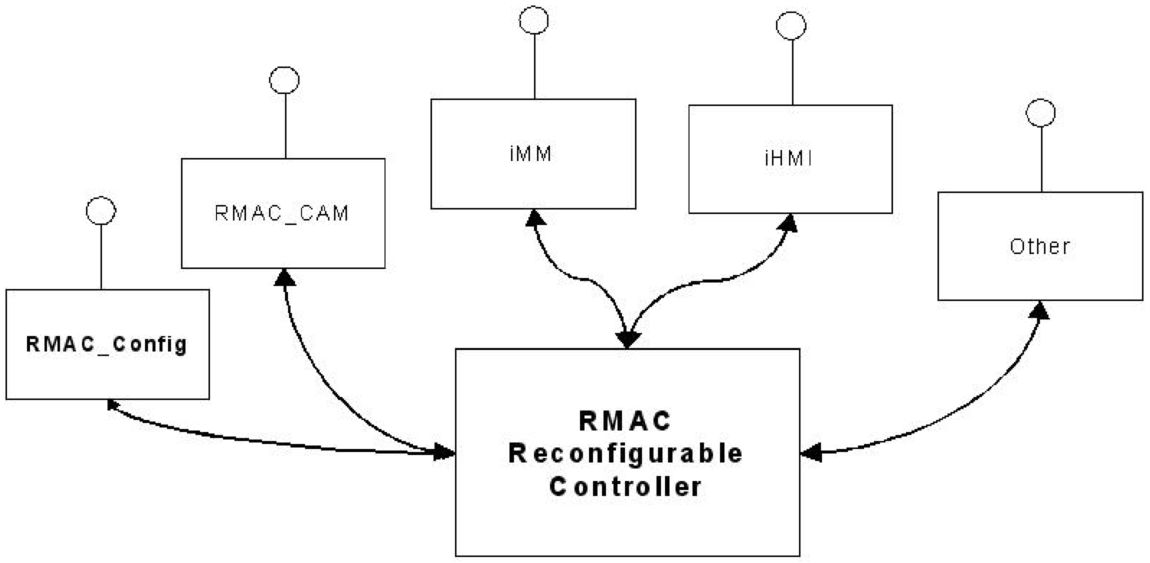


Fig. 4.10 COM-based plug-ins connected to theRMAC reconfigurable controller

The design strategy of developing a separate RMAC_Config interface that is completely independent of the RMAC_CAM interface separates the machine reconfiguration and direct machining process instructions into two COM interfaces; thus the RMAC_CAM interface can still be used on any existing DMAC controllers. The RMAC_CAM COM interface is a separate plug-in that can be connected to the RMAC controller when necessary. It contains well-defined function calls that allow a device driver to instruct the RMAC controller to reconfigure its motion planner, servo controller, and digital control interface, which, in turn, is necessary to control a particular RMAC-compliant mechanism.

The interface functions contained within the RMAC_Config interface are divided into three groups. The first group of interface APIs reconfigures the motion planner. The set of functions within this group contains methods to allow the device driver to instruct the RMAC controller to map mechanism-specific kinematics and machine actuator mapping algorithms from the corresponding DLLs, or to correctly set up machine parameters. The second group of interface APIs reconfigures the servo controller. Each kinematics joint may need a specific control method and servo control algorithm. The interface APIs within this group contain the methods to configure each kinematics joint with a desired servo control algorithm. The last group of interface APIs reconfigures the digital control interface and digital I/O interface.

The provided functions with no description (see Appendix IV – for description) are as follows:

Group 1: Configure motion planner

1. SetMechanismKinematics
2. SetMechanismActuatorMap
3. SetMechanismNumJnts
4. SetMechanismNumSpindle
6. SetMechanismJointType
7. SetMachanismJointLimitType
8. SetMechanismJointMaxLimit
9. SetMechanismJointMinLimit
10. SetMechanismJointMaxSpeed
11. SetMechanismJointMaxAcceleration
12. SetMechanismJointMaxJerk
13. SetMechanismSpindleMaxRPM

Group 2: Configure servo controller

1. SetMechanismNumActuators
2. SetMechanismJointControlMethod
3. SetMechanismJointServoControlLaw
4. SetMechanismJointServoControlGains
5. SetMechanismSpindleControlMethod
6. SetMechanismSpindleServoControlLaw
7. SetMechanismSpindleServoControlGains

Group 3: Configure digital control interfa

1. SetMechanismDigitalControlInterface
2. InitializeMechanismDigitalControlInterface
3. SetMechanismDigitalIOInterface
4. InitializeMechanismDigitalIOInterface

4.5 RMAC_CAM Interface

After a device driver instructs the RMAC controller to reconfigure its control software components, the RMAC controller is ready to take motion and control commands for direct machining.

Once control commands have been accepted by the device driver software, they are ready to be passed to the RMAC controller. The device driver communicates with the RMAC controller through a COM interface called RMAC_CAM. RMAC_CAM is directly interfaced with the motion planner, and allows the device driver to pass motion commands to the RMAC controller as well as to send and receive other control information.

The interface APIs defined within the RMAC_CAM COM interface are divided into two groups. The first group of APIs relates to the milling machine; the second group relates to the CMM.

The milling machine APIs are further divided into two sub groups. The first group is used by the device driver to instruct the RMAC controller in executing manufacturing process plans. The second group allows the device driver to obtain current operational parameters, such as the current spindle speed.

Similarly, the interface APIs defined for the CMM are also divided into two sub groups. The first group of APIs relates to a measurement process plan; the second group relates to current machining operational parameters.

Appendix IV provides a list and description of the RMAC_CAM interface APIs.

4.6 RMAC Reconfigurable Controller

The existing DMAC architecture does not permit reconfigurable control. To control a particular machine, any machine-specific functions or classes currently must be designed and built into a DMAC controller. After these machine-specific modules are linked to the DMAC controller, they become a permanent part of the DMAC system. As a result, the DMAC controller becomes unchangeable at run-time for controlling different machines. For instance, DMAC can control a three-axis tabletop mill, a three-axis Sugino mill, and a five-axis Tarus mill. However, each of these controllers is tailored for a particular machine. End users cannot use the three-axis Sugino mill's controller to control the five-axis Tarus mill.

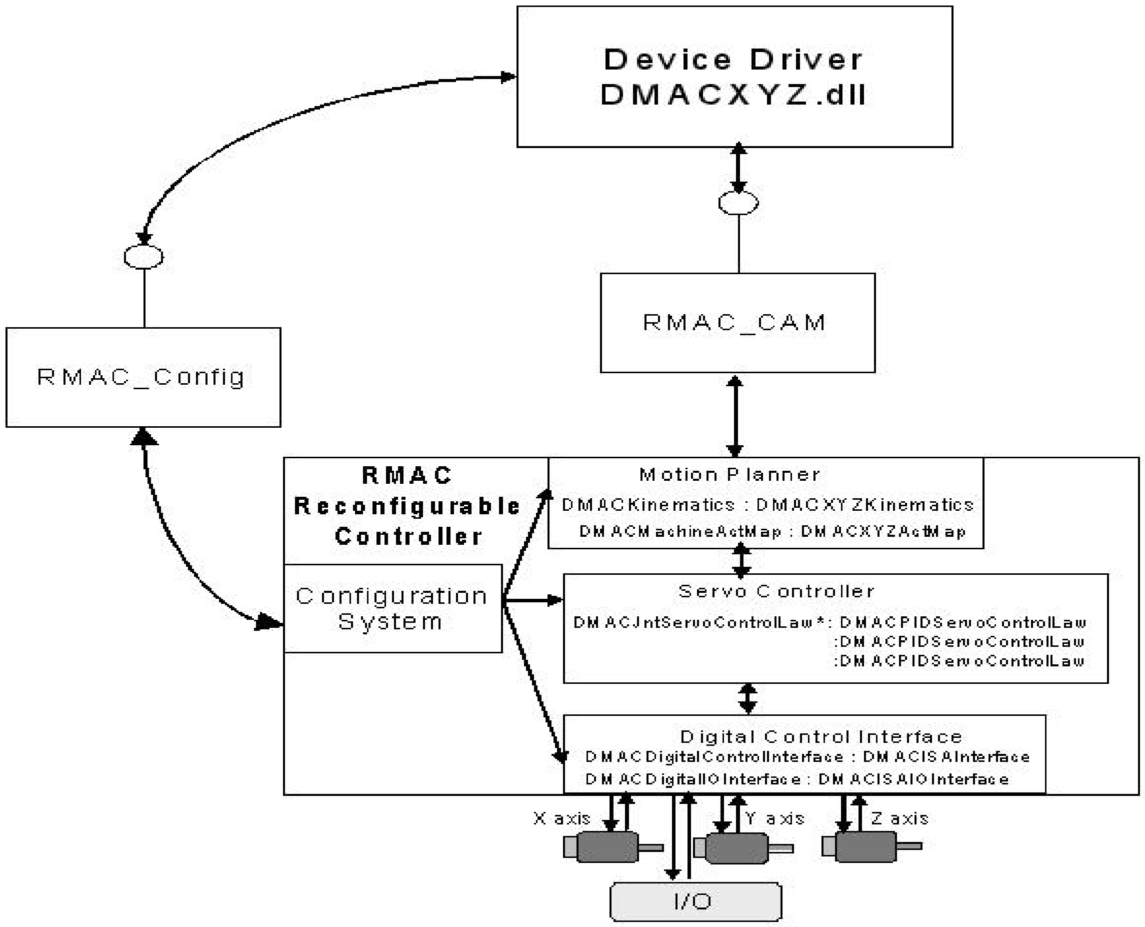
To allow reconfiguration of the motion planner and servo controller necessary for controlling different mechanisms, some additions and modifications must be made to DMAC's existing architecture, as shown in Fig. 3.6.

As described in section 4.3, any machine-specific module is designed and linked separately into a DLL. This module must first be mapped into the RMAC controller before CAD/CAM applications can send down process plans to the RMAC controller for direct machining. The software module is mapped into the RMAC controller through a configuration system, as displayed in Fig. 3.6 and Fig. 4.11.

The configuration system is directly interfaced with the RMAC_Config COM interface so that it can receive configuration commands from this interface. Based on these different configuration instructions, the configuration system will do one of two operations. It will either set up a correct machine operational parameter, such as machine joint limits; or load the corresponding DLLs and then map any mechanism-specific module, such as the machine kinematics object, into the RMAC controller.

After the configuration system finishes all of these configuration processes, the RMAC controller is dynamically reconfigured for a particular mechanism. CAD/CAM applications can then pass the manufacturing process instructions to the RMAC controller for direct machining.

To better understand how this reconfiguration process occurs, Fig. 4.11 demonstrates how the RMAC controller is reconfigured for a three-axis tabletop mill.



Device Driver
DMACXYZ.dll
RMAC_CAM
RMAC_Config
RMAC Reconfigurable Controller
Configuration System
Motion Planner
DMACKinematics : DMACXYZKinematics
DMACMachineActMap : DMACXYZActMap
Servo Controller
DMACJntServoControlLaw*: DMACPIDServoControlLaw
:DMACPIDServoControlLaw
:DMACPIDServoControlLaw
Digital Control Interface
DMACDigitalControlInterface : DMACISAInterface
DMACDigitalIOInterface : DMACISAIOInterface
X axis
Y axis
Z axis
I/O

Fig. 4.11 Reconfiguring the RMAC controller for a three-axis tabletop mill

Here, the motion planning, servo controlling, and digital control interface modules, which are specific to this three-axis mechanism, are all designed and linked as separate DLLs. To correctly control this three-axis tabletop mill, the RMAC controller needs to map these mechanism-specific software modules into the controller software.

The device driver software, in this case the DMACXYZ.dll, uses the following RMAC_Config interface function calls to pass these DLL names to the RMAC controller. For instance, the driver software (DMACXYZ.dll) calls the interface function *SetMechanismKinematics(char* pDMACKin)* to pass the kinematics DLL name (DMACXYZKinematics.dll) to the RMAC controller. Upon receiving this name, the RMAC controller locates the DLL and loads it into the memory. It then imports the corresponding kinematics class from this loaded DLL. For convenience, this three-axis mill kinematics class is also named DMACXYZKinematics. Likewise, the RMAC controller can import all other necessary machine-specific software modules (at run-time) for controlling this three-axis mill.

```
SetMechanismKinematics(char* pDMACKin)
SetMechanismActuatorMap(char* pDMACMachineActMap)
SetMechanismJointControlMethod(int JntNum, SHORT DMACServoControlMethod)
SetMechanismJointServoControlLaw(int JntNum, char* pControlLaw)
SetMechanismDigitalControlInterface(char* pMotorInterface)
SetMechanismDigitalIOInterface(char* pIOInterface)
```

In addition to mapping these mechanism-specific software components into the RMAC controller, the device driver software must instruct the RMAC controller to correctly set up the machining operational parameters that are specific to this three-axis mechanism. For instance, the device driver obtains the mechanism maximum feedrate from the *device object* data structure and uses the following RMAC_Config function call to correctly set up the RMAC controller:

SetMechanismMaxFeedrate(double MaxFeedrate)

After this reconfiguration process, the necessary mechanism-specific algorithms are mapped into the RMAC controller, as shown in Fig. 4.11, and the machining operational parameters are correctly set up. At this point, the RMAC controller is dynamically adapted, and therefore capable of controlling the three-axis mechanism.

4.7 Simulation System

As introduced in section 4.1, a simulation system is developed inside CAD/CAM systems, thus, enabling end users to debug and validate a controller's functionality without operating the real machine tool when the controller is reconfigured.

CAD/CAM users can set the device driver to work in a virtual mode (see Fig. 4.4). The only difference between virtual mode and real control mode is that the servo algorithms reflect the motion directly back to CAD/CAM systems as kinematics joint values. These joint values are used by CAD/CAM systems to update the animation display. UG has a built-in machine simulator allowing UG users to build a virtual machine. This virtual machine can be simulated by taking the kinematics joint setpoints at each trajectory step. CATIA does not have a built-in machine simulator, but it can be integrated with DELMIA. Similar to UG, DELMIA users can build a virtual machine that can be animated by taking the motion setpoints from the RMAC controller through a device driver interface.

Whenever a user or CAD/CAM applications switches the controller to a different mechanism, the simulation system is used to run the machining process plans on a virtual machine to validate the controller's functionality.

The simulation system serves two purposes. First, it enables CAD/CAM users to estimate cycle time for a manufacturing process plan. Second, the simulation system checks any collisions that might occur during a process plan. This, in turn, assists CAD/CAM users in verifying a machine tool before they run the manufacturing process plan directly on a physical machine.

Once the controller's functionality is verified, the machining process plans can be sent down to the RMAC controller to execute the process on a physical machine.

4.8 New Sequences of Operations

Under the RMAC architecture, the sequence of control information flow is as follows:

1. CAD/CAM users create a solid model.
2. Users generate the manufacturing process plan.
3. Users search and select a best machine tool to perform the process plan.
4. The device driver manager locates the device driver and informs it to CAD/CAM.
5. CAD/CAM loads the device driver and communicates directly with the driver.
6. The device driver uses the RMAC_Config interface to instruct the RMAC controller to reconfigure its motion planner, servo controller, and digital control interface for the selected machine.

7. CAD/CAM applications pass the process plans to the device driver.
8. The device driver interprets the process plans and uses the RMAC_CAM interface to pass motion and control commands to the RMAC controller for direct machining.
9. The RMAC controller executes the manufacturing process plans and operates the machine to make a desired part.

With the development of the RMAC architecture, it changes the traditional machine control method. Under this new paradigm, CAD/CAM users can select an optimal machine to meet the specific needs of a manufacturing operation. A device driver will then be automatically loaded to connect the select machine tool to the CAD/CAM system, thus allowing the manufacturing process plans to be executed directly on the selected machine tool. The post-processing of these process plans into the traditional M&G files is no longer necessary.

CHAPTER 5 PROTOTYPE IMPLEMENTATION

This chapter first shows the successful implementation of the RMAC architecture on a three-axis tabletop mill. It then shows the proposed implementation of RMAC on a Tarus five-axis full-size mill, and a Coordinate Measuring Machine (CMM). Lastly, it demonstrates the simulation of a three-axis tabletop mill and a five-axis full-size mill in UG using the developed RMAC controller.

5.1 Control Hardware

5.1.1 Three-axis tabletop mill

To implement this reconfigurable controller on a three-axis tabletop mill, a device driver was developed to connect it directly with CATIA.

The controller runs on a Dual-Pentium 1 GHz computer. One processor runs Windows XP, under which CATIA operates; the second processor uses VentureCom's

Real-time Extensions (RTX) for Windows XP, using multiprocessor version 5.1.1. This makes machine tool control possible by giving the controller a real-time environment.

The machine tool is a three-axis tabletop mill shown in Fig. 5.1. Each mill axis is controlled by a digital torque drive developed by Semifusion, Inc. These digital torque drives send and receive digital data via two fiber-optic cables that connect them to the computer. The controller software uses these digital torque drives as torque slaves, sending commanded torque to the motors and receiving actual position, speed, torque, current, and errors as feedback, all as digital information. In the future, this mill will use IEEE 1394 Firewire to replace its current fiber optic communication protocol.

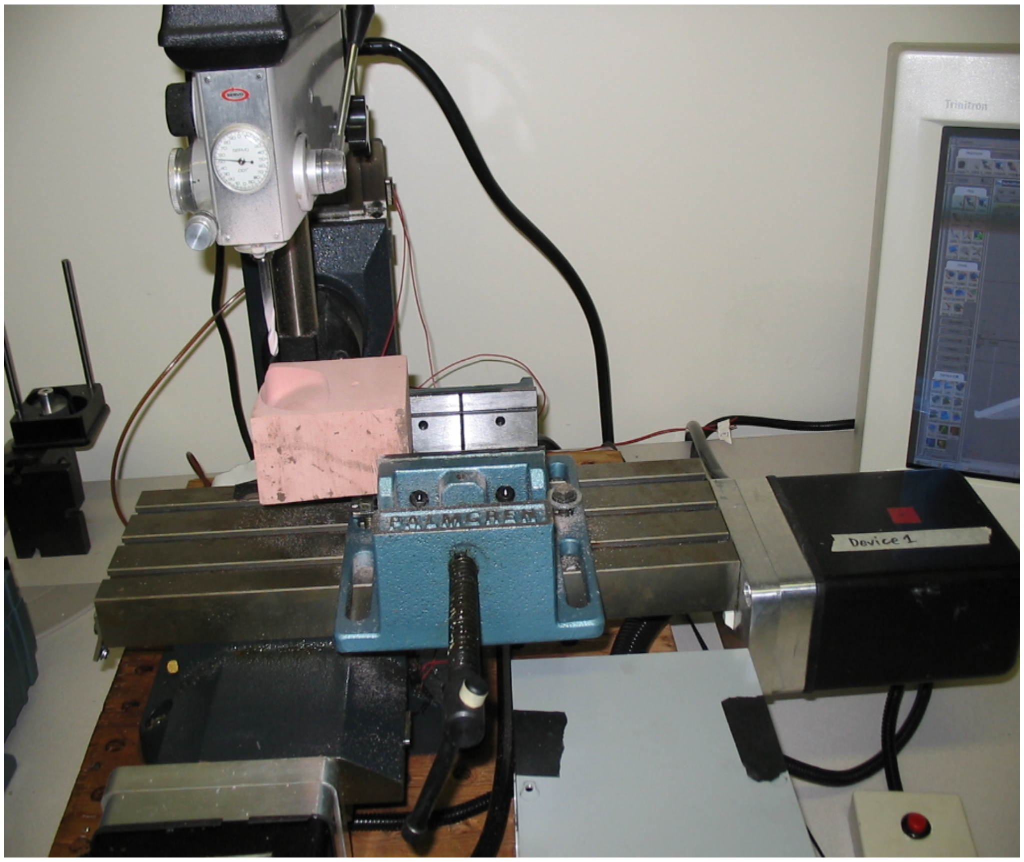


Fig. 5.1 Three-axis tabletop mill

5.1.2 Five-axis full-size Tarus mill

Aside from the three-axis table top mill, this research will also be implemented on a five-axis full-size Tarus mill, donated by General Motors (GM). A device driver to connect this mechanism directly with CATIA needs to be completed.

Fig. 5.2 shows this Tarus mill. Each motor is controlled by an ORMEC ServoWire SM digital drive. Communication between the RMAC controller and each digital motor drive is handled by the IEEE 1394 Firewire, which is able to send information fast enough to control multiple motors at rates above 4000 Hz.

I/O connections are handled through an ICP DAS data acquisition system. The system consists of a PCI card (model no: PIO-D48) that is mounted in the host computer, and a daughter board (model no: DB-24P) that connects to the PCI card and provides the actual I/O connections.

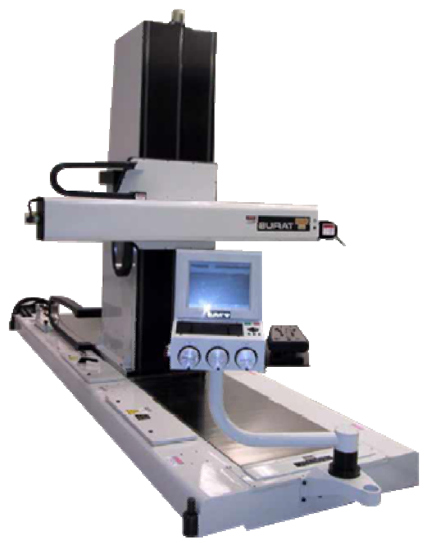


Fig. 5.2 Five-axis full-size Tarus mill

5.1.3 CMM

The third implementation of this research will be on a Sugino V9, donated by Sugino Machine, Inc. of Japan, for in-cycle measurement.

The physical mill is a three-axis mill shown in Fig. 5.3. Its digital motor communication protocol, IEEE 1394 firewire, and its I/O connections are the same as those previously mentioned for the five-axis Tarus mill. As a result, the controller software uses the same digital control interface and digital I/O interface to communicate with the digital devices.

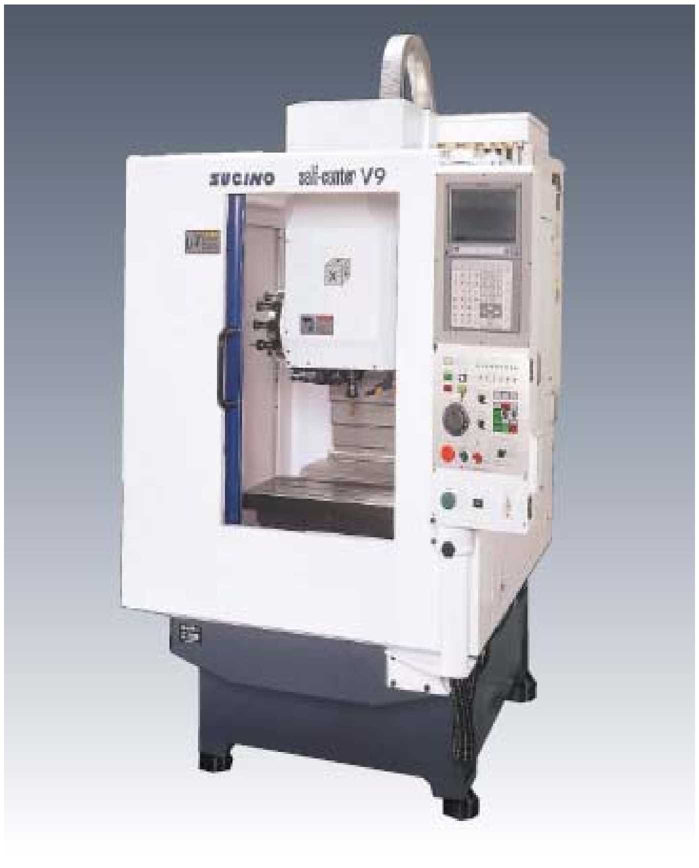


Fig. 5.3 Sugino V9 for CMM

5.2 Control Software System

5.2.1 DMACXYZ tabletop mill device driver

Fig. 5.4 diagrams the overall control software system. The major development effort to control this mechanism was to develop a device driver, named DMACXYZ.dll, to connect this mill directly to CAD/CAM systems.

A mechanism-specific device database, named DMACXYZ.mdb, has been designed to contain detailed information about this mechanism device. Three separate DLLs were designed to contain mechanism-specific kinematics, machine actuator mapping, and servo control algorithms. These three DLLs are DMACXYZKinematics.dll, DMACXYZMachineActMap.dll, and DMACXYZPIDServoControlLaw.dll.

In addition, DMACISAInterface.dll and DMACISAIOInterface.dll were developed. These two DLLs contain the API functions to handle communication among the RMAC controller, the digital motor drives, and the I/O sensors. The device driver software contains the interface APIs to instruct the RMAC controller to map these mechanism-specific algorithms into its memory space, which is necessary for controlling this mechanism.

The device driver DMACXYZ.dll, the device database DMACXYZ.mdb, and the other five DLLs are located in the three-axis folder, which is subordinate to the milling machine folder, as described in section 4.2.1.

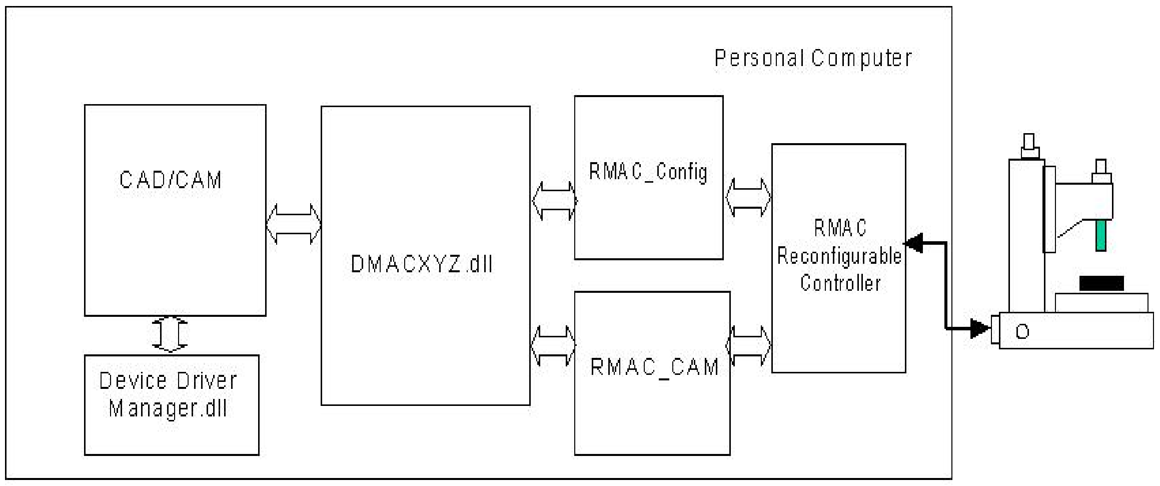


Fig. 5.4 DMACXYZ tabletop mill device driver

5.2.2 TarusXYZCA five-axis mill device driver

Fig. 5.5 shows the diagram of the proposed overall control software system for the Tarus five-axis mill. To control this mechanism, a device driver, named TarusXYZCA.dll, must be developed.

The development work for this device driver shares some similarities with that of the DMACXYZ device driver described in the previous section. A mechanism-specific device database, named TarusXYZCA.mdb, must be developed to contain detailed information about this mechanism device. TarusXYZCAKinematics.dll, TarusXYZCAMachineActMap.dll, TarusXYZCAPFFServoControlLaw.dll, TarusXYZCAFirewireInterface.dll, and TarusXYZCAICPDasIOInterface.dll are under development. These DLLs contain the kinematics, machine actuator mapping, and servo control algorithms that are specific to this mechanism. In addition, the last two DLLs contain a set of APIs that can be mapped into the RMAC controller to allow it to communicate with the digital motor drives and the I/O sensors.

The device driver TarusXYZCA.dll, the device database TarusXYZCA.mdb, and the other five separate DLLs are placed in the five-axis folder, as shown in Fig. 4.6.

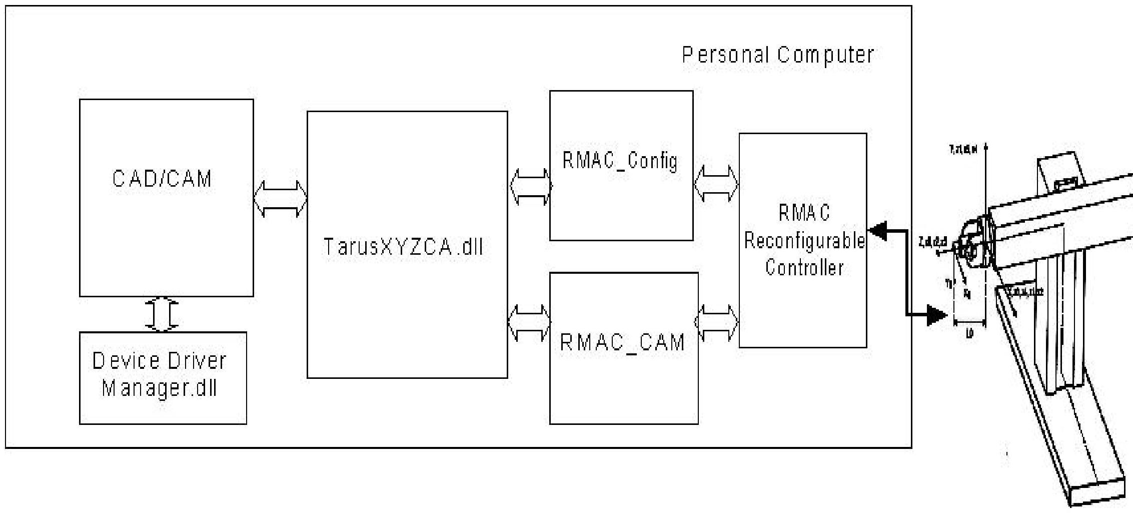


Fig. 5.5 TarusXYZCA full-size mill device driver

5.2.3 CMM device driver

Fig. 5.6 diagrams the proposed overall software system for CMM control. To create direct CMM control, it was necessary to develop a generic, high level driver named WADriver.dll, and a machine-specific low level driver named WAILLDriver.dll. These connect the Sugino V9 directly with the PC-DMIS. Further, the WADriver exposes generic function calls to allow the PC-DMIS to command the CMM generically. The low level driver (WAILLDriver) interprets the generic functions and sends down the motion or control commands required by the RMAC controller.

The CMM high level driver (WADriver.dll) and low level driver (WAILLDriver.dll) have been developed and implemented. Further work needs to be done to integrate these drivers to the overall software system shown in Fig. 5.6.

The CMM driver allows the RMAC controller to act as a CMM controller, regardless of what type of machine is actually being controlled. This flexibility allows the RMAC controller to pause the cutting process, and switch to a measurement mode. The part can then be measured in-process, and the results can be used immediately to update the manufacturing process, and to compensate for measured errors.

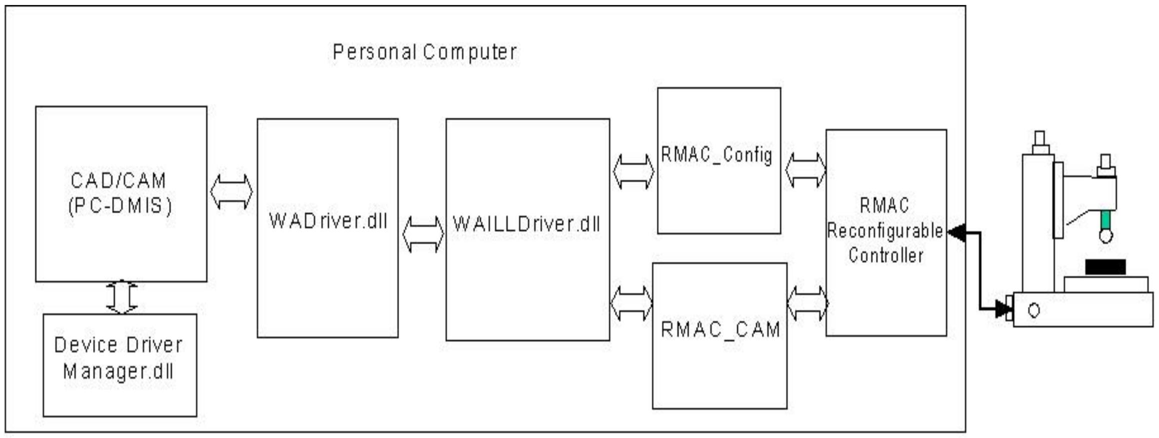


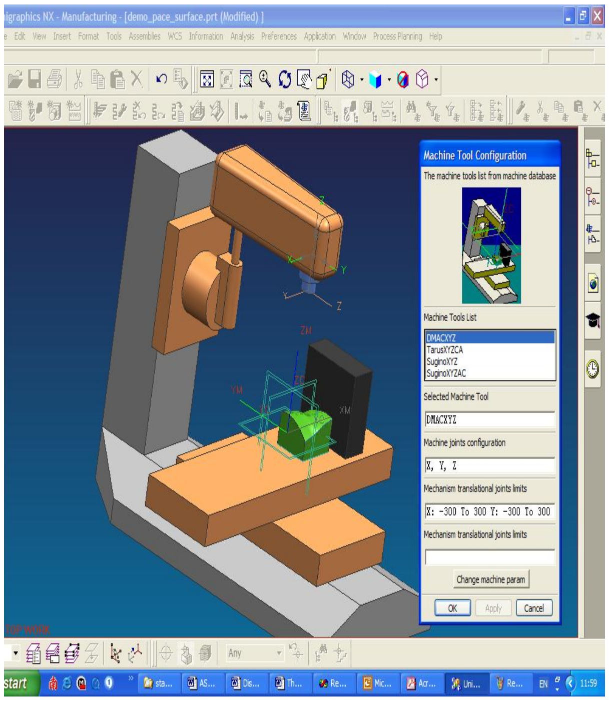
Fig. 5.6 CMM device driver

The CMM driver uses the kinematics currently loaded for the machine under control. Because of this, the driving software remains blind to the type of machine under control and thus simply passes motion commands as normal. The RMAC CMM driver then provides functionality for commands related to measurement. Unique commands are included for measuring a part manually or automatically, obtaining the position of the measured hit, switching the probe on and off, checking for measurement errors, and setting measurement parameters. By adding these functions, any controlled machine can be used as a CMM.

While many controllers offer some CMM functionality, their commands are limited to the basic M&G commands that manufacturers have implemented, giving them limited abilities to communicate the results. The RMAC CMM driver allows an external CMM software package to obtain complete control of the machine. The first implementation of this driver has been demonstrated by utilizing the PC-DMIS measurement software package. PC-DMIS is the world leader in measurement software, and provides many advanced capabilities for part measurement and result reporting. By using this software, RMAC can offer a capable and familiar CMM user interface for measurement specialists.

5.3 Simulation

A simulation for implementing the three-axis tabletop mill and Tarus five-axis mill device drivers has been successfully created in Unigraphics (UG), as shown in Fig. 5.7 and Fig. 5.8. A machine selection user interface (see Fig. 4.2) has also been developed inside UG, providing end users with a graphic tool to dynamically reconfigure the controller, and to change machine tool parameter settings. As displayed, whenever a user picks up a new machine, the CAD/CAM software will unload the old device driver DLL, and load the new device driver DLL into its memory. By importing the necessary motion planning and servo controlling algorithms into the RMAC controller, the functionality of the RMAC controller changes from a three-axis to a five-axis mill at run-time.


nigraphics NX - Manufacturing - [demo_pace_surface.prt (Modified)]
Machine Tool Configuration
OK
Apply
Cancel

The machine tools list from machine database

Machine Tools List

DMACXYZ
TarusXYZCA
SuginoXYZ
SuginoXYZAC

Selected Machine Tool: DMACXYZ

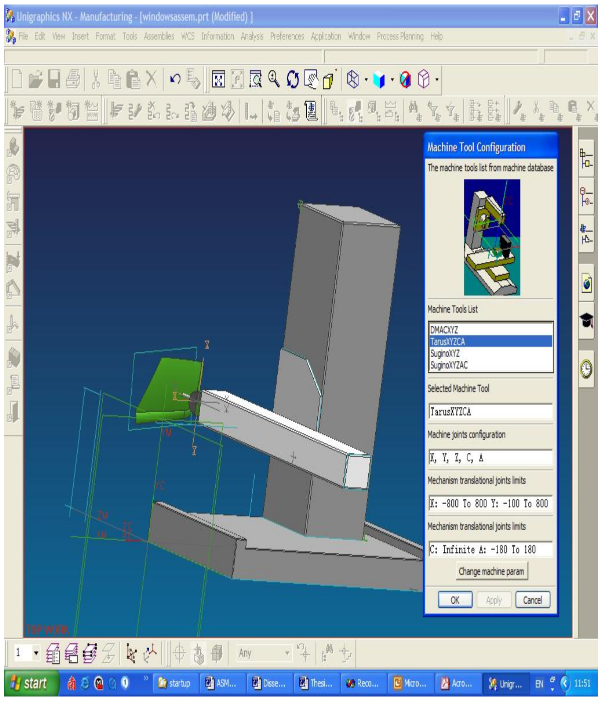
Machine joints configuration: X, Y, Z

Mechanism translational joints limits: X: -300 To 300 Y: -300 To 300

Mechanism translational joints limits:

Change machine param

Fig. 5.7 DMACXYZ simulation


Unigraphics NX - Manufacturing - [windowsassem.prt (Modified)]
Machine Tool Configuration
OK
Apply
Cancel

The machine tools list from machine database

Machine Tools List

DMACXYZ
TarusXYZCA
SuginoXYZ
SuginoXYZAC

Selected Machine Tool: TarusXYZCA

Machine joints configuration: X, Y, Z, C, A

Mechanism translational joints limits: X: -800 To 800 Y: -100 To 800

Mechanism translational joints limits: C: Infinite A: -180 To 180

Change machine param

Fig. 5.8 TarusXYZCA simulation

CHAPTER 6 Experimental Results

The final objective of this dissertation is to implement the RMAC architecture on a prototype mill. This chapter presents the experimental results of the prototype developed, as explained in chapter four.

6.1 Simulation

As part of this research, a simulation program was developed in UG (see Fig. 5.7 and Fig. 5.8). A machine selection user interface was also developed to allow end users to select different machines and dynamically change the controller's functionality from the three-axis mill to the five-axis mill. Experiments illustrated that the machine device drivers can be successfully unloaded and loaded when users selected different machines to run simulation. These experiments proved that dynamically reconfiguring the controller software by loading a machine device driver is feasible.

6.2 Three-axis Tabletop Mill Experiment

After a device driver was developed to connect a three-axis tabletop mill (see Fig. 5.1) to CATIA, two experiments were conducted to directly machine a CATIA surface on the three-axis tabletop mill. The following sections describe this process in greater detail.

6.2.1 Direct reconfigurable machining application start-up

To actuate the direct reconfigurable machining application, a direct reconfigurable machining tool bar is plugged into the CATIA surface machining workbench. This customized tool bar interacts between end users and the CATIA system. To launch the direct reconfigurable machining application in CATIA, users must select the *surface machining* option from the *NC Manufacturing* menu (see Fig. 6.1). The direct reconfigurable machining tool bar will then be automatically launched, as shown in Fig. 6.2.

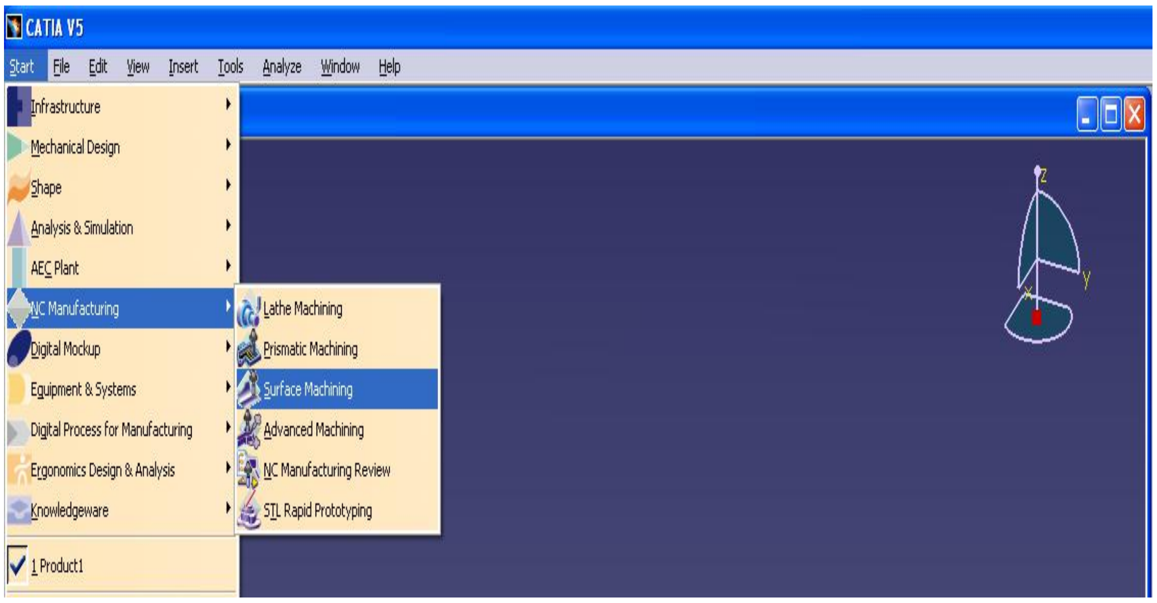


Fig. 6.1 Start up direct reconfigurable machining application

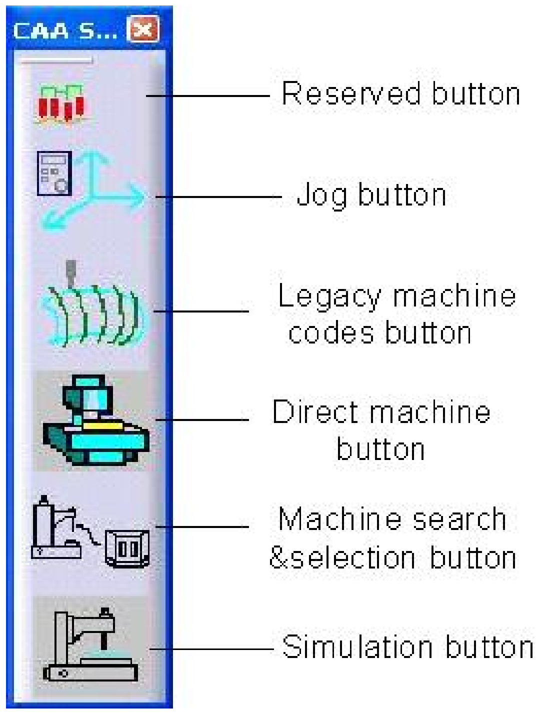


Fig. 6.2 Direct reconfigurable machining tool bar

This direct reconfigurable machining tool bar is composed of the following six buttons:

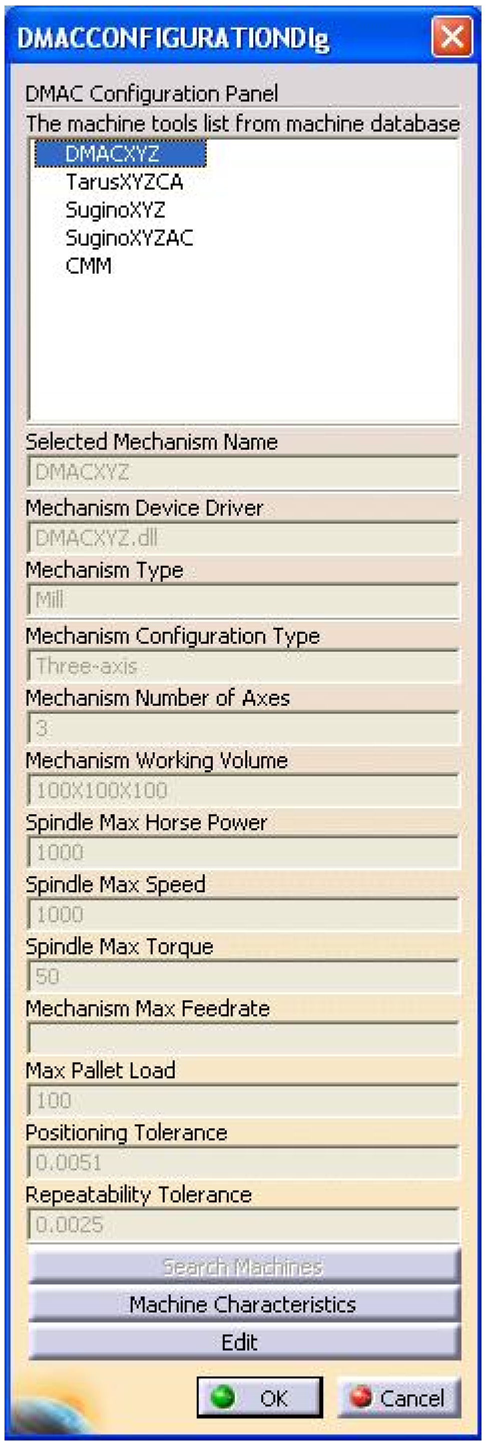
- Reserved button: reserves for future direct machining applications. Currently, when users click this button, a customized dialog box launches to allow them to select a surface. When users select a surface, a new dialog box pops up to prompt them to create tool paths. At this development stage, no customized tool path planning algorithm has been implemented yet. But for future direct machining applications, some advanced tool path planning algorithms, such as curvature matched machining, can be implemented.

- Jog button: allows users to launch a jog dialog box. Users can then employ this dialog box to position the cutter at any X, Y, and Z position in relation to a coordinate system on a 3-axis mill.
- Legacy machine codes button: allows users to read legacy machine codes, such as APT and M&G codes, and execute them on a machine tool.
- Direct machine button: allows users to send the tool paths directly to the RMAC controller to create a physical model.
- Machine search and selection button: allows users to launch a customized machine search and selection dialog box. Based on certain searching schemes, users can narrow down their machine selections to choose one machine capable of performing the manufacturing process.
- Simulation button: allows users to preview a process plan on a virtual machine prior to executing the manufacturing process on a physical machine.

6.2.2 Machine search and selection dialog box

Once CAD/CAM users create a manufacturing process plan and launch the direct machining tool bar (see Fig. 6.2), they must first select a machine tool to perform this manufacturing process.

When searching for different machines, the machine search and selection dialog box (see Fig. 6.3) forces users to only click on different machines to review the information relevant to this machine. The “search machines” button and all of the machine data fields are disabled to prevent users from accidental operations. If users want to search for particular machines, they can press the “edit” button. The “search machine” button and all of the machine search fields then become active, as shown in Fig. 6.4.



DMAC Configuration Panel

The machine tools list from machine database

- DMACXYZ
- TarusXYZCA
- SuginoXYZ
- SuginoXYZAC
- CMM

Selected Mechanism Name: DMACXYZ

Mechanism Device Driver: DMACXYZ.dll

Mechanism Type: Mill

Mechanism Configuration Type: Three-axis

Mechanism Number of Axes: 3

Mechanism Working Volume: 100X100X100

Spindle Max Horse Power: 1000

Spindle Max Speed: 1000

Spindle Max Torque: 50

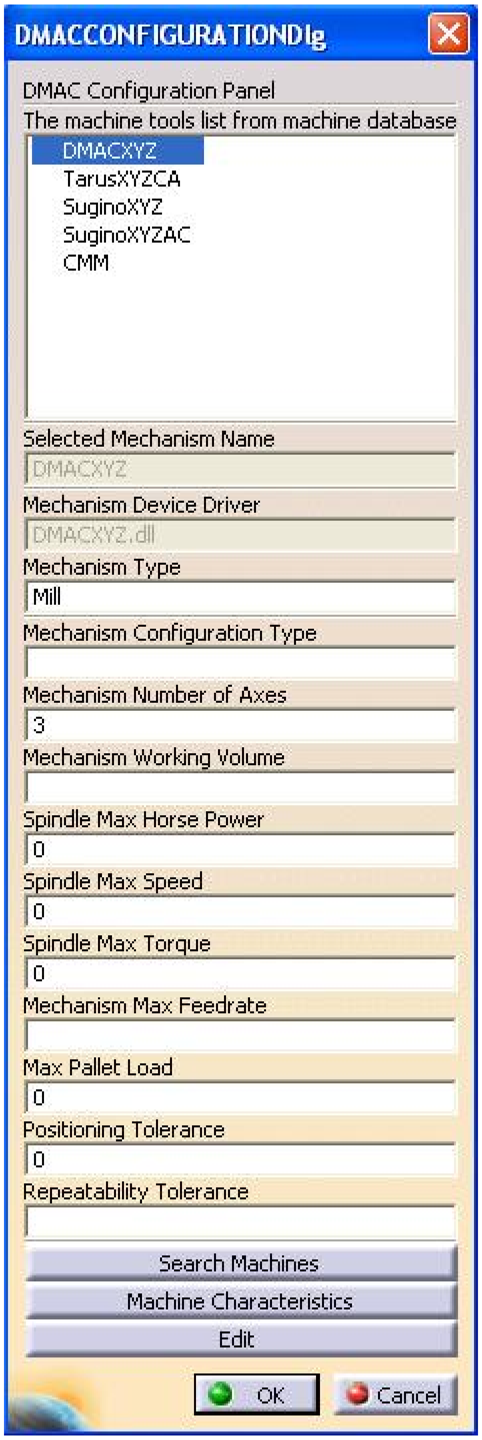
Mechanism Max Feedrate:

Max Pallet Load: 100

Positioning Tolerance: 0.0051

Repeatability Tolerance: 0.0025

Fig. 6.3 Machine search and selection dialog box


DMACCONFIGURATIONDlg

DMAC Configuration Panel

The machine tools list from machine database

DMACXYZ
TarusXYZCA
SuginoXYZ
SuginoXYZAC
CMM

Selected Mechanism Name: DMACXYZ

Mechanism Device Driver: DMACXYZ.dll

Mechanism Type: Mill

Mechanism Configuration Type:

Mechanism Number of Axes: 3

Mechanism Working Volume:

Spindle Max Horse Power: 0

Spindle Max Speed: 0

Spindle Max Torque: 0

Mechanism Max Feedrate:

Max Pallet Load: 0

Positioning Tolerance: 0

Repeatability Tolerance:

Fig. 6.4 Search machines enabled

In this example, users are searching for three-axis milling machines. Fig. 6.5 displays a list of selected machine tools. If users want more details about a particular machine, they can pick one machine and click the "machine characteristics" button (see Fig. 6.4 and Fig. 6.5), where a machine characteristics dialog box will pop up, as shown in Fig. 4.4.

Once users choose a proper machine to perform the manufacturing process, they can then use the "jog" button and the "direct machine" button to operate the selected machine to create a physical part.

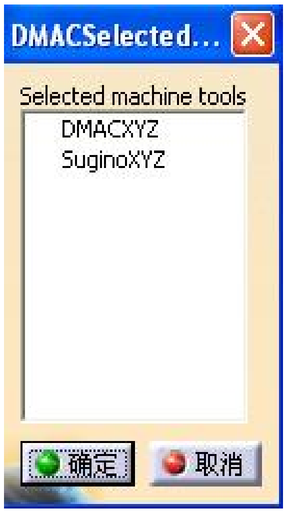


Fig. 6.5 Selected machines

6.2.3 Jog dialog box

Because the three-axis tabletop mill does not have a teach pendent to allow users to manually jog the mill, a jog dialog box (see Fig. 6.6) must be created in CATIA to allow users to position the cutter prior to machining. This dialog box enables the positioning of the cutter at the desired location prior to actual machining. The cutter's initial location must be positioned at the same location where X, Y, and Z are set to zero in the frame used by the CATIA manufacturing process plan. The dialog box created to jog the cutter has the capability of jogging the X, Y, and Z axes.

The slider provided in this dialog box is used to set the distance (in mm) that the cutter tool will be jogged. If the slider has been set to 5 and the "JogXPlus" button is pressed once, the cutter tool will be jogged 5 mm in the positive X direction. If the "ToReferencePoint" button is pressed, the cutter tool will be moved back to the zero position in the CATIA reference frame.

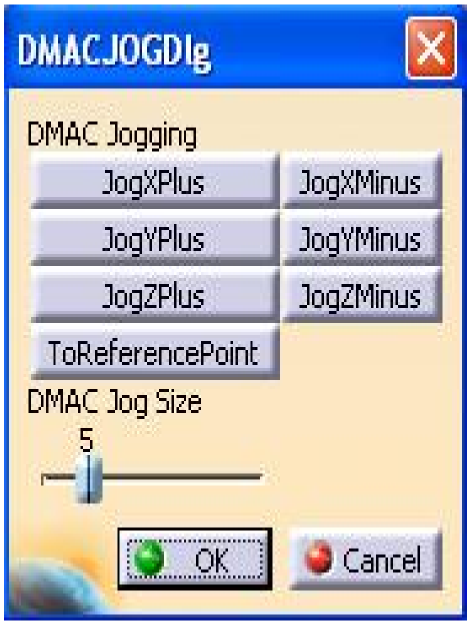


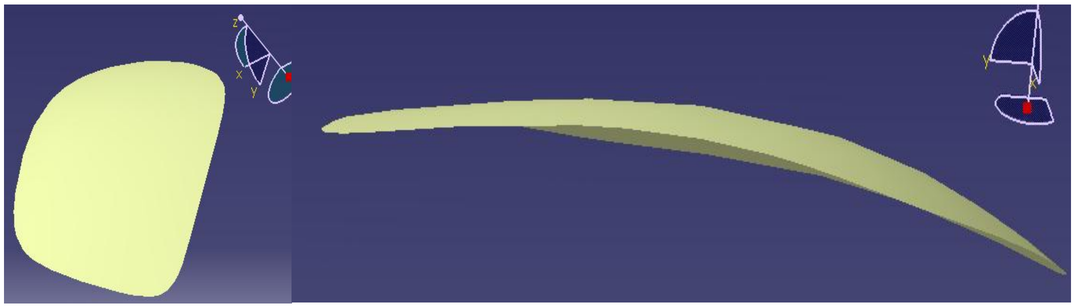
Fig. 6.6 Tool jog dialog box

6.2.4 Experiments

A first experiment was arranged using a scaled 3D CAD data model of a car headlight, similar to data that would typically be used in production at GM (see Fig. 6.7). The headlight was used because it consists of only one free-form surface, but still demonstrates a fair amount of curvature and shape. Tool paths for the surface were generated in CATIA and sent to a three-axis tabletop mill (see Fig. 6.7) that was directly connected to CATIA through a three-axis mill device driver, called DMACXYZ.dll. The computer runs dual Pentium III processors at 1 GHz. The non real-time CATIA application runs on one processor while the real-time applications run on the second processor.

The headlight surface was machined on the three-axis mill by using the three-axis mill driver (DMACXYZ.dll), as shown in Fig. 6.7. The tool paths and the physical part machined directly from CATIA are shown in Fig. 6.8. The same process was also completed by using a conventional Tarus three-axis mill utilized at GM. The processing time comparison between the direct reconfigurable machining process and the traditional M&G method is shown in Table 6-1. As this table shows, it took four steps for the direct reconfigurable machining process to create a physical part from a 3D CAD model. However, it took eight steps and seventeen more minutes to create a part from the same CAD model through the conventional M&G code method.

The resulting decrease in processing time does not come from a reduction in actual machining time, but from a decrease in the time required for tool path post-processing and file handling. The direct reconfigurable machining method eliminates unnecessary intermediate files, generated for the conventional controllers, and unnecessary process steps used on the conventional machining method. Therefore, it greatly simplifies the traditional design-to-manufacturing processes.



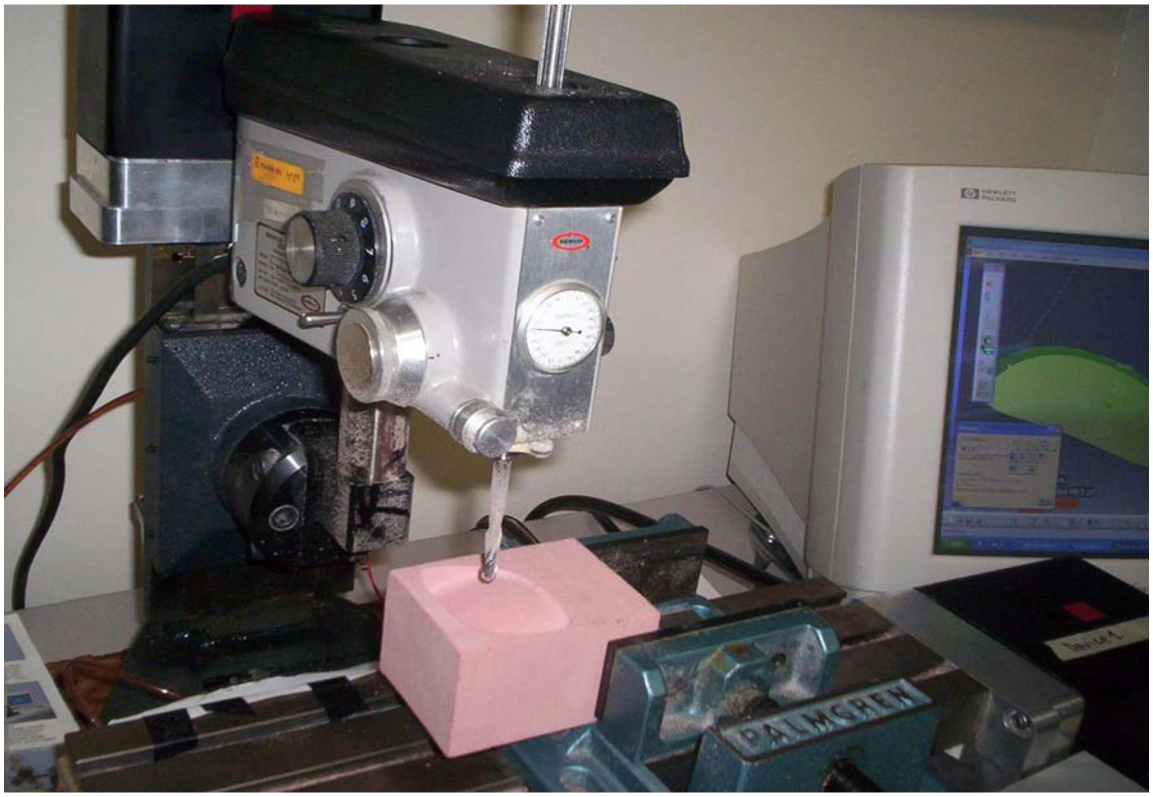
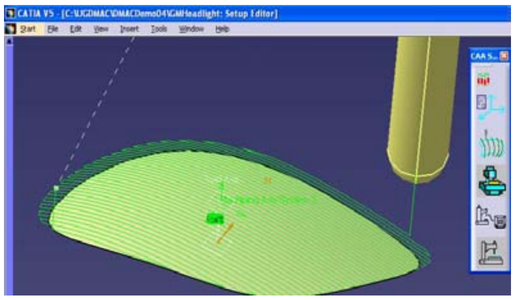


Fig. 6.7 Direct machining a car headlight on the three-axis mill


CATIA V5 - [C:\UGDMAC\DMACDemo04\GMHeadlight: Setup Editor]
Start File Edit View Insert Tools Window Help

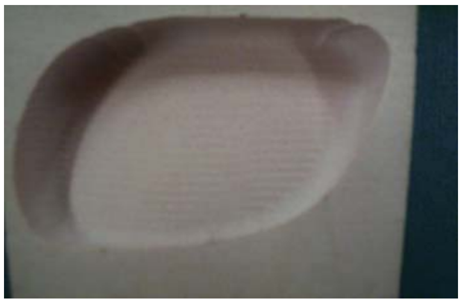
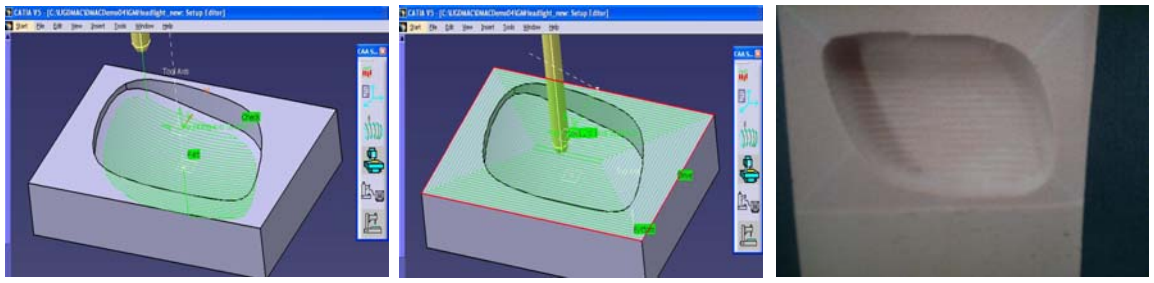


Fig. 6.8 GM headlight surface with process plan and the machined part

Table 6-1 Direct reconfigurable machining process vs. conventional process

Method	Direct reconfigurable machining		Conventional Machining	
	Description	Time	Description	Time
Steps	Creates Part	equivalent	CAD designer Creates Part	Equivalent
			Sends to tool path planner	5 min.
	Searches and selects a proper machine	equivalent	Tool path planner Searches and selects a proper machine	equivalent
	Generates Tool Paths	equivalent	Generates Tool Paths	equivalent
			Post-processes to M&G codes	2 min.
			Sends M&G files to machine operator	5 min.
			Machine operator loads M&G file on CNC machine	5 min.
	Runs program	equivalent	Runs program	equivalent
Totals	4 steps	0 min.	8 steps	17 min.

A second experiment was conducted on the three-axis prototype mill. Two process plans were created for a GM headlight surface as shown in Fig. 6.9. The first process plan was a sweeping operation (see Fig. 6.9.a) intended for the three-axis mill configuration. To execute this process plan directly on the three-axis mill, a device driver called DMACXYZ.dll was first loaded. After the headlight surface was made, a new device driver called DMACXY.dll was then loaded. This driver commanded the prototype mill as a two-axis milling machine. The face milling process plan (see Fig. 6.9.b) was then sent to the prototype mill for direct machining. Fig. 6.9.c shows the machined part after these two operations.



(a) **(b)** **(c)**

Fig. 6.9 GM headlight surface with two process plans and the machined part

As these two experiments demonstrate, the RMAC control system streamlines the design-to-manufacturing processes by eliminating unnecessary intermediate process steps used in the traditional machining method. Meanwhile, it brings tremendous flexibilities into the manufacturing systems. A single machine can be reconfigured to operate differently to fulfill specific manufacturing operation requirements. This is a great advantage over the traditional M&G machining method.

CHAPTER 7 CONCLUSIONS AND RECOMMENDATIONS

This chapter presents the research conclusions and gives recommendations for future work.

7.1 Conclusions

Emerging reconfigurable manufacturing systems require reconfigurable control systems. With the technology of controller software design and the traditional machine tool control in practice today, it is extremely difficult to develop a reconfigurable control architecture that is completely open for easy reconfiguration of the controller software.

Based on the Direct Machining And Control research at Brigham Young University, this dissertation describes software architecture for a dynamically reconfigurable machine tool controller. Because this RMAC controller is completely software-based and is independent of control hardware that is both proprietary and closed to end users, it is possible to develop a driver-like paradigm for a reconfigurable control system.

This dissertation proposed and developed a new control architecture that will allow mapping of each mechanism's configuration and capability into a device driver and use this device driver to reconfigure a RMAC open-architecture controller. This provides CAD/CAM users with greater flexibilities to fulfill different manufacturing operations.

Under the RMAC paradigm, CAD/CAM users can search for an optimal machine tool based on the needs of the current manufacturing process. A mechanism device driver will then be automatically loaded to connect the selected machine tool directly to a CAD/CAM application. With the establishment of this direct link, the exact surface geometry, in its native mathematical format, can be passed to the device driver. The device driver interprets the parametric tool paths and the manufacturing process instructions contained within the process plan. It then properly reconfigures the RMAC open-architecture controller necessary for controlling the selected machine tool. As a result, CAD/CAM organizations no longer need to develop specific post-processing software to interface with each individual machine tool. Instead, a standardized device driver interface is developed to connect various machine tools directly to any CAD/CAM application.

Moreover, with the development of the device driver architecture and the control software reconfiguration methods, it is possible to develop new machine tools with multiple functions. Traditionally, machine tools have only had one behavior, but this research presents the new and unique ability to reconfigure a machine tool controller instantaneously. By unloading and loading a mechanism device driver, the mill is instantly commanded as a material removal machine, and, seconds later, through the loading of a CMM driver, the machine tool can be controlled via an inspection program like PC-DMIS. Experiments with UG simulation on a three-axis and five-axis mill and a machine device driver implemented on the three-axis tabletop mill demonstrate that this is feasible.

While this research project is not yet complete, it already demonstrates promising and exciting capabilities. The development of this reconfigurable machine tool controller architecture around the device driver paradigm could have a major impact on the manufacturing organizations that still rely on the traditional M&G machining methods. Meanwhile, it will bring some significant advantages to the machine tool industry.

First, it shifts much of the programming burden from the manufacturing process software to the controlling software. Under this new paradigm, the tool paths entities, in their original mathematical descriptions, are directly sent down to the controller software which interpolates those tool paths on the controller side. Because the mathematic representation of tool paths is independent of any machine tool controller, and because the machine-dependent M&G codes are completely eliminated, under this new direct control scheme, the standardized device driver interface allows different CAD/CAM software to directly access the internal device driver functions, thus making it possible for the same mechanism controller to communicate directly with any CAD/CAM system. It is no longer necessary for CAD/CAM organizations to spend their valuable resources on developing machine-specific, post-processing interface software. The interface connection from machine control applications to CAD/CAM applications becomes much simpler and easier.

Second, the traditional design-to-manufacturing processes will be greatly streamlined and simplified. The traditional process of transforming the tool paths into some intermediate CL and APT files and post-processing them into the M&G codes is completely eliminated. Instead, a new way of directly controlling a machine by a CAD/CAM application is truly realized. Under this new control paradigm, the original CAD description can now be sent directly to the machine tool through a device driver interface and the modifications made on the shop floor become feedback to the CAD designer to update the original CAD models through the same interface.

Third, the separation of device driver architecture enables control vendors to work independently on various parts of the control system. Each mechanism device driver developer can independently design, debug, and link the mechanism device driver into an executable DLL. The independent and modular nature of device driver software allows endless combinations of third party hardware and control algorithms. Thus, controller vendors can now develop more cost effective, more efficient, and more advanced controllers.

Last but not the least, the modular and independent nature of the device driver architecture offers an extensible software foundation for integrating any new control algorithm. The development of the standardized interfaces between CAD/CAM software, the device driver software, and the controller software, allows the CAD/CAM organizations, the device driver vendors, and the controller vendors to work independently and consistently, integrating new functionalities into their existing products. As a result, end users experience greater benefits by easily upgrading their existing machine tools with any new functionality developed by those different vendors.

7.2 Recommendations

Even though this dissertation makes several original contributions to the field of machine control, these contributions are just the beginning of the work that is required to successfully integrate this reconfigurable control architecture into industry. Some improvements still need to be made to the architecture as this research project moves forward. The following section will provide recommendations for future research.

Firstly, the proposed architecture was only successfully implemented on a three-axis tabletop mill. As the development effort continues, some incremental extensions and refinements need to be made into the existing control architecture and the set of device driver interface APIs.

Secondly, the current set of device driver interface APIs are designed to only support milling and CMM machines. Future efforts must extend the current set of interface functions to support different types of machines, such as robots, drilling machines, and lathes. In addition, more interface functions need to be developed to allow the future reconfigurable control system to be applied on various manufacturing applications, such as welding, drilling, and parts assembly.

Third, an ideal CAD/CAM system should provide *true* simulation of the manufacturing process it generates. In other words, a CAD/CAM system should provide a simulation engine that contains a set of machine tools on which the program simulates the manufacturing process—on the computer monitor—with no difference from the actual machine tool. The current simulation is limited to only allowing animation of the machine tool motion. Future developments must create the capability of simulating the machine dynamics so that CAD/CAM users can obtain the actual machining force and torque information prior to physically operating a machine. This will better assist CAD/CAM users to evaluate and select an optimal machine tool.

Lastly, and most important of all, significant efforts need to be made to ensure collaboration between DMAC and the major CAD/CAM and controller vendors. The ultimate goal of this research project is to successfully integrate the RMAC architecture into industry. The support from these major CAD/CAM and controller vendors is necessary for the success of this research.

BIBLIOGRAPHY

.

[1] Y. Koren, U. Heisel, F. Jovane, T. Moriwaki, G. Pritschow, G. Ulsoy, H.V.Brussel., Reconfigurable Manufacturing System, Annals of the CIRP vol. 48/2/1999, pp. 527-540.

[2] Y. M. Moon, S. Kota., Synthesis of Reconfigurable Machine Tools with Screw Theory, Proceedings of DETC'00, ASME 2000 Design Engineering Technical Conferences and Computers and Information in Engineering Conference, Baltimore, Maryland, September 10-13, 2000, pp. 157-166.

[3] S. Fujita, T. Yoshida, OSE: Open System Environment for Controller, 7th International Machine Tool Engineers Conference, Nov. 1996, pp. 234-243.

[4] OMAC Users Group, http://www.arcweb.com/omac.

[5] OSACA Organization, http://www.osaca.org.

[6] P. Lutz and W. Sperling, OSACA – The Vendor Neutral Control Architecture, Proceedings of the European Conference on Integration in Manufacturing IiM'97, Dresen, Germany, 1997.

[7] G. Pritschow, P. Lutz, Open System Controllers – a Challenge for the Future of the Machine Tool Industry, Annals of the CIRP vol. 42/1/1993, pp. 449-452.

[8] P. Wright, S. Schofield, Open-architecture Controllers for Machine Tools, Part 1: Design Principles, Journal of Manufacturing Science and Engineering, vol. 120, May 1998, pp. 417-424

[9] P. Wright, F. C. Wang, Open Architecture Controllers for Machine Tools, Part 2: A Real-timeQuintic Spline Interpolator, Journal of Manufacturing Science and Engineering, vol. 120, May 1998, pp. 425-432

[10] Y. Koren, Open-Architecture Controllers for Manufacturing Systems, *Open Architecture Control Systems*, ITIA Series, 1998.

[11] R. G. Landers, B. K. Min, Y. Koren, Reconfigurable Machine Tools, Annals of the CIRP, vol. 50/1/2002, pp. 269-275.

[12] K. D. Oldknow, I. Yellowley, Design, Implementation, and Validation of a System for the Dynamic Reconfiguration of Open Architecture Machine Tool Controls, International Journal of Machine Tools & Manufacture, vol. 41, 2001, pp. 795-808.

[13] P. Proctor, The Enhanced Machine Controller, http://www.isd.mel.nist.gov.projects/emc/emc.htm

[14] W. P. Shackleford, F. M. Proctor, Use of Open Source Distribution for a Machine Tool Controller, Proceeding of SPIE, vol. 4191, pp. 19-30.

[15] E. Messina, H. Huang, H. Scott, An Open Architecture Inspection System, Proceedings of the 2000 Japan-USA Flexible Automation Conference, July 23-25, 2000, Ann Arbor, Michigan, pp. 113-118.

[16] D. Chang, A. Spence, S. Bigg, J. Heslip, J. Peterson, An open architecture CMM motion controller, Proceedings of SPIE, vol. 4563 (2001), pp. 1-9.

[17] K. S. Hong, K. H. Choi, J. G. Kim, S. Lee, A PC-based open robot control system : PC-ORC, Robotics and Computer Integrated Manufacturing, vol. 17 (2001), pp. 355-365.

[18] W.S. Atkins, Strategic Study on the EU Machine Tool Sector, Management Consultants, 1990.

[19] M. Zatarain, E. Lejardi, and F. Egana, Modular Synthesis of Machine Tools, Annals of the CIRP, Vol. 37/1, 1998, pp. 333-336.

[20] http://www.sfb467.uni-stuttgart.de/objectives/index.html

[21] S. Birla, Software Modeling for Reconfigurable Machine Tool Controller, Ph.D. Dissertation, The University of Michigan, 1997.

[22] S. Wang, K.G. Shin, Constructing Reconfigurable Software for Machine Control Systems, IEEE Transactions on Robotics and Automation, vol. 18, 2002, pp. 475-486.

[23] N. Erol, Y. Altintas, M. Ito, Open System Architecture Modular Tool Kit for Motion and Machining Process Control, IEEE/ASME Transactions on Mechatronics, vol. 5, No. 3, 2000, pp. 281-191.

[24] S. Kolla, J. Mlchaloski, W. Rippy, Evaluation of Component-Based Reconfigurable Machine Controllers, Proceedings of the World Automation Congress (WAC) 2002, Orlando, FL, June 9-13, 2002, pp. 625-630.

[25] F. Proctor, J. Michaloski, S. Birla, and G. Weinert, Analysis of Behavioral Requirements for Component-Based Machine Controllers, Proceedings of SPIE, vol. 4191 (2001), pp. 10-18.

[26] D. Stewart, R.A. Volpe, P.K. Khosla, Design of Dynamically Reconfigurable Real-Time Software Using Port-Based Objects, IEEE Transactions on Software Engineering, vol. 23 (1997), pp. 759-776.

[27] D. Kalita, P.P. Khargonekar, Formal Verification for Analysis and Design of Logic Controllers for Reconfigurable Machining Systems, IEEE Transactions on Robotics and Automation, vol. 18 (2002), pp. 463-474.

[28] Y. Zhang, K. D. Roufas, M. Yim, Software Architecture for Modular Self-Reconfigurable Robots, Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems, Maui, Hawaii, USA, 2001, pp. 2355-2360.

[29] I.M. Chen, Rapid response manufacturing through a rapidly reconfigurable robotic workcell, Robotics and Computer Integrated Manufacturing, vol. 17 (2001), pp. 199-213.

[30] K. Feldmann, M. Wenk, Flexible Sensor Control Solutions for Robot Controls by Reconfigurable Software Architectures, Proceedings of the 2000 Japan-USA Flexible Automation Conference, 2000, pp. 847-851.

[31] W.J. Schonlau, MMS: A Modular Robotic System and Model Based Control Architecture, Proceeding of the SPIE, vol. 3839 (1999), pp. 289-296.

[32] http://www.step-nc.org/

[33] S. Suh, B. Chung, B. Lee, J. Cho, S. Cheon, H. Hong, H. Lee, Developing an Integrated STEP-Compliant CNC Prototype, Journal of Manufacturing Systems 2002, vol, pp. 350-362.

[34] Suh, Suk-Hwan; Cho, Jung-Hoon; Hong, Hee-Dong, On the architecture of intelligent STEP-complaint CNC, Int. Journal of CIM, V.15/2, 2002, pp. 168-177.

[35] OMAC Users Group, The Value Proposition for STEP-NC, version 4, May 2002.

[36] STEP-NC consortium, ESPRIT Project EP 29708, STEP-NC Final Report, version 1, Nov. 2001.

[37] S.T. Newman, R.D. Allen, and R.S.U. Rosso, CAD/CAM solutions for STEP complaint CNC manufacturer, Proceedings of the 1st CIRP(UK) Seminar on Digital Enterprise Technology, 2002, pp. 123-128.

[38] R.S.U. Rosso Jr, R.D. Allen, S.T. Newman, Future Issues for CAD/CAM and Intelligent CNC Manufacturer, Proceedings of the 19th International Manufacturing Conference, 2002.

[39] J. Bosley, A Modular, Open-Architecture Controller for Direct Machining, M.S. Thesis, Brigham Young University, Provo, Utah, 2000.

[40] C. L. McBride, An Open Architecture Controller for Direct Machining and Control, M.S. Thesis, Brigham Young University, Provo, Utah, 2002.

[41] H. Miza, DEVELOPING A DIRECT MACHINING INTERFACE FOR COMMERCIAL CAM SYSTEMS, M.S. Thesis, Brigham Young University, Provo, Utah, 2003.

[42] C.P. Bassett, C.G. Jensen, W.E. Red, M.S. Evans, Direct Machining: a New Paradigm for Machining Data Transfer, Proceedings of DETC2000/DFM-14298:

ASME 5th Design for Manufacturing Conference, Baltimore, Maryland, September 10-13, 2000.

[43] C.P. Bassett, C.G. Jensen, J.E. Bosley, Y. Luo, W.E. Red, M.S. Evans, Direct Machining Architectures Using CAD-CAM Generative Methods, Proceedings of the IASTED International Conference on Control and Applications 2000, pp. 287-295.

[44] W.E. Red, M.S. Evans, C.G. Jensen, J.E. Bosley, Y. Luo, Architecture for Motion Planning and Trajectory Control of a Direct Machining Application, Proceedings of the IASTED International Conference on Control and Applications 2000, pp. 484-489.

[45] M.S. Evans, W.E. Red, C.G. Jensen, C. L. McBride, G. Ghimire. Open Architecture for Servo Control using a Digital Control Interface, Proceedings of the IASTED International Conference on Control and Applications 2000, pp. 339-344.

[46] Frank Wei Li, T. Davis, C.G. Jensen, and W.E. Red, Rapid and Flexible Prototyping through Direct Machining, Computer-Aided Design and Applications, Vol. 1, Nos. 1-4, CAD'04, 2004, pp. 91-100.

[47] Wei Li, W.E. Red, C.G. Jensen, and T. Davis, "Dynamic Reconfigurable Machine Tool Controller", Proceedings of IMECE04, IMECE-61317, November 13–19, 2004, Anaheim, California, USA.

[48] Z.Y. Chen, W. Li, R. Cheatham, J.G. Wang, C.G. Jensen, and W.E. Red, "A DIRECT MACHINING SYSTEM FOR COMMERCIAL CAD/CAM PACKAGES", Proceedings of IMECE04, IMECE-59992, November 13–19, 2004, Anaheim, California USA.

[49] W.E. Red. A dynamic optimal trajectory generator for Cartesian Path following. Robotica 2000;18:451-458.

[50] J.J.Craig, Introduction to Robotics: Mechanics and Control (Addison-Wesley, 2nd edition, 1989).

[51] G.F. Franklin, J.D. Powell, A.E. Naeini, Feedback Control of Dynamic Systems (Addison-Wesley, 3rd edition, 1994).

[52] G.F. Franklin, J.D. Powell, M. Workman, Digital Control of Dynamic Systems (Addison-Wesley, 3rd edition, 1997).

[53] G. Ghimire, Specification and Application of a Digital Control Interface, M.S. Thesis, Brigham Young University, Provo, Utah, 2000.

[54] Walter Oney, Programming the Windows Driver Model (Microsoft Press, 1999).

[55] Microsoft, Windows 2000 Driver Design Guide (Microsoft Press, 2000).

[56] M. Klein, Windows Programmer's Guide To DLLs and Memory Management (SAMS, 1st edition, 1992).

APPENDICES

APPENDIX I

Device_Driver_Object and Device_Object Specification

Specification version 1.0

This appendix contains the definition of the *Device_Driver_Object* and *Device_Object* data structure, as described in section 3.4.2.3 and 3.4.3.2. These two data structures are declared in the device driver manager's header file.

```
/**************************************************/
/******Data structure for the DEVICE_DRIVER_OBJECT*****/
/**************************************************/
typedef struct _DEVICE_DRIVER_OBJECT{

    int         MechanismID;
    char        MechanismName[100];
    char        DeviceDriver[100];
    char        MechanismType[100];
    char        MechanismConfigurationType[100];
    int         MechanismNumOfJoints;
    char        MechanismWorkingVolume[100];
    double      MechanismSpindleHorsePower;
    double      MechanismSpindleMaxSpeed;
    double      MechanismSpindleMaxTorque;
    double      MechanismMaxFeedrate;
    double      MechanismMaxPalletLoad;
    double      MechanismPositioningTolerance;
    double      MechanismRepeatabilityTolerance;
    char        MechanismSpeedCapability[100];
    char        DeviceDriverVersion[100];

}DEVICE_DRIVER_OBJECT, *PDEVICE_DRIVER_OBJECT;


/**************************************************/
/**********Data structure for the DEVICE_OBJECT********/
/**************************************************/
typedef struct _DEVICE_OBJECT{

    char        MechanismName[100];
    char        MechanismKinematics[100];
    int         MechanismNumOfJoints;
    char        MechanismConfiguration[100];
    char        MechanismActuatorMap[100];
    char        MechanismWorkingVolume[100];
    double      MechanismSpindleHorsePower;
    double      MechanismSpindleMaxSpeed;
    double          MechanismSpindleMaxTorque;
    double          MechanismMaxFeedrate;
    double          MechanismMaxPalletLoad;
    double          MechanismPositioningTolerance;
    double          MechanismRepeatabilityTolerance;
    int             MechanismJntTypes[DMAC_MAX_JOINTS];
    double          MechanismJntMinLimit[DMAC_MAX_JOINTS];
    double          MechanismJntMaxLimit[DMAC_MAX_JOINTS];
    double          MechanismJntMaxSpeed[DMAC_MAX_JOINTS];
    double          MechanismJntMaxAccel[DMAC_MAX_JOINTS];
    double          MechanismJntMaxJerk[DMAC_MAX_JOINTS];
    char            MechanismJntServoControlLaw[DMAC_MAX_JOINTS][100];
    char            MechanismSpindleServoControlLaw[100];

}DEVICE_OBJECT, *PDEVICE_OBJECT;
```

APPENDIX II

Device Driver Manager’s Interface APIs Specification

Specification version 1.0

This appendix contains the definition of the device driver manager's interface APIs, as described in section 3.4.2.4. These interface APIs are declared in the device driver manager's header file.

```
/**************************************************/
/****** Microsoft access database functions (Using ODBC) ****/
/*************************************************/

BOOL OpenDeviceDriverDatabase()
            Returns true if the device driver database is opened, otherwise, returns
            false.

BOOL CloseDeviceDriverDatabase()
            Returns true if the device driver database is closed, otherwise, returns
            false.

int     GetTotalNumberOfRecords()
            Returns the total number of machine records contained within the device
            driver database.

int     SearchDatabaseRecordSet(char*     pMachineSQLStatement)
            Returns the total number of searched machines using the given SQL
            statement. If none record is found, return zero.

BOOL GetDatabaseRecordSetColumns()
            Returns true if are columns are obtained from the device driver database,
            otherwise, returns false.

VOID  OnNextRecord()
            Move to the next database record.

VOID  OnPreviousRecord()
            Move to the previous database record.

VOID  OnFirstRecord()
            Move to the first database record.

VOID  OnLastRecord()
            Move to the last database record.
```

```
/**************************************************/
/************* Mechanism device related functions **********/
/*************************************************/


int       Machine_GetMechanismID()
                  Returns the mechanism ID defined within the device driver database.

Char*   Machine_GetMechanismName()
                  Returns the mechanism name defined within the device driver database.

Char*   Machine_GetMechanismDeviceDriver()
                  Returns the mechanism device driver name defined within the device
                  driver database.

int       Machine_GetMechanismDeviceDriverVersion()
                  Returns the mechanism device driver version defined within the device
                  driver database.

Char*   Machine_GetMechanismType()
                  Returns the mechanism type defined within the device driver database.

Char*   Machine_GetMechanismConfigurationType()
                  Returns the mechanism configuration type defined within the device
                  driver database.

int       Machine_GetMechanismNumberOfJoints()
                  Returns the mechanism's number of joints defined within the device driver
                  database.

Char*   Machine_GetMechanismWorkingVolume()
                  Returns the mechanism working volume defined within the device driver
                  database.

double   Machine_GetMechanismSpindleHorsePower()
                  Returns the mechanism spindle horse power defined within the device
                  driver database.

double   Machine_GetMechanismSpindleMaxSpeed()
                  Returns the mechanism spindle maximum speed defined within the device
                  driver database.
```

double   Machine_GetMechanismSpindleMaxTorque()
Returns the mechanism spindle maximum torque defined within the device driver database.

double Machine_GetMechanismMaxFeedrate()
Returns the mechanism maximum feedrate defined within the device driver database.

double   Machine_GetMechanismMaxPalletLoad()
Returns the mechanism maximum pallet load defined within the device driver database.

double   Machine_GetMechanismPositioningTolerance()
Returns the mechanism position tolerance defined within the device driver database.

double   Machine_GetMechanismRepeatabilityTolerance()
Returns the mechanism repeatability tolerance defined within the device driver database.

APPENDIX III

Device Driver’s Interface APIs Specification

Specification version 1.0

This appendix contains the definition of the device driver's interface APIs, as described in section 3.4.3.3 and 3.4.3.4. These interface APIs are declared in each device driver's header file. The excerpt in this appendix is from a three-axis mill device driver's header file-DMACXYZ.h.

```
/**************************************************/
/****** Microsoft access database functions (Using ODBC) ****/
/**************************************************/

BOOL OpenDeviceDatabase()
            Returns true if the device database is opened, otherwise, returns false.

BOOL CloseDeviceDatabase()
            Returns true if the device database is closed, otherwise, returns false.

BOOL GetDatabaseRecordSetMechanismColumns()
            Returns true if all mechanism columns are obtained from the device
            database, otherwise, returns false.

BOOL GetDatabaseRecordSetJointColumns(int JntNum)
            Returns true if the specified joint columns are obtained from the device
            database, otherwise, returns false.


/**************************************************/
/************* Mechanism related functions ***************/
/**************************************************/

Char*   Machine_GetMechanismName()
            Returns the mechanism name defined within the device database.

Char*   Machine_GetMechanismKinematics()
            Returns the mechanism kinematics defined within the device database.

Char*   Machine_GetMechanismActuatorMap()
            Returns the mechanism actuator mapping object defined within the device
            database.
```

int Machine_GetMechanismNumberOfJoints()
Returns the mechanism's number of joints defined within the device database.

Char* Machine_GetMechanismConfiguration()
Returns the mechanism configuration defined within the device database.

Char* Machine_GetMechanismWorkingVolume()
Returns the mechanism working volume defined within the device database.

double Machine_GetMechanismMaxFeedrate()
Returns the mechanism maximum feedrate defined within the device database.

double Machine_GetMechanismSpindleHorsePower()
Returns the mechanism spindle horse power defined within the device database.

double Machine_GetMechanismSpindleMaxSpeed()
Returns the mechanism spindle maximum speed defined within the device database.

double Machine_GetMechanismSpindleMaxTorque()
Returns the mechanism spindle maximum torque defined within the device database.

double Machine_GetMechanismMaxPalletLoad()
Returns the mechanism maximum pallet load defined within the device database.

double Machine_GetMechanismPositioningTolerance()
Returns the mechanism position tolerance defined within the device database.

double Machine_GetMechanismRepeatabilityTolerance()
Returns the mechanism repeatability tolerance defined within the device database.

```
/***************************************************/
/************* Mechanism joint related functions **********/
/**************************************************/


int       Machine_GetMechanismJntTypes(int JntNum)
              Returns the mechanism joint type, for the specified JntNum, defined
              within the device database.

double    Machine_GetMechanismJntMaxLimit(int JntNum)
              Returns the mechanism maximum joint limit, for the specified JntNum,
              defined within the device database.

double    Machine_GetMechanismJntMinLimit(int JntNum)
              Returns the mechanism minimum joint limit, for the specified JntNum,
              defined within the device database.

double  Machine_GetMechanismJntMaxSpeed(int JntNum)
              Returns the mechanism maximum joint speed, for the specified JntNum,
              defined within the device database.

double  Machine_GetMechanismJntMaxAccel(int JntNum)
              Returns the mechanism maximum joint acceleration, for the specified
              JntNum, defined within the device database.

double  Machine_GetMechanismJntMaxJerk(int JntNum)
              Returns the mechanism maximum joint jerk, for the specified JntNum,
              defined within the device database.

Char*   Machine_GetMechanismJntServoControlLaw(int JntNum)
              Returns the mechanism servo control law, for the specified JntNum,
              defined within the device database.



/***************************************************/
/*************** Configuration functions ***************/
/**************************************************/

VOID  Machine_SetMechanismNumberOfJoints(int NumJnts)
              Sets the mechanism number of joints for the DMAC controller.

VOID  Machine_SetMechanismMaxFeedrate(double MaxFeedrate)
              Sets the mechanism maximum feedrate for the DMAC controller.

VOID  Machine_SetMechanismSpindleMaxSpeed(double MaxSpindleSpeed)
              Sets the mechanism maximum spindle speed for the DMAC controller.
```

VOID Machine_SetMechanismSpindleMaxTorque(double MaxSpindleTorque)
Sets the mechanism maximum spindle torque for the DMAC controller.

VOID Machine_SetMechanismJntTypes(int JntNum, int JntType)
Sets the mechanism joint type for the joint specified by the JntNum.

VOID Machine_SetMechanismJntMaxLimit(int JntNum, double MaxJntLimit)
Sets the mechanism maximum joint limit for the joint specified by the JntNum.

VOID Machine_SetMechanismJntMinLimit(int JntNum, double MinJntLimit)
Sets the mechanism minimum joint limit for the joint specified by the JntNum.

VOID Machine_SetMechanismJntMaxSpeed(int JntNum, double MaxJntSpeed)
Sets the mechanism maximum joint speed for the joint specified by the JntNum.

VOID Machine_SetMechanismJntMaxAccel(int JntNum, double MaxJntAccel)
Sets the mechanism maximum joint acceleration for the joint specified by the JntNum.

VOID Machine_SetMechanismJntMaxJerk(int JntNum, double MaxJntJerk)
Sets the mechanism maximum joint jerk for the joint specified by the JntNum.

VOID Machine_ConfigureMotionPlanner()
Generic function call to instruct the controller software to reconfigure its motion planner. The actual interpretation of this generic function call is done inside each device driver. Returns true if the motion planner is successfully reconfigured. Otherwise, returns false.

VOID Machine_ConfigureServoController()
Generic function call to instruct the controller software to reconfigure its servo controller. The actual interpretation of this generic function call is done inside each device driver. Returns true if the servo controller is successfully reconfigured. Otherwise, returns false.

VOID Machine_ConfigueDigitalControlInterface()
Generic function call to instruct the controller software to configure its digital control interface. The actual interpretation of this generic function call is done inside each device driver. Returns true if the digital control interface is successfully configured. Otherwise, returns false.

APPENDIX IV

RMAC_Config and RMAC_CAM interfaces specification

Specification version 1.0

This appendix contains the definition of the RMAC_Config and RMAC_CAM interface APIs, as described in section 3.4.4 and 3.4.5. The RMAC_Config interface APIs are declared in the RMAC_Config COM interface header file and the RMAC_CAM interface APIs are declared in the RMAC_CAM COM interface header file.

```
/**************************************************/
/*************RMAC_Config interface functions **********/
/**************************************************/

VOID  SetMechanismKinematics(char* pDMACKin)
            Sets the mechanism kinematics. pDMACKin is the kinematics class name
            and it is used by DMAC to map the kinematics object from the
            corresponding DLL.

VOID  SetMechanismActuatorMap(char* pDMACMachineActMap)
            Sets the mechanism actuator map.  pDMACMachineActMap is the joint to
            actuator map class name and it is used by DMAC to map this class from
            the corresponding DLL.

VOID  SetMechanismNumJnts(int NumJnts)
            Sets the mechanism number of joints with the specified NumJnts.

VOID  SetMechanismNumSpindle(int NumSpindle)
            Sets the mechanism number of spindles with the specified NumSpindles.

VOID  SetMechanismJointType(int JntNum, int JointType)
            Sets the mechanism joint type for the specified joint (refereed by the
            JntNum). Suitable values for JointType are:

            DMAC_ROT_JNT
            DMAC_LIN_JNT
            DMAC_SCREW_JNT

VOID  SetMechanismJointLimitType(int JntNum, int JointLimitType)
            Sets the mechanism joint limit type for the specified joint (refereed by the
            JntNum). Suitable values for JointLimitType are:

            DMAC_LIMITED_JNT
            DMAC_INFINITE_JNT
            DMAC_INFINITE_JNT_POS
            DMAC_INFINITE_JNT_NEG
```

VOID SetMechanismJointMaximumLimit(int JntNum, double MaxJntLimit)
Sets the mechanism maximum joint limit for the specified joint (refereed by the JntNum). MaxJntLimt is the value for the maximum joint limit.

VOID SetMechanismJointMinimumLimit(int JntNum, double MinJntLimit)
Sets the mechanism minimum joint limit for the specified joint (refereed by the JntNum). MinJntLimt is the value for the minimum joint limit.

VOID SetMechanismJointMaximumSpeed(int JntNum, double MaxJntSpeed)
Sets the mechanism maximum joint speed for the specified joint (refereed by the JntNum). MaxJntSpeed is the value for the maximum joint speed.

VOID SetMechanismJointMaximumAcceleration(int JntNum, double MaxJntAccel)
Sets the mechanism maximum joint acceleration for the specified joint (refereed by the JntNum). MaxJntAccel is the value for the maximum joint acceleration.

VOID SetMechanismJointMaximumJerk(int JntNum, double MaxJntJerk)
Sets the mechanism maximum joint jerk for the specified joint (refereed by the JntNum). MaxJntJerk is the value for the maximum joint jerk.

VOID SetMechanismSpindleMaxRPM(double MaxSpindleRPM)
Sets the mechanism maximum spindle speed. MaxSpindleRPM is the value for the maximum spindle speed.

VOID SetMechanismMaxFeedrate(double MaxFeedrate)
Sets the mechanism maximum feedrate. MaxFeedrate is the value for the maximum feedrate.

VOID SetMechanismNumActuators(int NumActuators)
Sets the mechanism number of actuators with the specified NumActuators.

VOID SetMechanismJointControlMethod(int JntNum, SHORT DMACServoControlMethod)
Sets the mechanism joint servo control method for the specified joint (refereed by the JntNum). Suitable values for DMACServoControlMethod are:

DMAC_POSITION_CONTROL
DMAC_VELOCITY_CONTROL
DMAC_FORCE_CONTROL

VOID SetMechanismJointServoControlGains(int JntNum, double* pServoControlGains)
Sets the mechanism joint servo control gains for the specified joint (refereed by the JntNum). pServoControlGains is the pointer to an array of servo control gains.

VOID SetMechanismJointServoControlLaw(int JntNum, char* pControlLaw)

Sets the mechanism joint servo control law for the specified joint (refereed by the JntNum). Suitable values for pControlLaw are:

DMAC_PID_CONTROL
DMAC_FEEDFORWARD_CONTROL
DMAC_FEEDFORWARDFEEDBACK_CONTROL
DMAC_FUZZY_CONTROL

VOID SetMechanismSpindleControlMethod(SHORT DMACServoControlMethod)

Sets the mechanism spindle servo control method. Suitable values for DMACServoControlMethod are:

DMAC_POSITION_CONTROL
DMAC_VELOCITY_CONTROL
DMAC_FORCE_CONTROL

VOID SetMechanismSpindleServoControlLaw(char* pControlLaw)

Sets the mechanism spindle servo control law. Suitable values for pControlLaw are:

DMAC_PID_CONTROL
DMAC_FEEDFORWARD_CONTROL
DMAC_FEEDFORWARDFEEDBACK_CONTROL
DMAC_FUZZY_CONTROL

VOID SetMechanismSpindleServoControlGains(double* pServoControlGains)

Sets the mechanism spindle servo control gains. pServoControlGains is the pointer to an array of servo control gains.

VOID SetMechanismDigitalControlInterface(char* pDigitalControlInterface)

Sets the mechanism digital control interface. pDigitalControlInterface is the digital control interface class name and it is used by DMAC to map this class from the corresponding DLL.

BOOL InitializeMechanismDigitalControlInterface()

Returns true successfully initialize mechanism's digital control interface, otherwise, returns false.

VOID SetMechanismDigitalIOInterface(char* pDigitalIOInterface)

Sets the mechanism digital I/O interface.

BOOL InitializeMechanismDigitalIOInterface()

Returns true successfully initialize mechanism's digital I/O interface, otherwise, returns false.

/**/
/************** RMAC_CAM interface functions ***********/
/**/

//Milling operation related functions

BOOL CycleStart(long Mode)

Returns true if successfully starts a process cycle, otherwise, returns false. The cycle start can be set in manual mode or automatic mode.

VOID SetCycleStartStatus(BOOL status)

Sets the cycle start status (either true or false).

BOOL GetCycleStartStatus()

Returns the cycle start status (either true or false).

BOOL TurnCoolantOn()

Returns true if successfully turns on the coolant, otherwise, returns false.

BOOL TurnCoolantOff()

Returns true if successfully turns off the coolant, otherwise, returns false.

BOOL DoToolChange(long ToolNumber)

Returns true if successfully changes to the specified tool (referred by ToolNumber), otherwise, returns false.

VOID SetAnimation(BOOL mode)

Sets the animation mode. Turns on the animation if mode is true, otherwise, turns off the animation.

VOID SetSpindleRPM(long SpindleNumber, double RPM)

Sets the spindle RPM for the specified spindle (referred by SpindleNumber). RPM is the spindle speed.

double GetSpindleRPM(long SpindleNumber)

Gets the spindle RPM from the specified spindle (referred by SpindleNumber).

VOID SetSpindleMaxRPM(long SpindleNumber, double MaxRPM)

Sets the maximum spindle RPM for the specified spindle (referred by SpindleNumber). MaxRPM is the maximum spindle speed.

double GetSpindleMaxRPM(long SpindleNumber)

Gets the maximum spindle RPM from the specified spindle (referred by SpindleNumber).

VOID SetFeedrate(double Feedrate)
Sets the current feedrate. Feedrate is the set value for mechanism feedrate.

double GetFeedrate()
Gets the current feedrate.

VOID SetMaxFeedrate(double MaxFeedrate)
Sets the maximum feedrate. Maxfeedrate is the set value for mechanism maximum feedrate.

double GetMaxFeedrate()
Gets the maximum feedrate.

VOID SetPathAccelRise(double newVal)
Sets the value for path acceleration rise. newVal is the set value for path acceleration rise.

double GetPathAccelRise()
Gets the path acceleration rise value.

VOID SetPathAccelFall(double newVal)
Sets the value for path acceleration fall. newVal is the set value for path acceleration fall.

double GetPathAccelFall()
Gets the path acceleration fall value.

VOID SetJointInterpSpeed(double newVal)
Sets the joint interpolation speed. newVal is the set value for joint interpolation speed.

double GetJointInterpSpeed()
Gets the joint interpolation speed.

VOID SetJointAccelRise(long JointNum, double newVal)
Sets the value for joint acceleration rise for the specified joint (referred by JointNum). newVal is the set value for joint acceleration rise.

double GetJointAccelRise(long JointNum)
Gets the joint acceleration rise value from the specified joint (referred by JointNum).

VOID SetJointAccelFall(long JointNum, double newVal)
Sets the value for joint acceleration fall for the specified joint (referred by JointNum). newVal is the set value for joint acceleration fall.

double GetJointAccelFall(long JointNum)

Gets the joint acceleration fall value from the specified joint (referred by JointNum).

VOID SetPartFrame(TDMACFrame PartFrame)

Sets the part frame. PartFrame is the set part frame.

VOID SetToolInterfaceFrame(TDMACFrame ToolInterfaceFrame)

Sets the tool interface frame. ToolInterfaceFrame is the set tool interface frame.

BOOL MoveAlongNurbsND(long MoveID, TDMACNurbsND NurbsND)

Returns true if the NurbsND path is successfully sent, otherwise, returns false. NurbsND is the instance of the TDMACNurbsND data structure representing the mathematics description a ND Nurbs.

BOOL MoveAlongNurbs(long MoveID, TDMACNurbs Nurbs)

Returns true if the Nurbs path is successfully sent, otherwise, returns false. Nurbs is the instance of the TDMACNurbs data structure representing the mathematics description a Nurbs.

BOOL MoveAlongArc3(long MoveID, TDMACVector ViaPoint, TDMACFrame StartPoint, TDMACFrame EndPoint)

Returns true if the circular path is successfully sent, otherwise, returns false. The circular path is defined by a start point (StartPoint), a via point (ViaPoint), and an end point (EndPoint).

BOOL MoveAlongArc(long MoveID, long RotSign, BOOL Closed, TDMACFrame CenterPoint, TDMACFrame StartPoint, TDMACFrame EndPoint)

Returns true if the circular path is successfully sent, otherwise, returns false. The circular path is defined by a start point (StartPoint), an end point (EndPoint), and a center point (CenterPoint).

BOOL MoveAlongLine(long MoveID, TDMACFrame StartPoint, TDMACFrame EndPoint)

Returns true if the linear path is successfully sent, otherwise, returns false. The linear path is defined by a start point (StartPoint) and an end point (EndPoint).

BOOL MoveToPathTarget(long MoveID, TDMACFrame TargetFrame)

Returns true if the path target move is successfully sent, otherwise, returns false. TargetFrame is the target frame that tool needs to move to.

BOOL MoveToJointTarget(long MoveID, TDMACFrame JointTargetFrame)

Returns true if the joint target move is successfully sent, otherwise, returns false. JointTargetFrame is the target frame that tool needs to move to.

BOOL CycleStop(long Mode)

Returns true if successfully stops a process cycle, otherwise, returns false. The cycle stop can be set in manual mode or automatic mode.

VOID SetCycleStopStatus(BOOL status)

Sets the cycle stop status (either true or false).

BOOL GetCycleStopStatus()

Returns the cycle stop status (either true or false).

//CMM related functions

BOOL CurrentPosition([out, retval] double pVal[5]);

Assigns the machine's current joint values to pVal.

BOOL LastHitPosition([out, retval] double pVal[5]);

Assigns the machine's joint values at the time of the last recorded hit to pVal.

BOOL NumMovesInBuffer([out, retval] long *pVal);

Sets pVal equal to the number of moves currently stored in the DMAC motion buffer.

BOOL ControlMode([out, retval] long *pVal);

Sets pVal equal to the current CMM control mode. Suitable values are:

MODE_MANUAL
MODE_AUTOMATIC_MOVE
MODE_AUTOMATIC_MEASURE
MODE_INACTIVE
MODE_STARTUP
MODE_SHUTDOWN

BOOL ControlMode([in] long newVal);

Sets the current CMM control mode equal to the value of newVal.

BOOL Parameter([in] long ParamNumber, [out, retval] double *pVal);

Sets pVal equal to the value of the control parameter defined by ParamNumber. Suitable values for ParamNumber are:

MACH_PREHIT_DISTANCE
MACH_SEARCH_DISTANCE
MACH_RETRACT_DISTANCE
MACH_SCAN_SPEED

BOOL Parameter([in] long ParamNumber, [in] double newVal);

Sets the value of the control parameter specified by ParamNumber equal to newVal. Suitable values for ParamNumber are listed above.

BOOL StatusMajorCode([out, retval] unsigned int *pVal);

Sets pVal equal to the current value of the StatusMajorCode. Suitable values for StatusMajorCode are:

MACHINE_SUCCESS
MACHINE_ERROR

BOOL StatusMajorCode([in] unsigned int newVal);

Sets the StatusMajorCode equal to newVal.

BOOL StatusMinorCode([out, retval] unsigned int *pVal);

Sets pVal equal to the current value of the StatusMinorCode. Suitable values for StatusMinorCode are:

MACHINE_ERR_UNKNOWN
MACHINE_ERR_EMERGENCY_STOP
MACHINE_ERR_NO_AIR
MACHINE_ERR_TRAVEL_LIMIT
MACHINE_ERR_SPEED_LIMIT
MACHINE_ERR_ACCELERATION_LIMIT
MACHINE_ERR_PROBE_NOT_ARMED
MACHINE_ERR_SCALE
MACHINE_ERR_PART_NOT_FOUND
MACHINE_ERR_UNEXPECTED_HIT
MACHINE_ERR_COMM_TIMEOUT
MACHINE_ERR_RESPONSE_TIMEOUT
MACHINE_ERR_INVALID_PARAMETER
MACHINE_ERR_INVALID_PARAMETER_VALUE
MACHINE_ERR_COMMAND_QUEUE_FULL
MACHINE_ERR_UNSUPPORTED_OPERATION
MACHINE_ERR_OTHER

BOOL StatusMinorCode([in] unsigned int newVal);

Sets the StatusMinorCode equal to vewVal.

BOOL Ok([out, retval] BOOL* pVal);

Sets pVal equal to true if no errors have occurred on the machine, false otherwise.

BOOL NewHit([out, retval] BOOL* pVal);

Sets pVal equal to true if a hit has occurred since the calling program lasted updated position values, false otherwise.

BOOL ProbeActive([out, retval] BOOL* pVal);

Sets pVal equal to true if the probe is powered on and is communicating with the controller successfully.

BOOL Home([out, retval] BOOL* pVal);

Sends all of the joints to their home position.

BOOL StopNow();

Stops a manual measure move, and clears the command buffer. This function should stop the machine as well, if possible.

BOOL MoveToXYZ([in] long MoveID, [in] double x, [in] double y, [in] double z, [in] double Speed, [out, retval] BOOL* pVal);

Moves the machine to the specified X, Y, Z location at the input speed.

BOOL MoveRotaryAxis([in] long MoveID, [in] long JointNum, [in] double JointPose, [in] long Direction, [in] double Speed, [out, retval] BOOL* pVal);

Moves the specified rotary axis to the specified position.

BOOL AutoMeasure([in] long MoveID, [in] double x, [in] double y, [in] double z, [in] double Speed, [out, retval] BOOL* pVal);

Makes the machine move toward the specified point with the specified speed until a hit occurs. If a hit occurs, the machine will back up from the specified point in the opposite direction a distance that is specified by previous calls to the interface.

BOOL MoveInArc([in] long MoveID, [in] double x, [in] double y, [in] double z, [in] double CenterX, [in] double CenterY, [in] double CenterZ, [in] double i, [in] double j, [in] double k, [in] double Speed, [out, retval] BOOL* pVal);

Moves the machine from it current position in an arc defined by the included parameters.

BOOL MoveAllAxis([in] long MoveID, [in] double Position[5], [in] double LinearSpeed, [out, retval] BOOL* pVal);

Performs a joint move to the specified joint positions at the specified speed.

BOOL SendConditionChangedEvent([in] double JointValues[5], [in] long NumJoints, [in] unsigned __int64 Mask, [in] unsigned __int64 Condition, [out, retval] BOOL* pVal);

Simulates a probe hit for use when the machine is offline.

BOOL SetCalibratedPartFrame();

Sets the current part frame such that all joint values read zero at the current position.

BOOL ManualMeasure();

Puts the machine into manual control mode and specifies that a probe hit is expected.

BOOL SpinProbe([in] double RPM, [in] double Seconds, [out, retval] BOOL* pVal);

Used to switch the probe on or off by momentarily spinning the spindle.