January 1988

RFMS Software Reference Manual

Hong Zhang
University of Pennsylvania

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-88-01.

RFMS Software Reference Manual

Abstract

This manual explains the software of the Robot Force and Motion Server (RFMS), a high performance robot control system designed and implemented in the GRASP laboratory. In this system, the robot manipulator is considered a force/motion server to the robot and a user application is treated as a request for the service of the manipulator. The user application is created on one of the Unix/VAX machines in 'C' programming language as a set of function calls. The application is carried out in a multi-processor controller, which consists of Intel single board computers and provides computing power necessary for computationally intensive tasks. The VAX machine and the Intel controller communicate through Ethernet, a local area network, which also allows interaction between the user and sensors. Design principles of the system can be found in Section 2.

Comments

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-88-01.

RFMS SOFTWARE REFERENCE MANUAL

Hong Zhang

MS-CIS-88-01
GRASP LAB 130

Department of Computer and Information Science
School of Engineering and Applied Science
University of Pennsylvania
Philadelphia, PA 19104-6389

January 1988

Acknowledgements: This research was supported in part by National Science Foundation under Grant No. ECS-8411879, NSF-CER grant MCS-8219196, and U.S. Army grants DAA29-84-K-0061, DAA29-84-9-0027.

RFMS SOFTWARE REFERENCE MANUAL

Hong Zhang

Department of Computer and Information Science
The University of Pennsylvania
Philadelphia, PA 19104-6389

December, 1987

CONTENTS

APPENDICES

RFMS SOFTWARE REFERENCE MANUAL

Hong Zhang
Department of Computer and Information Science
The University of Pennsylvania

1. Introduction

This manual explains the software of the Robot Force and Motion Server (RFMS)[1], a high performance robot control system designed and implemented in the GRASP laboratory. In this system, the robot manipulator is considered a force/motion server to the robot and a user application is treated as a request for the service of the manipulator. The user application is created on one of the Unix/VAX machines in 'C' programming language as a set of function calls. The application is carried out in a multi-processor controller, which consists of Intel single board computers and provides computing power necessary for computationally intensive tasks. The VAX machine and the Intel controller communicate through Ethernet, a local area network, which also allows interaction between the user and sensors. Design principles of the system can be found in [2].

The software of the system involves a variety of computers: the user interface is written to be executed on a Unix/VAX machine; the control software is written to be executed on Intel 8086-based single board computers; and the network software is written to be executed on a Unix/VAX machine on one end and Intel processor on the other. The rest of the documentation will be organized according to where the execution of the program is. Section Two will discuss user interface, and for those who intend to only use the system for specific applications, it is adequate to read this section. Section Three will discuss the implementation the Ethernet software. This section is useful only if one would like to make changes to the communication protocols between the user and the Intel controller. Section Four will discuss the software written for the robot controller which consists of Intel single board computers to control the robot manipulator, a PUMA 260 in our case. It is important for one to understand this section if what is provided in the system is insufficient to carry out his applications.

This material is based on work supported by the National Science Foundation under Grant No. ECS-8411879. Any opinions, findings, conclusions, or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the National Science Foundation.

We would like to mention that the system is yet to be finalized, for we have been using it for research and thus need to constantly make changes. Several versions of the system exist among the people who have used and modified the system for their own needs. We will try to be consistent throughout this documentation, though confusion may occur from time to time. The programs are organized by the processor on which they are executed, with one directory per processor and common include files in two separate directories. The following table roughly explains the contents of the directories under */usr/users/hz on robo.cis.upenn.edu* and */usr/users/hz/robo* on *grasp.cis.upenn.edu.*

Directory	*Content*
/VAX	user interface and Ethernet driver on the VAX side
/include	include files for /VAX directory
/186	Ethernet driver on the Intel side
/h	include files for the Intel controller
/super	programs written for the supervisor of the Intel controller
/J*i*	programs on the *i*th joint
/math	programs for the math processor
/sys	library functions for 8086 (I/O, interrupt control, vector operations, etc.)
/c86	cross compiler for 8086, loader, and optimizers
/c186	cross comipler for 80186

Table 1. RFMS Directories

All source files will be *underlined* and all functions will be *italicized.*

2. User Interface

From a user's point of view, the available functions can be classified into three categories: world-model definition, motion record definition, and motion requests. Another category, task synchronization, enables the user to wait until the completion of a sub-task before the next one starts. Although it is not available at this time, it can be easily added. Sensor input is another area yet to be integrated into the system, and all the mechanisms exist. The structure of the program is similar to that of an RCCL

program in spirit, whose underlining principles can be found in [3]. A user requests the service of the robot controller by making function calls from a 'C' function named *pumatask()*.

2.1. Programs of the User Interface

A total of eight programs constitute the user interface of the system. Since the emphasis of the system is not to construct a comprehensive robot programming system, effort made to create the user process is kept at minimum. We have used this part of the system only for testing the robot controller.

A user defines a task by making calls to the system functions. A task defines the world model in terms of the transformations (relationships between coordinate frames of interest) and position equations (definitions of points in the work space to which the manipulator is to move). The fashion in which a move to a position is conducted such as segment time, compliance specification, etc., is defined by a motion record. Upon any call to create one of these, the created data structure is first stored in the corresponding symbol table and then a copy of it is sent to the RFMS through the Ethernet. To initiate an action, a move is called with two parameters: a pointer to the destination position and a pointer to a motion record. Fundamental to the user interface are the three symbol tables storing transformations, position equations, and motion records that have been created. The move requests are not stored in a symbol table because they are not referred to by other variables. This may change, however, once task synchronization is needed for the system has to keep track of the move requests have been issued. Once the application is created and compiled, one can run the application like any other 'C' programs by *a.out*.

The *main.c* allocates memory for static symbol tables for the user process, initializes the communication link between the user process and the RFMS, and then calls *pumatask()* defined in, say, *myapp.c*, by the user, which contains a stream of function calls to the system. After defining an application, the user may call the function *debug()*, which logs data coming from the Intel controller in real-time and store them in six different files, corresponding to six joints of the robot manipulator. The nature of the data is entirely up to the user, but there must be an agreement in what the Intel controller sends and what the user interprets. This function call is optional and has been used as a debugging tool so far. One can expect to log one set of data every four to five sampling periods.

There are currently a number of ways to create a transformation: a transformation with pure translation and no rotation by *gentr_trsl()*, a rotation transformation defined in terms of either Euler angles or roll-pitch-yaw angles by *gentr_eul()* or *gentr_rpy()*. All functions related to transformation creation are defined in *trans.c*.

A position equation is created by a call to *makepst()* in *pst.c*. One must provide a name to the position as a string of characters in the first argument and three constants for the three configurations *lefty*, *up*, and *flipped*, associated with the PUMA 260. Since a position equation may contain a number of transformations on either side, *makepst()* must be able to handle variable number of arguments[4]. The last argument of *makepst()* when defined is declared to as a pointer to a transformation, the same data type as the rest of the arguments that follow it when the actual call is made. Two key words, EQ and TL in the actual call help interpret where left-hand side ends and which transformation is the tool transformation [5].

A motion record specifies how a motion is to be executed. and it contains such attributes as segment time, acceleration time, mode of the motion, and compliance specification. These attributes then become the four input arguments to a call to *makemot()*, which is contained in the program *mot.c*. Both segment time and acceleration time are in seconds, and mode of the motion can be either Cartesian or joint. Compliance uses a bit pattern as in Figure 1 to indicate the physical constraints to the motion

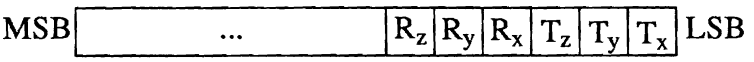


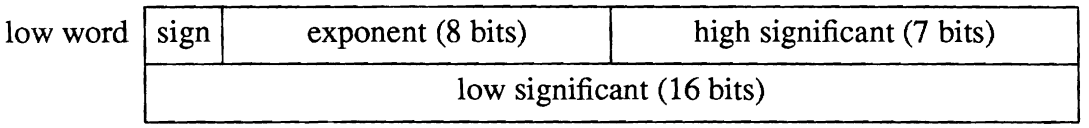
Figure 1. Bit Pattern Representing Compliance

where R_i represents rotational compliance along a certain Cartesian direction and T_i translational compliance along a certain Cartesian direction. In this example, four motion records are defined. The first simply defines a joint motion with a segment time of 2 seconds an acceleration time of 0.2 seconds. The third motion records defines a Cartesian motion with a 20 second segment time, a 0.5 second acceleration time, and compliance along *z* direction.

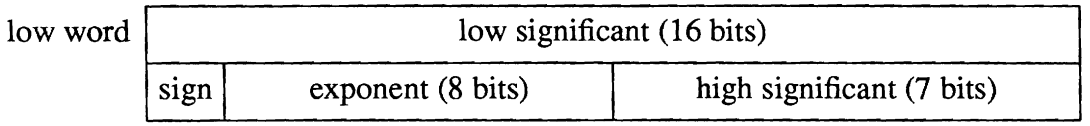
The program *move.c* contains the function *move()*. The function uses the two input arguments, a pointer to position and a pointer to the motion record, to issue a move request.

At the end of each function call, a message is issued to the RFMS. Functions in the file *mess.c* handle packet preparation. Currently, the user application is not receiving any messages, even though the software could handle it. The format of the messages is defined in *msgs.h*. The message type identifies the content and interpretation of the message. A message is written into the buffer, *msg*, before function *mess()* is called, which prepares the Ethernet packet and invokes Ethernet function *Send()* in *comm.c* to send it.

All floating point numbers are modified before being sent, since the VAX machine and Intel computers represent a floating point number differently, as illustrated in Figure 2.



Intel Floating Point Representation



DEC Floating Point Representation

Figure 2. Floating Point Representation

We choose to convert floating point numbers on the VAX machine since it is faster than any Intel computer and time on the Intel computers is more valuable. The function *convert()* in *mess.c* performs the conversion.

2.2. An Example

The following example further illustrates how an application program is created.

```
# include "../include/datdef.h"
# include "../include/extdef.h"
# include "../include/condef.h"

pumatask()
{
        TRSF *t2;
        PST *home_pst;
        MOT *mjnt, *mwait, *mcart, *mcwait, *mcartcz, *mcartcx;

        t2 = gentr_trsl("t2", 203.2, -126.23, 203.2);        /* home */
        home_pst = makepst("home", RIGHT, DOWN, FLIP, t6, EQ, t2, TL, t6);
        mjnt = makemot(2.0, 0.2, JNT, 0);
        mcart = makemot(4.0, 0.3, CAR, 0);
        mcartcz = makemot(20.0, 0.5, CAR, 0x4);
        mcartcx = makemot(15.0, 0.5, CAR, 0x1);

        move(home_pst, mcartcz);
        move(home_pst, mjnt);
```

}

The three include files in the beginning are necessary for the user to define local variables of the data types created for robot programming *(h/datdef.h),* to make function calls to the system *(h/extdef.h),* and to make use of the constants defined in the system *(h/condef.h). TRSF, PST,* and*MOT* represent data type transformation, position equation, and motion request, respectively. In the instruction section of *pumatask()*, a transformation is first created by providing function *gentr_trsl()* with three translational components of the *p* vector in the order of x, y, and z.

The Function call, *makepst()*, creates a position equation with transformations either known to the system or defined by the user. In our case, it has *t6*, which is known to the system, on one side and *t2*, which is defined by the user, on the other. Configurations of this position are specified as right, down and flip. Four motion records are defined in this program, with one joint motion, and three Cartesian motion, of which two require compliance.

Two motions are requested in this task. The arm will move to the same position as the initial position (i.e., remain stationary), while complying along *z* direction. Once this is finished, the arm will move back to home position.

Once the application is created, it can be compiled and linked with the rest of the system. The application is executed in the same fashion as any other Unix executable file, when the Intel controller is initialized and ready to accept tasks.

3. Ethernet Interface

The user and the Intel controller communicate through Ethernet, a local area network. The implementation details of this interface can be found in [6] and in Appendix A. Here we only outline some of its features users need to know in order to use it.

The interface on the users' side is performed on a Unix/VAX machine. Unix supports Ethernet and, for robot control, our software is built as the data link layer by making use of the Data Link Interface (DLI). The interface on Intel's side is built from scratch and has two layers, the data link and logic link. The protocol used between the two machines is *one-bit-sliding window and positive acknowledgement with retransmission*, which means the machine sending a message keeps trying until it receives acknowledgement or the number of trials exceeds a limit. A token exists which determines who can send a message at any given moment. It is usually held by the VAX machine and the Intel machine has it only when the VAX machine requests a message from the Intel controller. Typically the VAX machine sends a message to the Intel machine whenever it wants and the arrival of a message creates an interrupt to the Ethernet board 186/51[14] of the Intel controller, which then reads the message in its interrupt handling procedure. The Intel controller, on the other hand, cannot send a

message to the VAX unless it is explicitly asked to do so. This is caused by the fact that the software on the VAX side is not written as an interrupt handler, but rather as a listener and therefore can not deal with any unexpected incoming messages.

Two primitives on the VAX for sending and receiving a message have the syntax:

Send (buffer, size)

and

Recv(buffer, size).

The counterpart on Intel side employs two primitives:

Recv_Frame(buffer)

and

Send_Ack() or *Ans_Send_Req()*.

Which one to use to send a packet depends upon if the message just received is a real message or a request for a message to be sent to the VAX. Once messages are received by the 186/51, they are queued in an array, waiting to be processed by the supervisor of the Intel controller.

The communication software for VAX is contained in one file *comm.c*, and for the Intel controller there are three 'C' files in the directory */186*, *dld.c*, *llc.c*, and *main.c*. The program *dld.c* contains the data link layer, and the program *llc.c* contains the logic link layer. The program *main.c* first initializes the data link layer by *Init_586()*, sets up a linear array of messages in which the incoming messages are stored, and inform the supervisor of the array address by storing it at a fixed memory location accessible to both supervisor. Two other assembly programs in this directory, *reint.a86* and *handler.a86*, deal with the interrupt control of the 186/51.

There is only limited memory space on the 186/51 and, therefore, the size of the message queue can be of only a finite length. Currently, a total of 100 messages can be stored, of which each has a fixed size of RBUF_SIZE bytes. Since the supervisor keeps looking in the queue for available new entries, overflow never occurs if we assume the speed of processing messages by the supervisor is faster than the that of the incoming messages. The system fails if this assumption is not valid. A dirty bit in the last byte of a message buffer indicates if the buffer contains an unprocessed message.

There are currently two 186/51 computers of different models: one is an ES and the other an S. In additional to their difference in jumper locations and notations, the only software difference one needs to know is the Ethernet address defined for the Ethernet chip 82586. The S model has an address of

0x08, 0x00, 0x2b, 0x02, 0x89, 0xfc,

and the ES model has an address of

0x08, 0x00, 0x2b, 0x02, 0x96, 0x74.

4. Intel Controller

This part of the software runs on Intel single board computers, and it is developed on a VAX machine where the user process is and cross-compiled and down-loaded to the targets via a serial line. (The information on the cross-compiler can be found in Appendix B) The controller is a multi-computer system with shared memory and a common bus, through which data communication and control signals are transmitted. Each computer in the system contains dual-ported memory, of which part is defined as global so that other computers in the system can access it as well. Information exchange takes place in the form of mail boxes and system synchronization is achieved by interrupts. There are currently nine computers running in parallel, six joint processors, a supervisor, a math processor, and an Ethernet computer. There is a real-time synchronized interrupt driven process on each of the joint processors, the supervisor and the math processor. In addition, there is a background process on the supervisor and the math processor. 186/51 runs asynchronously with the rest of the system.

Supervisor, joints and the rest of the system need to communicate with each other and exchange information. Also the kind of data each one requires of any other is known a priori. To facilitate such communication, *mail-box*es are created on each computer with their addresses stored at pre-defined memory locations. These addresses are currently stored in the topmost part of the memory from segment *0xff00* so as not to interfere with the code, data, or stack segments. During the initialization process, supervisor waits until ready flags are cleared in all processors before it picks up addresses of the mail-boxes where it will either drop or pick up mails. Most of the global memory access is done by the supervisor. Currently the only access by the joints is during the compliance when every joint needs to collect other joints' errors. Two system functions, *rblock()* and *wblock()* facilitate global memory access. The sources and destinations of the mail boxes are summerized in the following table.

data type	*source buffer (origin)*	*destination buffer*	*description*
S_MAIL	MAIL (supervisor)	MAIL	one copy to each joint to instruct what actions to take
M_MAIL	MMAIL (supervisor)	MMAIL	information math processor needs to compute Jacobian matrices and dynamics
J_MAIL	JMAIL (joints)	JMAIL*i* *i=1...n*	one from each joint to the supervisor to return the status of the joint
PARCEL	PARi (math)	PARC*i* *i=1...n*	results computed by math and collected by supervisor for one of the joints
PARCEL	PARC*i* *i=1...n* (supervisor)	PARC	one on each joint distributed by the supervisor

Table 2. Mailbox Description

Both trajectory generation and inverse kinematics are performed on this parallel processor and a lot of efforts have been devoted to computation distribution. Trajectory generation at Cartesian level, *i.e.*, calculation of the end effector position and orientation, is performed on the supervisor. Joints, on the other hand, plan their individual trajectories given the end effector coordinates. The dependency exists among the inverse kinematics of the joints, for the *i*th joint requires solutions of all prior i - 1 joints. This dependency, however, can be eliminated when each joint uses other joints' solutions in the previous period. This scheme is approximate, but it allows the system to compute the kinematics in parallel thus speeding up the system substantially. The details of the trajectory trajectory can be found in [7] and the details of the parallel inverse kinematics can be found in [8].

4.1. Supervisor

Two concurrent processes, one being interrupt driven and the other in the background, are executed on the supervisor. The background process reads the messages stored in the 186/51 and sets up data structures, which the second interrupt driven process uses to coordinate the operation of the controller and the generation of motion trajectories. Supervisor runs on an iSBC 86/30 computer[22].

The program, *main.c*, initializes the system and interacts with the user to go through the *manual* mode, the *calibration* mode, and then onto the *set-point* mode. Its serial port is connected to a terminal where the user operates for the purpose of down-loading the code and monitoring the controller operation during system development. Eventually, the interactive session should take place between the VAX machine where the user really is and the control system through the Ethernet.

4.1.1. Background Process

The background process program is stored in *bkgd.c*. To process messages stored on the 186/51 (refer to Section 3), the supervisor maintains a pointer to the next available message in the message queue. Depending upon the type of the message, different action is taken. The format of the messages are defined in the include file *h/msgs.h*. Data structure definitions in this file must agree with those in *include/datdef.h*, if the supervisor is to interpret the messages correctly. When there is no message in the queue, the background process simply waits.

Upon the arrival of a message, the type of a message is determined, and a corresponding data structure may be created and added to the world model. Currently, there are six possible types, INIT, STOP, TTR, TPOS, TMD, TREQ. The first two simply are signals for the beginning and end of a task definition. The rest are for a transformation, a position, a mode, and, motion request message, respectively. The definitions of these data structures can be found in *h/datadef.h*.

These data structures refer to or are linked with each other. For example, a position contains pointers to transformations defined previously. If the messages came from the same machine as the one that receives it, the addresses could be used as pointers. Unfortunately this is not the case. A linked structure must be sent piece by piece and the receiving machine must be able to resolve all the cross references. In order to be able to locate the dependencies, we associate each message of a given type with an identification number. To facilitate a fast search, four symbol talbes, *ttbl*[], *mtbl*[], *ptbl*[], *rtbl*[], are set up to store the pointers to the data structures and the id numbers are indeces in the symbol tables.

When a position equation message arrives, a ring structure is created[5]. The program, *pstn.c*, contains functions necessary to create the structure. A ring consists of a number of *items* representing transformations in the equation, of which each contains a pair of *atoms* containing the forward and inverse transformation. Function *Atom()* allocates memory for one atom, *NewTerm()* links a pair of atoms, *Listn()*'s link *n* terms, and *MakePos()* takes two lists of terms as left and right hand sides of the equation and forms the ring.

Processing of other messages requires much less work and is dealt without any primitive functions.

4.1.2. Real-time Process

The real-time process is executed upon a periodic interrupt signal generated by the programmable timer on the supervisor. The entire process runs like a finite state machine and action taken in each period depends on two state variables. The variable, *rtstate*, in program *rtisr.c*, changes among eight possible states, IDLE, FREE, MANU, CALIB, HOLD, SETP, STOP, and EMGCY. These constants are defined in file *comm.h*. The state the system may fall in is illustrated by the following graph.

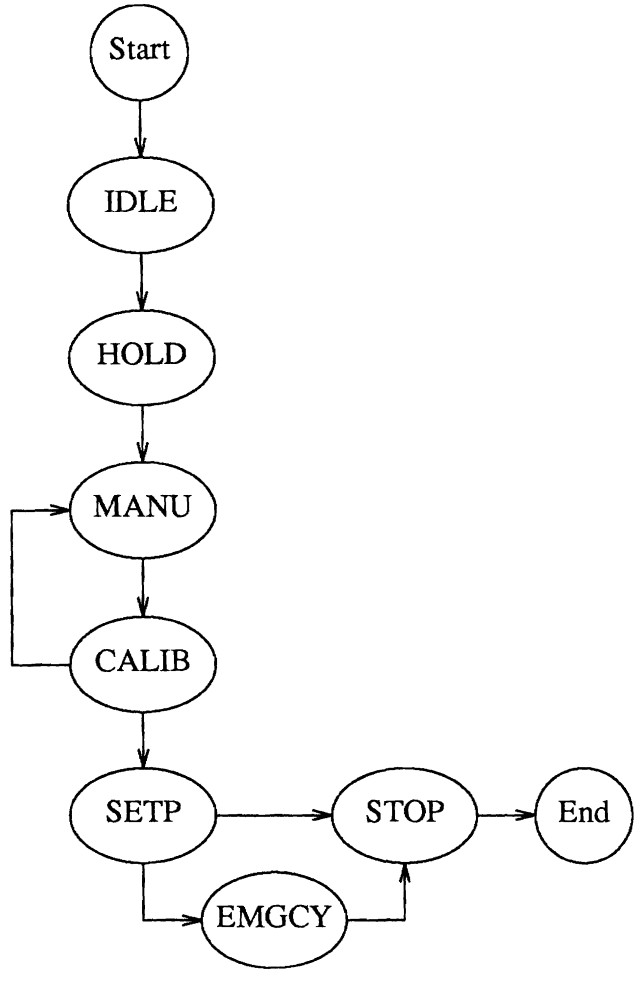


Figure 1: State Diagram of the RFMS

The interpretation of each state is summerized in the following table.

state	*action*
IDLE	get current position and keep the power off
FREE	get current position and send current compensating for gravity
CALIB	keep incrementing joint position until zero index is observed
SETP	call jsetp() and derive encoder position and compute observed sin and cos
HOLD	turn the power on
MANU	increment desired encoder position by 4 counts either clockwise or counterclockwise

Table 3. Real-Time State

In IDLE state, the system is in the initialization process. The free state is one in which the all joints are freed and compensate only for the gravity. This state is useful when we check the gravity loading constants we compute from the dynamics equations. In MANU state, the joints can be controlled manually in order to position the manipulator. The state CALIB indicates that the joints are going through a calibration procedure by looking for the zero indeces while making incremental moves. The state SETP is entered once the calibration is finished. Finally states STOP and EMGCY represent when the joints should stop and when the joints have detected abnormal conditions and need to come to a stop, respectively.

If the system is in SETP state, another variable *state*, in file *setp.c*, determines the stage in which the trajectory generation is. The number of states correspond to the number of cases in the motion control summary in [7], plus two additional states for the stationary case when there is no next motion command and for the case when the manipulator is coming to a stop. The state diagram in *state* is given in Figure 2.

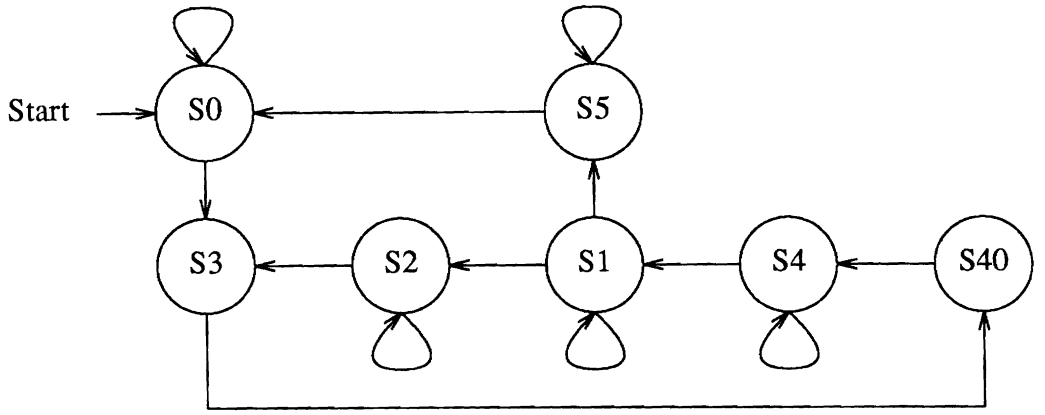


Figure 2. State Diagram of the Trajectory Generator

where the states are defined in Table 4.

state	*definition*
S0	wait for a new move request
S1	straight line motion segment
S2	one sampling period before the transition
S3	beginning period of the transition
S40	initialization of the transition
S4	during the transition
S5	end of a motion with no next move

Table 4. Definition of *state*

Whichever state the system is in, supervisor exchanges information with and for the rest of the system. Four data structures, also defined in *h/comm.h*, function as buffers holding information to be exchanged. The structure, S_MAIL, contains what to be shipped to the joints from the supervisor, J_MAIL, contains what to be shipped from the joints to the supervisor, and M_MAIL, contains information updated by the math processor for the joints. Another structure, PARCEL, contains information related to manipulator kinematics, such as dynamics and Jacobian matrices, that is provided to the joints at a low rate. In fact, each joint receives its new PARCEL every n periods, where n is the number of joints.

A few points concerning mails need to be clarified. First, there are three sets of sines and cosines returned from each joint in J_MAIL. The first two sets are expressed in terms of a sine and the sign the the cosine. They correspond to the sines and cosines

of the current and the next destination and positions, respectively. The third set is sine and cosine of the observed joint position. Secondly, The interpretation of the integer for the sign of the cosine is illustrated by the following figure where a clear bit in the corresponding position represents positive and set bit negative.

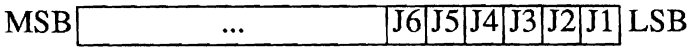


Figure 3. Bit Pattern Representing Signs of the Cosines

Thirdly, the fields in S_MAIL *Csigns* and *CsignsC* are simply the oring of the corresponding signes from all the joints.

The program *setp.c* depends on a number of functions. Functions *Dequeue()* and *Unqueue()* either take next motion request out of or and put back a fetched motion request to the motion request queue. *GetEX()* and *GetLDR()* compute the next T_6 in joint motion and Cartesian motion, respectively. All these functions are stored in file *expr.c*. Another function *InitD()*, defined in file *drive.c*, initializes the constant parts of the drive transformation for the next segment of Cartesian motion.

4.2. Joint Process

We describe the joint processes by showing how one joint works, since other joints are simply replicates of this example and differ mainly in the constants used in the programs. There are two joint independent programs, *jsetp.c* and *jrtc.c*, In addition, there is one joint dependent program in each joint directory, *jnti.c*, where *i* refers to the joint number, in each joint directory. The executable file of each joint is made up of the joint dependent and independent files. Joints share only the source code, not the executable code.

The program, *jnti.c*, contains the entry point, *main()*, that initializes joint dependent global variables and calls *rtc()* in *jrtc.c* to begin joint's operation. Two other functions in *jnti.c, InvKine(),* and, *InvKineC()*, compute inverse kinematics from two different set of parameters provided in supervisor mail. *ManuInc()* is used during manual mode to compute the amount of position increment. *IsReady()* determines if the joint should start calibrating or wait. This is necessary to overcome the mechanical coupling among joints during calibration. *AngToEng()* performs conversion between the encoder count and the joint angle in radians. *WriteEnc()* writes the change in its joint angle to the other joints that are coupled with this joint in order for them to make compensation. *ReadChgs()* copies the changes in other joints written in its memory into 'C' variables so as to be refered to later. Function *PID* calculates the control law.

Finally *StartS()* informs the supervisor of the completion of the joint's initialization.

The interrupt handler *Rtisr()* in *jrtc.c* is dictated by the same *rtstate* variable as on the supervisor to determine what the joint should do. It is executed at the same rate as the supervisor's interrupt service routine and computes the desired joint position in encoder count. According to the current *rtstate*, the fashion in which the desired position is computed varies. The result is passed on to the function *Servo()*, which actually performs servoing of the joint with the position computed in the previous sampling period. Currently it is either a PD or a PID control with gravity and friction compensations. Should the compliance be required, the servo error is adjusted in *Adjust()* before used to compute reacting torque.

JSetp() in *jsetp.c* computes the joint set-point. The variable *state* drives the process. There are several worth-noting points. First, all information needed by the joints is assumed to be available in the data structure MAIL, the buffer sent by the supervisor. Secondly, since in general the kinematic solution for *i*th joint requires the solution of inner *i*-1 joints, values of those joints computed in the previous sampling period are used in order for the joint not to wait for solutions to be computed, as has been mentioned previously. Finally, the joints should not have to wait for the supervisor to finish before they can start doing inverse kinematics. Instead, the T_6 is pipelined so that supervisor and joints start computing at the same time.

4.3. Math Process

The purpose of this process is to compute dynamic coefficients and Jacobian-relation matrices. Current computed joint angles are passed to this process as input and it provides gravity loadings and the compliance matrix as output to one joint per sampling period cyclically. The reason for only one joint per period is that the update of the parameters takes place at a much slower rate than the sampling rate and there is no point of sending XX The incoming information is deposited in MMAIL, the mail box for the math process from the supervisor and the output is returned in the buffer PARi, whose content applies to the joint specified in MMAIL.joint.

Again there is a real-time interrupt driven process that handles interaction with the supervisor and there is a background process that computes in an endless loop. The calculation of the dynamic coefficients is based on equations in [9], which uses Lagrangian mechanics to express dynamic terms explicitly and determines the constants in the coefficient from experiments. Procedures in [10] are used for the calculation of the compliance matrix. In order to prevent from happening the situation where the real-time interrupt service routine copies results partially updated by the interrupted process, a binary variable is used to indicate which of the two copies of a particular quantity, such as Jacobian matrix, is valid.

Currently only the Jacobian matrices from the base of the robot to the end-effector are considered. Should a tool be added to the system, modification would be necessary. Further, velocity dependent dynamic coefficients as well as the effects of a load at the robot end effector on the dynamics are not considered.

5. Postscript

One of the lessons we have learned from the RFMS project is that it is extremely difficult to program a multiprocessor system without a powerful development system. It is then predictably difficult to try to explain the system to someone wishing to understand and modify the system. To fully master the system requires a lot of time. It is however not as overwhelming to simply use the existing software to program the robot. This single document provides but a portion of the knowledge one must learn before he can feel comfortable working with the controller. It is strongly recommended that one read other related documentations and the hardware reference manual being prepared for this system for a better understanding.

Appendix A

RoboNet: A Local Area Network for Robot Systems

This documentation is about RoboNet, an Ethernet-based local area network that the we have designed and implemented. This documentation serves two purposes: as a user's guide to give robot system users a brief description on how to use the network software to transfer data from one machine to another, and as a system programmer's manual for those who maintain this network and those who are interested in customizing part of this network or extending it for other applications.

The remainder of this documentation is organized in four sections. Section two describes the network software function calls, their usage, and the results of those calls; Section three describes the network and its layers; Section four describes the logical link layer of RoboNet; and Section 5 describes the data link layer of RoboNet. Two appendixes describe how to compile the network software, where to find the files, and how to maintain the network software. For those who are interested only in using the software, we suggest that you read section two and three. For system programmers, we suggest that you read the entire documentation.

A.1. User's Guide

Currently only Grasp (VAX 11/785), Robo (Microvax II) and Intel 186/51 have RoboNet software. These machines are physically all attached to the Ethernet cable. We use RoboNet to transfer messages from the VAX machines to the Intel 186/51 and vise versa. Exchanges of messages among VAX machines are performed by software already available on these machines running Unix. The RoboNet is illustrated in Figure 1.

The VAX users can send messages to the Intel machines by invoking the network software. If the VAX user desires a particular piece of information from the Intel, he must send a message request to the Intel. The Ethernet communication on the Intel side is not accessible at the user level. A user can assume that process exists on the 186/51 that handles the messages and message request.

This appendix is an edited and revised version of the reference manual, *"RoboNet: A Local Area Network for Robot Systems"*, prepared by Pearl Pu, the Department of Computer and Information Science, the University of Pennsylvania.

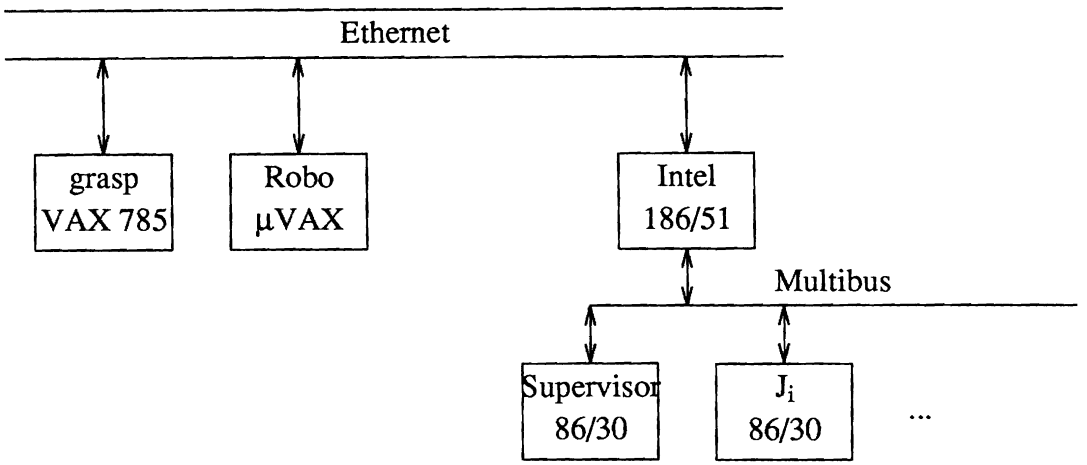


Figure A.1. RoboNet

A.1.1. The Network Software Function Calls for the VAX users

To be able to use these function calls, you have to have a Grasp, or Robo account. You have to know how to program in C. And finally you have to know what you are doing with these messages on the Intel side.

In order to use the software, you have to do the following:

1. Include *vax.h* in your program.
2. Compile your program with *vax_llc.o*.

The network software, seen at the user level, consists of the following C function calls: *Init_Comm_Link()*, *Sync()*, *Send()*, and *Recv()*.

Init_Comm_Link():

This function initializes the communication link between the host where the user is located and the Intel 186/51. The Intel Ethernet address is specified in this routine automatically as the destination address in sending and source address in receiving. Note that if the Intel address changes, one needs to notify the system programmer to modify this address accordingly.

Sync():

This routine synchronizes certain variables between the user process on the VAX and the communication process on the Intel 186/51.

Send(msgsptr, length):

msgsptr is a pointer to the buffer which contains the message you want to send, and *length* is an integer that specifies the length of the message string. Note that *length* can not be greater than *MAX_FRAME* or less than *MIN_FRAME* in *vax.h*.

Recv({type, msgsptr, length}):

type specifies what type of information you would like to receive from the Intel side. There are ten types of such information. *msgsptr* points to the buffer area where you want to receive the message. *Length* returns the actual length of message received. For certain reasons, all messages coming from Intel must be of one size. That size is specified by *R_SIZE* (receive packet size) in *vax.h.*

An example program, which illustrates how to use the network software on the VAX side, is shown in Figure 2.

```
main()
{
        int i;
        char msgs[100], buffer[R_SIZE];
        int length;

        /* fill up the msgs to be sent out */
        for(i=0; i<100; i++)
                msgs[i]= 'a' + ( i% 10);


        Init_Comm_Link();
        Sync();

        /* send the same message 10 times */
        for (i=0; i<10; i++)
                Send(msgs, sizeof(msgs));

        Recv(type2,buffer, &length);   /* receive type2 message */
        buffer[length]= NULL;
        printf("The received message is %s", buffer);
}
```

Figure 2. An Example Program

A.1.2. The Network Software on the Intel Side

Currently user support on the Intel side is entirely tailored to the need of the robot controller, which is a multiprocessor system based on Intel 86/30s with a supervisor handling message bookkeeping. All the messages sent from a user process on any of the VAXes or Microvaxes are queued up in a large buffer area on the 186/51. The

beginning address of the large buffer area is stored in the RAM of the 186/51 at 0x1ff00. The robot controller decides where each message finally goes. If the user requests a piece of information to be sent back to the VAX side, the network software on the Intel side will take care of this request.

To bring up the network process on the Intel 186/51, you have to ask the system programmer to do so. This process, once brought up, should be running continuously.

A.2. RoboNet: An Overview

RoboNet is a research effort to investigate the feasibility of designing a tailored local area network for robot systems, and stimulate further interest in this area. The current trend for robot systems is to distribute user tasks and robot tasks on different processors to increase computation speed. This introduces, however, communication problems between the users and the robot controller. To solve the communication problems, there are two solutions: one is to use existing software; the other is to design new software.

The reason we designed and implemented our own communication network stemmed from the observation that existing local area network protocols[11][12] are for large data file transfers. The header in each packet is usually complicated and the data large. If we use these protocols for transferring messages of small sizes, which is the situation with communication in robot systems, the system will be inefficient.

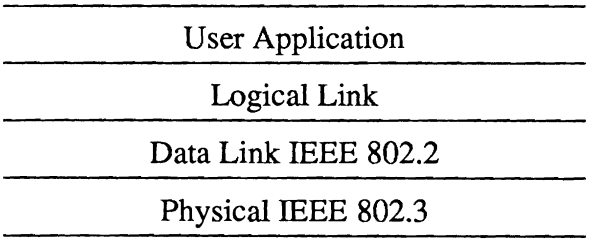


Figure A.2. Layers in RoboNet

RoboNet is designed with four layers as shown in Figure 3. The lowest layer, the physical layer, is an IEEE 802 standard. The data link layer is an IEEE compatible layer. IEEE 802.2 consists of data link and logical link layers. We only chose to implement the data link layer with the standard. It is hoped that RoboNet will be adaptable, should there be more suitable protocols. For instance, MAP (Manufacturing Automation Protocol) is another IEEE 802 standard. It is claimed that MAP is more efficient than Ethernet, and it does not degenerate when the load of the network becomes heavy. Therefore, if MAP is found to be more suitable for our application and affordable, we

can replace Ethernet by MAP without changing anything above. Another advantage of a standard implementation of the lower layers is to support heterogeneous machines. The robot system we have here contains VAX 11/785s, Microvax IIs, Intel microprocessors. In the future, it may also have Lisp machines. Since most computer manufactures now make Ethernet chips available to most of their machines, in order to install RoboNet on a machine we only have to install the upper three layers.

In the next two sections, we will describe the three lower layers. Section two is a description of the user application layer. Currently RoboNet is installed on Grasp (VAX 11/785), Robo (Microvax II), and Intel 186/51. As mentioned earlier, since this part of documentation is for system programmers, we will concentrate on not only design issues but also implementation details.

A.3. The Physical and Data Link Layers in RoboNet

As shown in Figure 3, the physical layer is the IEEE 802.3 (Ethernet) standard. On the VAX machines (VAXes, Microvaxes), this layer comes with the machine. On the Intel 186/51, there is a network coprocessor called the 82586, which is essentially an Ethernet chip that handles low level packet sending, receiving, framing, etc. For a detailed description of the 82586, refer to [13][14]. The 82586 is the coprocessor to the main CPU 80186.

The data link layer on the VAX machines uses the data link interface (DLI) from the Digital Equipment Cooperation. All packets sent out from the DLI are Ethernet packets. The DLI only takes care of damaged packets by verifying the check sum. Lost, duplicated, and out-of-order packets, however, are not taken care of.

On the Intel 186/51 microprocessor, the data link layer has to be implemented since there is no existing software. Fortunately, there is a manual[13] which describes how to program the 82586 coprocessor. We largely adopted an example from this manual as the data link layer. According to the manual, this example implements an IEEE 802.2 compatible data link layer.

Some differences between the example and our implementation are worth mentioning.

1. Multicast is not supported in our implementation.
2. The address for ISCP is found to be different in our case from that specified in the example. The correct ISCP address on our board is 0xff0 (absolute) instead of 0xfff0.
3. The interrupt from 82586 is the zeroth interrupt instead of the third.
4. Broadcast mode is disable, i.e., no broadcast messages from the Ethernet will be received.

A.4. The Logical Link Control Layer in RoboNet

We designed this layer. The principal mechanism used to prevent the network from losing, duplicating, and sending out-of-order packet is called one-bit-sliding window and positive acknowledgement with retransmission protocol[15]. We describe the characteristics of the logical link control (LLC) in RoboNet by describing the LLC packets and the algorithms used on both the Intel and the VAX sides.

A.4.1. The LLC Packet Types

SYNC:

This type of LLC packets take care of synchronization problems between the two sides. A network process runs on the Intel 186/51 continuously, whereas network processes come and go on the VAX side. Synchronization of sequence numbers is a problem if not taken care properly. We solve this problem by sending a SYNC packet every time a network process comes up on the VAX side. Upon receiving this packet, the Intel network process will initialize the sequence number.

ACK:An acknowledgement packet is sent out whenever the network process receives a good packet (i.e., with good check sum) other than an acknowledgement packet, that is, we do not acknowledge ACK packets.

REG:A regular packet will be passed to the host for processing if the sequence number matches expected frame number (specified by FrameExpected in *llc.c*). This is to ensure that no duplicated packet, from retransmission, is passed to the host.

SendReq:

A packet of this type can only be sent out from the VAX machines. This type of packet will cause a message to be sent out from the Intel to the network process on the VAX. For instance, a SendReq packet with T6 specified in the first byte will cause the T6 matrix, which is stored and kept updated on the Intel 186/51, to be sent to the VAX. This way, the robot system users can be updated with information from the Intel machines.

A.4.2. The Algorithm for the LLC on the VAX Side

procedure Send(type, msgsptr, length):
/ type: one of (ACK, REG, SYNC, SendReq)*
msgsptr: points to data to be sent
length: the length of message
Functionality: this routine prepares a LLC header for each
message pointed by msgsptr by adding the type, sequence fields,
then sends out the message. If an acknowledgement does not

```
  arrive within the timeout period, this routine will send out
  again the same message. It keeps doing so until either an ack
  arrives, or exceeds the allowed trial limit (maxtimeout).
*/

var f:frame;

if (type== SendReq)
        sendreq= TRUE;
f.type = type;                  /* specify packet type */
f.seq = NextFrameToSend;        /* append sequence number  */
f.data = msgsptr;

Acked= FALSE;
timeoutcnt=0;

/* keep trying if no ack, and # of tries has not exceeded the limit */

while( timeoutcnt < maxtimeout AND Acked== FALSE) do
        begin
                sendf(f);       /* transmit a frame */
                Timeout=FALSE;
                StartTimer();
                Recv_Ack;  /* timer can timeout in this routine */
        end;

if (timeoutcnt >= maxtimeout)
        write("Error: a frame is lost.");
Inc(NextFrameToSend); /* invert sender seq number */

end; /* end of Send */

procedure Recv_Ack():
/* Functionality: this routine waits for an acknowledgement to
  arrive from the other side. If timer times out, it will stop
  waiting and return to Send, which will resend the same message
  If an ack comes, it will set the flag to indicate so.
*/
```

```
var r : frame;       /* place to put received frame */

While (Acked== FALSE AND Timeout== FALSE) do
      begin
            wait(event); /*note: timer can timeout while waiting */
            if (event== FrameArrival AND r.seq== NextFrameToSend)
                  Acked== TRUE;

                  /* if the packet sent out was a sendreq,
                   * then acknowledge packet contains info. */

                  if (sendreq == TRUE )
                        To_Host(r);    /* pass message to host */
                  Inc(FrameExpected);
      end;

end; /* end of Recv_Ack */

procedure Isr_Timer():
/* Functionality: this routine will be called when the timer times out.
*/
Timeout=TRUE;
timeoutcnt= timeoutcnt+1;

end;  /* end of Isr_Timer */

/* type specifies what type of information to be sent back
  msgsptr returns the address of received message
  length returns the length of received message
  Functionality: Receiving a message is similar to sending a message.
  The requested message is sent back from the Intel in the Acknowledge
  packet. This is called piggybacking.
*/

var   req : frame;

req.data[0] = type;       /* specify what information to receive */
Send(SendReq, req, sizeof(req));       /* send a request frame */
```

```
end;  /* end of Recv */

procedure Sync():
/* Functionality: This routine sends out a packet to synchronize
   sequence numbers on both VAX and Intel side.
*/

var f:frame;

Send(SYNC, f, sizeof(f));

end;  /* end of SYNC. */
```

A.4.3. The Algorithm for the LLC on the Intel Side

```
procedure Recv_Frame(f):
/* f points the received frame
   Functionality: This procedure is invoked when 82586 receives a frame
   and issues an interrupt to CPU. It does different things according
   to the type of messages it received.
*/

case f.type

        ACK:  /* there will be no ACK frame on the Intel side */

        REG:        Send_Ack (f.seq);
              if (f.seq == FrameExpected)
                   putf(f.data);        /* put f in big buffer */
                    Inc(FrameExpected);    /* invert seq */

        SYNC: Send_Ack( f.seq );
            FrameExpected=0;  /* reinitialize */
            NextFrameToSend;  /* reinitialize */

        SendReq:  Ans_Send_Req();    /* answer send request */
                 if (f.seq == FrameExpected )
                  Inc(FrameExpected);
```

```
end;  /* of case */

end;  /* of Recv_Frame */

procedure Send_Ack( seq ):
/* Functionality: this routine sends out an acknowledgement packet.
*/
var f: frame;

f.type= ACK;
f.seq= seq;
sendf (f);     /* transmit a frame */

end; /* of Send_Ack */

procedure Ans_Send_Req (seq);
/* Functionality: this routine piggyback the requested information
   in the acknowledgement packet.
*/

var f : frame;

f.type= ACK;
f.seq= seq;
f.data = getf(data);
sendf(f);      /* transmit a frame */

end; /* of Ans_Send_Req  */
```

A.5. Miscellaneous

The data link layer for the 186/51 is contained in file *dld.c*. The packet size from the Intel controller to the VAX can be changed by modifying constant *R_SIZE* in *vax.h*, in *llc.h*, and the field in *so_addr.choose_addr.dli_eaddr.dli_protype* in *vax_llc.c*. If you get errors like "ERROR: enable toget CB, TBD, or FD", you should consider to increase the size of the CB, or TBD, or FD queues by changing the *CB_CNT*, *TBD_CNT*, or *FD_CNT* in *dld_llc.h*.

Appendix B

Use of 8086 Cross Compiler Under Unix

B.1. Introduction

This document is interesting to those who intend to program an 8086/87-based single board computer under a VAX/Unix environment. The compiler introduced here was initially obtained from MIT Laboratory for Computer Science; however, it was written for an IBM-PC/MS-DOS environment. Modification to this compiler is mostly done to the I/O library and math library. In addition, Intel's iSDM (System Debug Monitor) is incorporated to the system to allow both down-loading of users' programs and debugging of them. Efforts have been made to optimize the intermediate assembly programs generated by the compiler so that a 15 to 30 percent better performance can be achieved after running the optimizer.

This document serves as a users' manual of the cross compiler without elaborating on the details. It assumes a user to have experience with C language and Unix. Knowledge of 8086/87 assembly language is necessary for debugging a program.

Throughout the discussion, host computer refers to the one where you develop your programs. The target computer is the 8086-based single board computer. Unix C compiler is simply called compiler and the cross compiler is explicitly qualified.

Running a C program consists of several steps. First, you should properly connect the hardware. The search path of your account should be set up correctly so that you can access the library files. The compilation of your C program using cross compiler follows similar syntax as to those of the C compiler. Before running an executable file, it should be down loaded to the target computer. Finally, you can run your program with the help of Intel's System Debug Monitor (referred to as SDM from now on).

B.2. Cable Hook-up

Your interface to both Unix and the target computer is all done from a single terminal. Normally, your terminal acts just like a regular Unix terminal and the target computer is simply another tty to the same host computer. You should connect your terminal to the a tty line and the serial port of the target to another tty line, both using standard RS232. After the lines are connected and power plugged in, turn on the switch of the the target system and initialize its line to Unix by

```
% stty 9600 raw -echo > /dev/ttyxy
```

where xy is the target's tty number.

B.3. Down Loading the Loader via SDM

Setting up your path on Unix correctly is important because your program need to find the libraries and you need to access several executable files. The directory of these files is machine dependent, but on Upenn-GRASP, the following in your .cshrc or .tcshrc is adequate:

set path=($path /usr/users/hz/c86/lib86)

If you are a shell user, use in .profile

PATH = $PATH:/usr/users/hz/c86/lib86

export PATH

Initiate the communication with the target by kermit function of Unix which changes your Unix terminal to a virtual terminal of the target. Kermit is invoked by the following command:

% kermit clb /dev/ttyxy 9600

You are then communicating to the target through the SDM from this point on. The SDM responds with the following message followed by either a dot (.) or asterisks(*), the latter indicating that SDM has not been booted and you are talking to it for the first time.

iSDM 86 Monitor Vx.y
Copyright 1983 Intel Corporation

To boot, type capital U and you will see the monitor respond with a dot indicating it has been booted. To exit kermit thereby exiting SDM upon completion of your job, type ^ followed by a letter c and message "C-kermit Disconnected" will be printed.

Although you could use SDM to down load your application program, the slow loading speed prohibits development of any large program. Alternatively, a fast loader is available to directly read your program from serial port and store it into memory without going through SDM. The idea is then to load the fast loader with SDM and to load your program with the fast loader. To load the fast loader, type:

% ldld

You will then asked if the tty of the target is the right one such as

ttyh3? (y/n)

You should answer accordingly. The loaded data and the corresponding addresses will echo on the screen. This fast loader is invoked later by the dl command to load the

application program.

B.4. Cross Compiler

As a C programmer, you may be used to writing programs under Unix and not aware of what is C and what is Unix. Therefore, it is important that you read through this document before attempting to write any C program. Basically, C is a high level language that allows you to express your algorithms in terms of C functions, whereas Unix is an operating system which provides C with an environment. Many things you use in the form of function calls are intrinsic to Unix, such as multi-processes, file systems, and I/O interface. When your program is intended for an 8086/87, many utilities on Unix are no longer available on your target board. For example, you can not open files or write to a file. Any library with which your program is linked must be created for 8086/87.

Theoretically, the language definition of the C cross compiler is 100% compatible with Unix C, i.e., all variable types, data structures, operations, type specifications, etc. follow the conventions in [16]. However, there are major differences between this compiler and Unix C compiler in the Unix interface and I/O libraries. In fact, the only system calls you can make are limited to those of standard I/O (see in Appendix A), although they may expand in the future. The reason for not implementing them is obviously that your single board computer does not contain a sophisticated operating system which actually provides these system functions. Our thought on I/O library support was that a total compatibility would require a major undertaking which may not be necessary although not impossible.

The options accepted by the compiler are the following:

-P	run only the C preprocessor (cpp) and leave the result in prog.i, where prog.c was the input file.
-S	do not run the assembler, leaving the assembly language output file in prog.a86, where prog.c was the input file.
-c	compile, assemble, but do not create a .com file, leaving binary file in prog.b, where pr.c was the input file.
-o name	changes the name of the generated default a.abs file to "name.abs".
-lm	links the program with the mathematics library
-lr	links the program with the RFMS library
-Ilib	specifies a directory to be searched when processing #include statements during preprocessor stage.

To cross compile your programs for the 8086/87 target system, use the shell script cc86 as

% cc86 [options] ...file ...

Unless -o option is specified, the default name of the output is a.abs, instead of a.out, where abs stands for absolute file. It has a format understandable by the fast loader and, apparently, it can not be executed on the host computer. The input to cc86 can be more than one file; it can be a combination of assembly programs, object files, and C programs. There are two standard libraries: I/O library, which is always linked with your programs, and the math library. Read Appendix B for the math functions provided by the math library.

As usual, there are three parts to this cross compiler: a compiler that produces assembly programs from input C programs, a assembler that reads the output of the compiler and the input assembly programs and assembles them to the object files, and a linker that links everything together. Unfortunately, the intermediate assembly language, A86, is not standard ASM-8086 assembly language but a hybrid between ASM-8086 and VAX-11 assemblers; nor is it equivalent to ASM-8086 particularly in its instructions dealing with data allocation and the floating point stack. Therefore, if you need to write assembly programs, the best you could do is using -S option of the cross compiler to generate sample assembly programs and figure your way out, with the help of 8086/87 and VAX-11 literatures [17][18][19][20]. Appendix C contains a table of encodings of 8087 stack arithmetic instructions, which may be useful when you need to program 8087 and would like to achieve efficiency.

Because of the nature of the program execution, the main program can no longer have arguments argc and argv, which are usually handled by the operating system. Also be warned that you are at your own risk if you do not initialize variables, local or global. Your target computer does not do everything the Unix does such as initializing memory. Failure to comply to this may result in meaningless outcomes. We have also found that the cross compiler can not handle functions which return a float; you must define these functions to return a double. Further, when a function is declared double, it must have a return statement to avoid underflow of the float stack on 8087. Finally, an integer variable on 8086 is 16 bits long rather than 32 as on VAX and a double is eight bytes.

B.5. Down Loading Your Application

The next step is to load your program to the memory of the target. The default location of the starting address of your program is at hex 1000 or 4 kilobytes from the beginning. This information is useful later when you debug your program. To load the program, simply type:

% dl <abs file>

The down loading speed is about one kilobytes per second, or 9600 baud. You may examine the size of your program to figure out how long a down loading takes.

The location of the code segment and data segment can be at any 16-byte boundary by changing two constants in the down loading program. Currently, the memory format of the target is set to the following diagram:

```
---------------------- 0x0000

  Reserved for SDM

---------------------- 0x0900
    Fast Loader
---------------------- 0x1000 CS (code segment)

       Code

---------------------- 0x10000 DS, SS, ES

   Data + Stack

---------------------- 0x1ff00 = top of the stack
                SP (stack pointer) = 0xff00
---------------------- End of 128 K Ram
```

The size of your programs is limited to almost an 8086 segment and can be as large as 60 kilobytes. Data and stack may take another 64K segment less 256. The sizes are examined by the linker and warnings are issued when the actual sizes exceed or approach the limits.

B.6. SDM - System Debug Monitor

SDM is an assembly language level debugger that offers such features as disassembling code, single step, changing register and memory contents, break point, and displaying register and memory contents. You can monitor your program on the target directly from your Unix terminal with the help of the on board SDM through kermit which changes a Unix terminal to that of your target computer. As mentioned above, this can be done by

% kermit clb /dev/ttyxy 9600

and you will also see SDM respond as before. In case it has crashed for any reason, push the reset button of the target and type capital U to reboot the system.

We will try to explain a few commands that are particularly useful in executing your program. It is strongly recommanded that you read [21] if you really want to learn how to use SDM. This section gives just a tiny subset of the rich debugging commands of SDM.

B.6.1 X Command

This command allows you to examine and modify registers.

.x

will display all the 8086 registers.

To modify a register, do

.x register = value

where value can be a hexadecimal number, another register, or an expression of the sum or difference of numbers and registers.

.xn

displays the 8087 registers and stack registers and you can change the values of stack registers by

.xst(i) = real number

where i is the stack register number from 0 through 7 and real number is represented in exponential notation such as 1.23 e-4

B.6.2 D Command

This command displays memory contents in a given data type which can be integer(i), long integer(li), long real(lr), short integer(si), short real(sr), binary code decimals(t), temporary real(tr, ten bytes), word(w), or disassembled instruction(x). Address is represented as segment:offset. The default segment is code segment(cs) and default offset is instruction pointer (ip). For example,

.14dx
displays 14 disassembled instructions from location cs:ip.
.d ds:5#16t
displays 16 decimal bytes in both hexadecimal and ASCII format, beginning at ds:5.
.5dtr 10
displays five temporary real values, beginning at cs:10 in both temporary real hexadecimal and decimal format.

B.6.3 G Command

This command instructs the monitor to begin executing your program at the current cs:ip. It can be followed by a starting address and addresses where you want to break the program. For example,
.g 7fa, 1f0:e20
will stop either at cs:7fa or 1f0:e20, whichever comes first.
.g 2d0:113, ip
tells the monitor to begin execution instructions at 2d0:113 and continue until it gets to the current cs:ip.

When the program stops at a break point, the following message is printed.
*BREAK at xxxx:yyyy

B.6.4. Bugs

As usual, there are bugs associated with SDM package. The single step feature is shaky at times when you use 8087. For example, to step through a program by G command may generate a different result from that you obtain to go all the way by G command; or when you single step, the board may not do what the next instruction says it will do, etc. We have no solutions to this and encourage you to ask Intel for help.

B.7. Miscellaneous

In lib86 directory, there exist several utility programs to convert files from one format to another.

abshex - converts an abs file to a hex file,
ldabs - converts an ld file (output of MIT compiler) to an abs file,
ldhex - converts an ld file to a hex file.

B.8. An Example

In this section, we will go through an example to demonstrate how the cross compiler and the debugger work. Suppose you have created the following program on Unix:

```
# include <math.h>
# define RAD_TO_DEG 57.29578

main()
{
        double x, y;
        int i;

        x = 0.1;
        for(i = 0; i < 10; i++) {
                y += x;
                printf("sin(%4.1f) = %f\n\r", y*RAD_TO_DEG, sin(y));
        }
}
```

First compile the program using the C compiler and test it on Unix as

```
% cc prog.c -lm
% a.out
sin( 5.7) = 0.099833
sin(11.5) = 0.198669
sin(17.2) = 0.295520
sin(22.9) = 0.389418
sin(28.6) = 0.479426
sin(34.4) = 0.564642
sin(40.1) = 0.644218
sin(45.8) = 0.717356
sin(51.6) = 0.783327
sin(57.3) = 0.841471
```

Of course, on Unix we can only test the portion of the program not dependent on the target hardware.

After making sure the program is free of errors as far as you can go on Unix, you can then cross compiler your program:

```
% cc86 prog.c -lm
```

An a.abs is created at this point for you to down load. You are then ready to try it out on your target computer. As the first step, properly connect the Unix tty (e.g. ttyh3) line to your target computer and turn on the power. A typical sequence of commands may look like:

```
% stty 9600 raw -echo > /dev/ttyh3
% kermit clb /dev/ttyxy 9600

iSDM 86 Monitor Vx.y
Copyright 1983 Intel Corporation

*** (capital U is pressed here)

.(^c)
C-Kermit Disconnected
% ldld
ttyh3 ? (y/n) y
S 0090:0000
0090:0000 00 - b8,
0090:0001 00 - 90,
0090:0002 00 - 00,
0090:0003 00 - 8e,
   .
   .
   .
0000:007F FF - 00,
0090:0080 FF -
```

```
% dl a.abs (wait approximate 5 seconds)
% kermit clb /dev/ttyxy 9600
iSDM 86 Monitor Vx.y
Copyright 1983 Intel Corporation

.x
AX = 0006 CS = 0100 IP = 0000 FL = F046 O0 D0 J0 T0 S0 Z1 A0 P1 C0
BX = 1AE3 SS = 1000 SP = 0000 BP = 0000
CX = 0000 DS = 009B SI = 0000
DX = 00D8 ES = 0000 DI = 0000
.x ip=0
.np,

0100:0000 FA          CLI -,
0100:0001 B83F13      MOV AX, 133FH        ;I = +4927-,
0100:0004 B104        MOV CL, 4

.g
sin( 5.7) = 0.099833
sin(11.5) = 0.198669
sin(17.2) = 0.295520
sin(22.9) = 0.389418
sin(28.6) = 0.479426
sin(34.4) = 0.564642
sin(40.1) = 0.644218
sin(45.8) = 0.717356
sin(51.6) = 0.783327
sin(57.3) = 0.841471

*BREAK at 0100:002B
.x
AX = 0006 CS = 0100 IP = 0020 FL = F046 O0 D0 J0 T0 S0 Z1 A0 P1 C0
BX = 1AE3 SS = 1000 SP = FF00 BP = 0000
CX = 0000 DS = 1000 SI = 0081
DX = 00D8 ES = 1000 DI = 0000
.(^c)
C-Kermit Disconnected
%
```

You are now at the end of a debugging session.

B.9. I/O Library

Only standard input and output functions are provided by the library, i.e., input to the program and output from the program can only go through your terminal. Furthermore, I/O functions are restricted to the following. Attempt to invoke any other will result in an undefined function error.

```
char getchar();
char *gets();
putchar(ch) char ch;
putw(word) int word;
puts(s) char *s;
printf(s, arg) char *s;
```

It should be pointed out that the line feed character '\n', when used to obtain a new line, must be accompanied by a carriage return '\r' in order to move the cursor back to the beginning of the next line. This second character is put out by Unix automatically so that your printing program need not use it explicitly.

B.10. Math Library

The following math functions are provided in the math library.

```
double fabs(), ldexp(), modf();
double sqrt();
double sin(), cos(), tan(), asin(), acos(), atan(), atan2();
double sc(sc_p, angle)
struct sncs *sc_p; double angle;
where sncs is
struct sncs {
        float sin;
        float cos;
};
```

B.11. 8087 Floating Point Stack Programming

The compiler does not make use of the floating point stack registers one through seven for the sake of simplicity. On the other hand, at times you may desire to achieve better efficiency by programming in A86 and taking advantage of the floating registers. Unfortunately, the A86 does not provide instructions which handle the float stack registers except for the top, it is necessary to program in 8087 machine code directly. The following table provides some of the frequently used arithmetic instructions to

manipulate on the float stack. An example is also presented to illustrate the idea and the technique.

Instructions	i=0	i=1	i=2	i=3	i=4	i=5	i=6	i=7
fadd st, s(i)	0xc0d8	0xc1d8	0xc2d8	0xc3d8	0xc4d8	0xc5d8	0xc6d8	0xc7d8
fadd st(i), st	0xc0dc	0xc1dc	0xc2dc	0xc3dc	0xc4dc	0xc5dc	0xc6dc	0xc7dc
faddp st(i), st	0xc0de	0xc1de	0xc2de	0xc3de	0xc4de	0xc5de	0xc6de	0xc7de
fsub st, st(i)	0xe0d8	0xe1d8	0xe2d8	0xe3d8	0xe4d8	0xe5d8	0xe6d8	0xe7d8
fsubr st, st(i)	0xe8d8	0xe9d8	0xead8	0xebd8	0xecd8	0xedd8	0xeed8	0xefd8
fsub st(i), st	0xe8dc	0xe9dc	0xeadc	0xebdc	0xecdc	0xeddc	0xeedc	0xefdc
fsubr st(i), st	0xe0dc	0xe1dc	0xe2dc	0xe3dc	0xe4dc	0xe5dc	0xe6dc	0xe7dc
fsubp st(i), st	0xe8de	0xe9de	0xeade	0xebde	0xecde	0xedde	0xeede	0xefde
fsubrp st(i), st	0xe0de	0xe1de	0xe2de	0xe3de	0xe4de	0xe5de	0xe6de	0xe7de
fmul st, st(i)	0xc8d8	0xc9d8	0xcad8	0xcbd8	0xccd8	0xcdd8	0xced8	0xcfd8
fmul st(i), st	0xc8dc	0xc9dc	0xcadc	0xcbdc	0xccdc	0xcddc	0xcedc	0xcfdc
fmulp st(i), st	0xc8de	0xc9de	0xcade	0xcbde	0xccde	0xcdde	0xcede	0xcfde
fdiv st, st(i)	0xf0d8	0xf1d8	0xf2d8	0xf3d8	0xf4d8	0xf5d8	0xf6d8	0xf7d8
fdivr st, st(i)	0xf8d8	0xf9d8	0xfad8	0xfbd8	0xfcd8	0xfdd8	0xfed8	0xffd8
fdiv st(i), st	0xf8dc	0xf9dc	0xfadc	0xfbdc	0xfcdc	0xfddc	0xfedc	0xffdc
fdivr st(i), st	0xf0dc	0xf1dc	0xf2dc	0xf3dc	0xf4dc	0xf5dc	0xf6dc	0xf7dc
fdivp st(i), st	0xf8de	0xf9de	0xfade	0xfbde	0xfcde	0xfdde	0xfede	0xffde
fdivrp st(i), st	0xf0de	0xf1de	0xf2de	0xf3de	0xf4de	0xf5de	0xf6de	0xf7de
fld st(i)	0xc0d9	0xc1d9	0xc2d9	0xc3d9	0xc4d9	0xc5d9	0xc6d9	0xc7d9
fxch st(i)	0xc8d9	0xc9d9	0xcad9	0xcbd9	0xccd9	0xcdd9	0xced9	0xcfd9
fst st(i)	0xd0dd	0xd1dd	0xd2dd	0xd3dd	0xd4dd	0xd5dd	0xd6dd	0xd7dd
fstp st(i)	0xd8dd	0xd9dd	0xdadd	0xdbdd	0xdcdd	0xdddd	0xdedd	0xdfdd

Table A.1. Encodings of 8087 Float Stack Arithmetic Instructions

Suppose you would like to program a partial sinus function using 8087's partial tangent call. It may look like:

```
        .globl _psin
|       double psin(x) x double; compute sinus of x in radians
_psin:  mov bx, sp
        fldd   *2(bx)
       fptan
       fwait
       .word  0xc8d8            | fmul st, st(0)
       fwait
       .word  0xc1d9            | fld   st(1)
       fwait
       .word  0xc8d8            | fmul st, st(0)
       fwait
       .word  0xc1de            | faddp       s(1), st(0)
       fsqrt
       fwait
       .word  0xf9de            | fdivp       st(1), st(0)
       ret
```

Note that every instruction must be preceded by a float wait o instruction to assure normal function of the hardware. Also, if you are serious about programming 8087, always remember to clean up the float stack before exiting a function, with the return value of the function on the stack if there is any. Pushing too many things on to the saturated float stack leads to unexpected result as the values at the bottom of the stack will not drop out as one would think.

REFERENCES

[1] *Paul, R.P. and Zhang, H.* 1985. "Design of a Robot Force/Motion Server". Proceedings of IEEE International Conference on Robotics and Automation, St.Louis, MO.

[2] *Paul, R.P., Zhang, H., Hashimoto, M., Durrant-Whyte, H., Izaguirre, A., Trinkle, J., Zhang, Y., Fuma, F., Ulrich, N., and Donham, M.* 1986. "A Distributed System for Robot Manipulator Control", Department of Computer and Information Science, the University of Pennsylvania. 1986.

[3] *Hayward, V. and Paul, R.* 1984. "Introduction to RCCL: A Robot Control C Library", Proceedings of IEEE International Conference on Robotics and Automation, Atlanta, GA.

[4] *Pu, P.* 1986. "RoboNet: A Local Area Network for Robot Systems", Department of Computer and Information Science, University of Pennsylvania.

[5] *Paul, R.P.* 1981. "Robot Manipulators: Mathematics, Programming, and Control", MIT Press.

[7] *Paul, R.P. and Zhang, H.* 1984. "Robot Motion Trajectory Specification and Generation", ISRR Proceedings , Japan.

[8] *Zhang, H. and Paul, R.P.* 1988. "A Parallel Solution to Robot Inverse Kinematics", Proceedings of IEEE International Conference on Robotics and Automation, Philadelphia, PA.

[9] *Izaguirre, A., Hashimoto, M., and Paul, R.* 1987. "A New Computational Structure for Real-time Dynamics". Proceedings of International Workshop on Robotics: Trends, Technology, and Applications, Madrid, Spain.

[10] *Paul, R. P. and Zhang, H.* 1986. "Computationally Efficient Kinematics for Manipulators with Spherical Wrists Based on the Homogeneous Transformation Representation". *International Journal of Robotics Research* 5(2):32 - 44.

[11] *Postel, J.,* 1980. "User Datagram Protocol", RFC 768, Information Sciences Institute.

[12] *Postel, J.,* 1982. "TCP-IP Implementations", Network Information Center, SRI Int.

[13] *Intel* 1985. "Local Area Networking (LAN) Component User's Manual", 230814-002, Intel Corporation.

[14] *Intel* 1984. "iSBC 186/51 COMMputer Board Hardware Reference Manual," 122136-002, Intel Corporation.

[15] *Tanenbaum, A.,* 1981. "Computer Networks", Englewood Cliffs, N.J., Prentice-Hall,

[16] *Kernighan, B.W and Ritchie, D.M.* 1978. "The C Programming Language", Prentice-Hall, Inc., Englewood Cliffs, New Jersey 07632.

[17] *Intel Corporation,* "iSBC 337 Multimodule Numeric Data Processor Hardware Reference Manual", Intel Corporation, 3065 Bowers Avenue, Santa Clara, California 95051.

[18] *Rector, R. and Alexy, G,* "The 8086 Book", Osborne/McGraw-Hill, 630 Bancroft Way, Berkeley, California 94710.

[19] *Levy, H.M. and Eckhouse, R.H.,* "Computer Programming and Architecture", Digital Equipment Corporation, Bedford, MA 01730.

[20] *Intel Corporation,* "ASM86 Language Reference Manual", Intel Corporation, 3065 Bowers Avenue, Santa Clara, California 95051.

[21] *Intel Corporation,* "iSDM 86 System Debug Monitor Reference Manual", Hardware Reference Manual", Intel Corporation, 3065 Bowers Avenue, Santa Clara, California 95051.

[22] *Intel* 1982. "iSBC 86/14 and iSBC 86/30 Single Board Computer Hardware Reference Manual," 14404-002, Intel Corporation.