February 1991

TRACS Users Manual and Software Reference Guide

Eric Paljug
University of Pennsylvania

Abstract
The Two Robotic Arm Coordination System (TRACS) of the GRASP Lab is designed to perform experiments in dynamic two arm control. The system is comprised of two PUMA 250 robot arms with modified controllers, a PA-AT host computer and an AMD 29000 high speed floating point processor board. This manual describes the system software architecture and the software interfaces between the system elements. It is intended to aid in developing software for the system.

Comments
University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-91-10.

TRACS Users Manual and Software Reference Guide

MS-CIS-91-10
GRASP LAB 254

Eric Paljug

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

February 1991

TRACS Users Manual
and
Software Reference Guide

Eric Paljug
General Robotics and Active Sensory Perception
(GRASP) Laboratory
University of Pennsylvania
3401 Walnut Street, Room 301C
Philadelphia, PA 19104

February 6, 1991

ABSTRACT

The Two Robotic Arm Coordination System (TRACS) of the GRASP Lab is designed to perform experiments in dynamic two arm control. The system is comprised of two PUMA 250 robot arms with modified controllers, a PC-AT host computer and an AMD 29000 high speed floating point processor board. This manual describes the system software architecture and the software interfaces between the system elements. It is intended to aid in developing software for the system.

Contents

1 **Introduction** 1

2 **Disk Organization** 1
- 2.1 The YARC Subdirectory . . . 2
- 2.2 The PKG Subdirectory . . . 2
- 2.3 The USER directory . . . 3

3 **The Yarc System Software** 4

4 **The Real-time Interrupt** 6

5 **Communication Interfaces** 7
- 5.1 PC-AT to AT-Super . . . 7
- 5.2 PC-AT to Unimation Controller . . . 8
- 5.3 PC-AT to Sensors . . . 9

6 **Creating Application Software** 9
- 6.1 Development . . . 9
- 6.2 Development features . . . 10
- 6.3 Compiling and linking . . . 11

7 **Running Application Software** 11
- 7.1 Debugger . . . 13

8 **Advanced Topic: Adding New Hardware** 14
- 8.1 Example: Adding a New Analog Board . . . 14

A **AT-Super Manual** 17

B **PC interrupt notes** 42
- B.1 Application Notes . . . 42
- B.2 Sample Listing . . . 42

C **TRACS Utility Programs** 44
- C.1 Encoder Calibration . . . 44
- C.2 Limp . . . 46
- C.3 Slow . . . 46
- C.4 Maintenance . . . 46

\dos	\lib	**\yarc**
\bin	\util1	**\pkg**
\include	\util2	**\usr**

Figure 1: PC-AT Root Directory

1 Introduction

This document is the primary reference for writing software for the Two Robotic Arm Coordination System (TRACS) of the University of Pennsylvania's Grasp Lab. A separate reference exists for the hardware components of TRACS, and is usually located in a binder near the robots.

The system controls two PUMA 250 robots with an IBM PC-AT (compatible) based system. The PC-AT is host for an AMD 29000 high speed floating point processor based board that does the servo loop calculations. This board is called the AT-Super and is produced by YARC Systems Corporation. The PC-AT provides the real-time environment for this system and the communications between the AMD 29000 and the PUMA robot joint boards. The software for the entire system is written in C. The code that runs on the AT-Super uses the HiC compiler by MetaWare and the PC-AT code uses the Microsoft C compiler.

This document provides information at many different levels of software development. At the highest level, it provides a basis for writing software to control the robots and incorporate existing software libraries, such as kinematics, friction compensation, gravity compensation, etc. At the intermediate level, it provides software guidelines for incorporating new hardware into the system. At the lowest level, it explains the real-time environment, communication and synchronization between the PC-AT, the AT-Super, and the PUMA robots. The interested TRACS user should be familiar with all the material in this document, however only the higher level topics are most important to those how wish to use the existing code without much modification in order to perform basic manipulation with the system.

This document is organized as follows: the location of software on the system is presented in Section 2 to provide a starting point and quick reference for programmers; Section 3 details the changes made to the software sources of the AT-Super supplied by the manufacturer for the TRACS application; the real-time environment on the PC-AT is covered in Section 4; the communication between the components of TRACS is developed in Section 5; Sections 6 and 7 explain how to create and write control application code for TRACS; and the final section discusses how to incorporate future additions. The appendices include: the original AT-Super Manual, the PC interrupt software application notes, and documentation on TRACS utility programs (e.g., calibration).

2 Disk Organization

This section will describe the organization of software on the hard disk of the PC-AT. It is intended to be an overview of the system and a guide to where the programs are located. The Hard disk of the PC-AT is divided into directories as shown in Figure 1. The top six subdirectories contain DOS, the C compiler and the standard binary files found on most of the PC's in the GRASP lab.

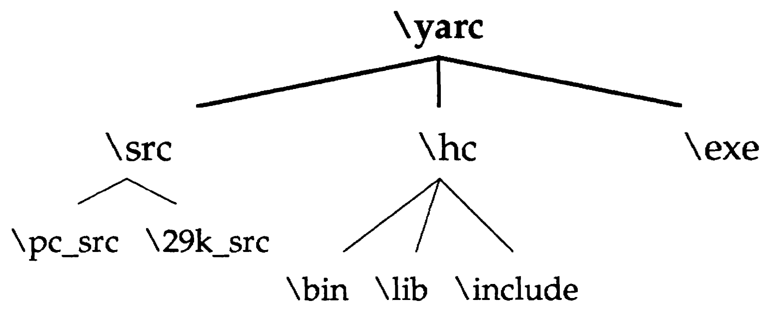


Figure 2: Yarc Directory

The remaining three subdirectories deal directly with the robot workcell. The following sections will briefly expand upon the organization of these latter three subdirectories.

2.1 The YARC Subdirectory

This directory is the root for all the software related to the YARC AT-Super coprocessor board. The `\yarc` directory and its subdirectory are depicted in Figure 2. There are other minor branches not shown in the figure that contain demo and example programs. The first level of the tree separates between the utility sources, the HiC compiler for the AT-Super, and the utility executable files.

Since the YARC AT-Super's executable is compiled, linked and loaded from the PC, there is a large segment of the utility code that is written for the PC and is compiled with Microsoft C. This is located in `\yarc\src\pc_src`. Note that this directory has a sub-directory containing the original sources because the sources in `\pc_src` are modified. The `\yarc\src\29k_src` directory contains the AMD 29000 software that is initially installed on the AT-Super and functions as its bootstrap. These source files were obtained from Yarc Systems Corp. through a nondisclosure agreement.

The compiler for the AMD 29000 processor of the AT-Super is located under the `\yarc\hc` directory. As with many C compilers, it is divided into `bin`, `lib`, and `include` subdirectories.

The executable version of the utilities that pertain to the AT-Super are located in `\yarc\exe`. The utility functions are used to reconfigure the AT-Super as well as to load and service the application code.

2.2 The PKG Subdirectory

This directory is the root for all software packages that are complete self-contained modules. The tree is shown in figure 3. As new sensors or boards are added to the system, they should be assigned an individual branch of the `\pkg` tree. Below is a brief description of each of the above branches.

- The `\p250` directory are the programs that deal with the PUMA 250 robots at their controller level. Programs to communicate with the controller, calibrate the robots and free (limp) the actuators are found here.

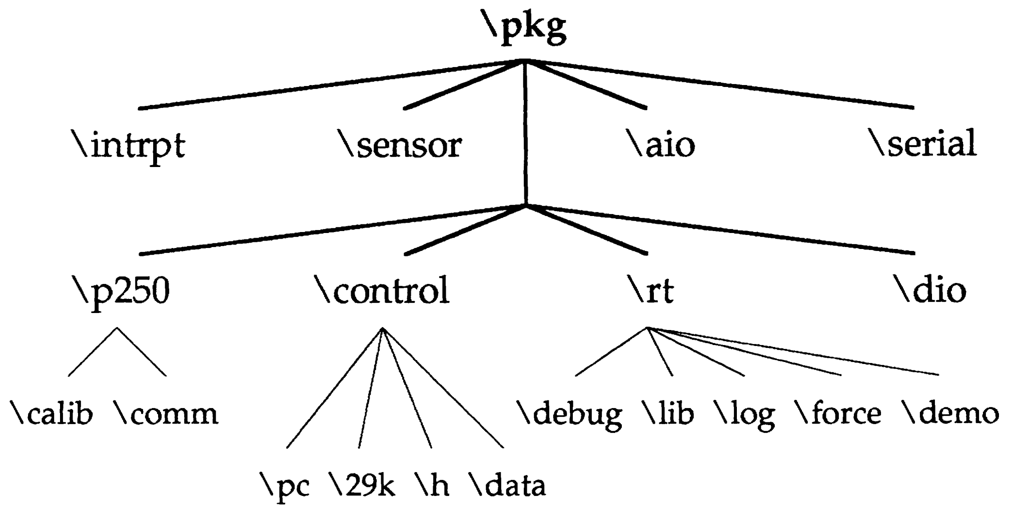


Figure 3: Package Directory

- The `\control` directory contains the skeleton of the control process. Programs that deal with the PC-to-29000 synchronization and communication are found here. Additionally, the foreground user interface code is also located here.
- The `\rt` directory has the programs that actually do the trajectory generation and servo control at the lowest level. These programs have robot encoder and sensor data as input and motor current as their output. They run on the YARC AT-Super. The subdirectories contain demonstration programs, libraries for common routines (such as kinematics), an offline control algorithm debugger, controller logged data, and other applications.
- The `\intrpt` directory contains the software that reconfigures the PC clock interrupt thus creating a real-time interrupt on the PC.
- The `\dio` directory has all the programs that deal with the digital I/O boards. These boards are used in the parallel communications to the PUMA 250 controllers.
- The `\aio` directory contains the programs that perform analog I/O from the various analog sensors and actuators (excluding the joint motors of the robots) in the workcell.
- The `\sensor` directory contains software used with the LORD force-torque sensor.
- The `\serial` directory contains a package of software that provides the function of a serial communications driver.

2.3 The USER directory

This directory should contain the application files specific to the workcell user. Depending on the intended usage, this can include specific foreground user interfaces, control routines, and trajectory generators. The user software should integrate the available system software described above.

3 The Yarc System Software

This section will describe how the AT-Super software has been changed to run in a real-time interactive system with the PC-AT host. It is recommended to anyone who plans to further develop any of these AT-Super programs that they first read the AT-Super manual in Appendix A.

The following is a synopsis of how the AT-Super original software functions. The user creates a program in the C language and compiles it with MetaWare's HiC compiler [3] by issuing the `ccomp` command. It is now ready to be run on the AT-Super. The program is executed by issuing the command **`29k program_name`**. The 29k program itself runs on the PC-AT. This program loads the AT-Super with both its reset boot-strap and the user program given in the command line. The 29k program on the PC-AT then starts up the AT-Super. After start-up, the PC-AT waits to service AT-Super (printf's, scanf's, etc.). Thus, when the AT-Super is busy computing, the PC-AT's processor is basically idle, waiting for the AT-Super to request a service. This situation is undesirable, because the PC-AT can be better utilized.

Through a nondisclosure agreement with YARC, the GRASP lab has obtained the sources to the 29k.exe program. When the 29k program has reached the idle stage, where it waits to service the AT-Super, it is actually running the code in the file `go29k.c`. This file has been altered to call a routine in `\pkg\control\pc`. Now the programmer can have the 29k program do any desired function.

In the case of the TRACS workcell, the desired function is to implement a real-time control system where the AT-Super calculates the control law and the PC-AT provides a real-time interrupt environment (called a background process) and a user interface (called the foreground process). Additionally, both processors must be synchronized to some extent and must be able to exchange information. The control law calculates the voltage output to the robot motors. This can be implementation specific, or the programmer can make use of previously created routines for kinematics, gravity compensation, etc. The background process will collect sensor data, communicate with the AT-Super, and output data to the robot actuators. The foreground process will provide a means for the operator to enter and receive information.

The 29k program altered for TRACS is named `rt.exe`. It is exactly the same as its predecessor, `29k.exe`, up to the point of the PC-AT being an idle servant to the AMD 29000. The makefile for rt.exe is also found in `\pkg\control\pc`. In this directory are three source files: `rt_comm.c`, `foregnd.c`, and `handler.c`. The three paragraphs to follow further describe these sources.

`Rt_comm.c` is the backbone of the real-time system on the PC. It does five tasks in the following order:

1. obtains the address of the shared memory data structure,
2. synchronizes the communication through the mailbox, one of the four bytes of PC-AT I/O space that the AT-Super occupies,
3. calls the routine that installs the real-time environment,
4. calls the foreground process loop, and
5. when an exit request or error is signaled, it calls the routine to shutdown the real-time environment and restores the PC-AT to its normal state.

`Foregnd.c` contains the routines that perform the real-time initialization, the foreground process itself and restoration of the PC-AT. They are called from `rt_comm.c`. Each of these routines will

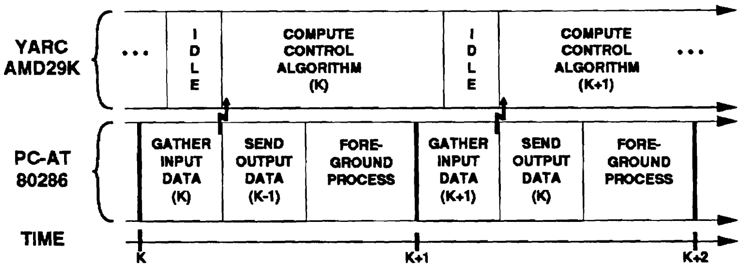


Figure 4: PC-AT and AT-Super Real-Time Process

be functionally described in their order of execution. The `roboinit` subroutine first initializes the digital I/O boards that are used to communicate with the PUMA Unimation controllers. Next the Unimation controllers are initialized and the arm power switch is disabled. A menu of information is printed out for the user. The menu explains what types of keystrokes are acceptable input into the foreground process. Finally, the routine that installs the real-time interrupt on the PC is called, along with an initialization of the interrupt handler (see `init_rt` in Section 4). Then arm power is permitted to be applied by the remote power switch. The `foregnd` routine will now be executed in a continuous loop (from `rt_comm`) until an exit request or error signal are detected. The `foregnd` routine simply processes the user's keyboard inputs and sets the appropriate flags. Usually, the flag signals a section of the real-time code to perform an action, such as setting the `force_exit` flag to stop the real-time environment, or setting the `drive_enable` flag to output control voltages to the actuators. The `roborest` routine disables the robot arm power and then calls the routine that restore the PC-AT interrupt vectors (see `rest_pc` in Section 4). It is called when the system is about to exit due to a request or an error.

`Handler.c` contains the routines required by the real-time interrupt package (see Section 4). These routines are the function definition of the control-break keystroke and the code that is executed at each interrupt. This code reads the robot encoders and sensor data, loads it into shared memory, signals the AT-Super to begin calculations, and outputs the results of the previous AT-Super calculation from shared memory to the robot actuators.

The programs that run on the AT-Super must have a front end that is compatible with the PC's real-time system. In other words, it must be able to synchronize and communicate with the above PC-AT routines. This is done by having a well defined front end that will call a generic user-defined routine. This user defined routine is the actual control calculations, and the front end takes care of real-time issues and data transfers. The front end code is found in the `\pkg\control\29k` directory. It is called `rt_29k.c`. This program zeroes out the shared memory structure, calls any user defined initialization routine, sends the shared memory buffer address to the PC, synchronizes the timing with the PC, and then loops through the upper level of the control routine. This upper level involves waiting for the calculation signal from the PC, retrieving the input data from the shared memory, calling the user defined routine that calculates the control law output, and placing this value into shared memory (see Figure 4). Also in this directory are prototypes for the user defined routines. They are called `user_init`, the routine to initialize any user variables, and `user_calc`, the routine to calculate the control law.

4 The Real-time Interrupt

The software package in `\pkg\intrpt` reconfigures the PC to enable real-time interrupts[1]. It does four basic system changes:

1. ignores the control-c interrupt,
2. changes the interrupt vector for control-break to call a user defined control-break routine,
3. changes the interrupt vector for the timer to call a routine within this package. This package routine can then in turn call the user defined interrupt handler, and
4. changes the frequency of the timer to the user's desired value.

There are also provisions to restore the PC to its original configuration, thus allowing a graceful exit of the program.

There are only a few *hooks* for the user's software to link into this package, and the rest is self-contained. The *hooks* are as follows:

- There are three routines within the **intrpt** package that the user's program must call. They are defined as follows:
  - `extern void init_rt( int tick_freq );` This initializes the PC to reroute the interrupt vector and change the interrupt rate. The `user_tick_handler` is not called when an interrupt (or "tick") occurs *yet*, because it has not been enabled. Note that tick_freq is in hertz.
  - `extern void tick_handler_switch( int on_or_off );` This will enable the routine `user_tick_handler` to be called when a tick occurs if the value of `on_or_off` = 1. If it is zero, then the `user_tick_handler` will not be called.
  - `extern void rest_pc( void );` This will restore the PC interrupt vectors to their original form and reset the system time, if a battery backed-up clock is available.
- There are two routines that must be defined in the user's program:
  - `void user_ctrl_break_handler(void);` This is executed when the user types a "CTRL-BREAK". Usually this routine will set a flag so that the non-real-time (foregound) process can exit gracefully by calling the `rest_pc` routine. Otherwise, should an error occur, the system will need to be reset because the timer interrupts will not have been restored correctly. The graceful exit can easily be implemented by having the main foreground loop check a ctrl-break flag. If it is set by the `user_ctrl_break_handler`, then the foregound exits the loop, processes the `rest_pc`, cleans up anything else and exits.
  - `void user_tick_handler(void);` This routine is executed each time a tick occurs. This is obviously application dependent.

  Note that both of these routines and any subroutines that they call must follow these guidelines:

[1]This package is adapted from software written by F.Fuma and J. Bradley of the GRASP Lab. It is *not* meant to be unique to TRACS and can readily be used on other IBM-PC systems running DOS 3.3.

- the routines must *not* contain any DOS calls (i.e., `printf()`, `scanf()`, etc.).
- the routines must terminate with a `return()` statement, not an `exit()` statement.
- the routines must be compiled with the `-Gs` switch.

Additional application notes as well as a sample or prototype listing of how to use this package is included in Appendix B.

5 Communication Interfaces

This section will describe the communication software interfaces within the system. They are divided into three groups, communications between the PC-AT and AT-Super, between the PC-AT and the Unimation controllers, and between the PC-AT and any sensors.

5.1 PC-AT to AT-Super

The AT-Super provides two forms of access from the PC, either through four bytes of I/O memory or 16k of shared memory (see Appendix A for the AT-Super manual).

The four bytes in I/O space start at 0x0208 in this application. The first byte is the AT-Super control port register. The AT-Super software modified for this workcell does not access this byte, however the unmodified portion does. The next two bytes of the I/O space memory are used to position the base of the 16k window looking into the AT-Super memory. This 16k of shared memory must be continuous but can start anywhere within the AT-Super's memory. The workcell software does make use of this feature as will be shown below. The last byte of I/O space memory is called the mailbox register. One byte of data can be transferred between processors through this mailbox. This feature is used extensively and requires further explanation.

The address of the mailbox for the PC-AT is 0x020B and for the AT-Super is 0x80800000. The PC-AT can read and write to this I/O memory address just as it would any other port in the I/O address space. The `outp()` and `inp()` routines are used for this function (see the Microsoft C Runtime Library Manual [4] for further information). The AT-Super can write to this port directly, however it must be done as a transfer of unsigned long data to the unsigned long address of 0x80800000, even though only the last byte of the unsigned long data is used. The AT-Super's kernel routine will get interrupted each time the PC-AT writes to the mailbox. The kernel interrupt routine will then immediately copy the contents of the mailbox into memory at 0x80000404. Thus, to access data from the PC-AT, the AT-Super's software polls the location 0x80000404 for an unsigned long value.

As previously stated, there is a 16k byte window of shared memory between the PC-AT and AT-Super. The system is configured such that the shared memory is always located at 0xD0000 through 0xD3FFF in the PC-AT memory. However, this memory can be located anywhere within the AT-Super half megabyte of memory. The PC has control over the location of the window by writing the address of the bottom of the window into the second and third bytes of the four byte I/O space memory (see Appendix A). The PC-AT's processor is an INTEL 80286 and it accesses memory least significant byte first, but the AT-Super's processor, an AMD 29000, accesses memory most significant byte first. Thus, even though they have shared memory, they can not transfer data (larger than a byte) without switching the byte position.

In this system implementation, the PC-AT first communicates to the AT-Super through the mailbox. The two processors send commands and data back and forth during the real-time initialization. The most important data the the PC-AT retrieves is the location of the large data buffer on the AT-Super. The PC-AT sets up the shared memory window base to access this data buffer. During the real-time process, the interrupt signal is propagated to the AT-Super through the mailbox (see figure 4). Upon receiving an interrupt, the PC-AT collects sensor data and loads it into shared memory. Then it checks the mailbox for the flag that signifies the AT-Super has completed the previous calculation and is ready for new data. If this flag checks out, the PC-AT sends the command that it may begin to calculate the next servo loop to the AT-Super through the mailbox. Then the PC-AT reads the shared memory for the output data from the previous calculation and outputs them to the robot actuators. This completes the interrupt service routine, and the PC-AT now processes any foreground work before receiving the next real-time interrupt.

The AT-Super has two structures for all data that is transferred between it and the PC-AT. One structure has data that is readily accessible by the user's program on the AT-Super and the other "mirror" structure is designated as the shared memory. The shared memory can be immediately accessed by the PC-AT because the AT-Super does the required byte switching when transferring the data to and from its usable data structure. Thus, when the AT-Super gets the flag to calculate, it retrieves the input data from the shared memory, does the byte switching, and puts it into the usable data structure. Then the user define control algorithm calculates the next set of outputs. These output values are then byte switched as they are loaded into the shared memory data structure from the usable data structure. Finally, the AT-Super writes the flag to the mailbox signally it has completed its calculation and the PC-AT can read the output values from the shared memory.

5.2 PC-AT to Unimation Controller

The Unimation controller consists of six joint processor boards (one for each joint), six analog signal boards, and an interface parallel I/O board. Each joint processor board runs a program that does the following: 1) updates the encoder reading (to 16 bits), 2) outputs power to the actuators (via D/A converter), and 3) can implement a joint-level PID positional control algorithm. In this real-time system, the third feature is disabled while running the AT-Super based control routines. The analog boards provide signal conditioning of the D/A output. The parallel I/O board is designed to communicate with a DEC DRV11 parallel board, a standard LSI-11 communication board.

The PC-AT of this system is equipped with digital I/O boards with have been configured to mimic a DRV11 that would communicate to the controller interface board. The program on the joint processor boards has a well defined set of commands that are described in Unimation's *Breaking Away from VAL* [1] document. A command/data pair can be transferred to or from a single joint, or a command with a six member array of data can be transferred to or from the set of six processor boards. The software on the PC-AT that provides these functions is located in `\pkg\p250\comm`. The routines are called `inj()`, `outj()`, `injv()`, and `outjv()`. The former two transfer data to one board at a time while the latter two transfer arrays (vectors) of six elements, one to each board. The command, data, and robot identification are specified in the calling statement of these routines. Also included in the listing is a facility to reset the Unimation controller.

The commands used by the real-time system most often are reading the encoders, initializing the encoders, and setting the D/A converters. There are other utility programs provided on the PC-AT that are not part of the real-time system and do not use the AT-Super. They are discussed in Appendix C.

5.3 PC-AT to Sensors

The sensor currently being implemented on the system are the Interlink Force Sensing Resistor contact/force sensors. These sensor's resistance varies logarithmically with the surface contact force. The sensor is usually used in a voltage divider configuration, with the voltage being converted to a force value in software. The software for this sensor will be dependent on the analog I/O board used in the PC-AT. At present, the DACA board is used for the A/D conversion. The software for this board is located in `\pkg\daca`.

The Lord Corporation Force Torque Sensor is another sensor that may be used in this system. It is a serial device. Since the PC-AT runs with a dedicated real-time interrupt when controlling the robots, the normal serial communication routines are not compatible. John Bradley has developed a serial device-driver for the PC within the GRASP lab which will implement the serial communication functions. This software is located in `\pkg\serial`. The driver is operated by polling, however, and additional development will have to be done to make it interrupt driven. The sensor has commands for zeroing itself and returning the force-torque values. Test programs to do this are located in the `\pkg\sensor` directory. At the time of this writing, the sensor has not yet been used on the robots. Future plans are to use this sensor, or a similar one, to measure the forces and torques being generated at the wrist of the robot.

6 Creating Application Software

This section will describe how to develop control software for the system, what features are available, and how to compile and link the source.

6.1 Development

The system design is such that the PC-AT and the AT-Super both run concurrent, synchronized processes. The PC-AT loads the AT-Super with its executable file, starts it up, and then synchronizes the communication between the two processors through the AT-Super mailbox (see Section 5). The program running on the AT-Super initializes variables and waits for the first real-time signal from the PC-AT. When the signal comes, the AT-Super call the user-defined control routine after doing some background bookkeeping. This section will describe how to develop the control software to fit within this scheme.

The system calls two user defined routines, `init_29k.c` and `user_29k.c`. Prototypes of these two programs exist in the directory `\pkg\control\29k`. Actual working models of these programs can be found in most of the `\pkg\rt` sub-directories. The `init_29k.c` routine is supplied to the user in order to initialize any variables within the user's code, before any control calculations are made. It is called once at the beginning of execution. The `user_29k.c` routine is called on each real-time interrupt. This is where the control law, trajectory, and any other real-time processes are calculated. These can all be newly written, or the user can make use of previously developed control routines in the `rt` directory. This routine also has an error code for its return argument. If this argument is non-zero, the robots' arm power will be disabled and the real-time system will be exited back to MS-DOS.

There are three important structures within this system defined in `\pkg\control\h\tracs.h`. These structures are declared global within the main routine from which `init_29k` and `user_29k` are called. They are listed and described below:

- `extern struct how pc_data[2];` This structure contains all the information from the PC-AT for that particular interrupt period concerning one of the robots (hence its dimension of two). Included in this structure are the encoder values, the sensor values, and the task phase values. The task phase values are the inputs from the operator through the PC-AT foreground process. Through this vehicle, the operator can have the control code segue between separate task phases of the program.

  `extern struct chg yarc_data[2];` This structure contains all the information from the AT-Super to the PC-AT resulting from the calculations of the interrupt period for one of the robots (again, note the dimension of two). Included in this structure are the values to be outputted on the actuator DAC's, the status of the controller (which could also be used as D/A outputs), and a special code that enables data logging for debugging and graphing purposes (see Section 6.2 below).

  `extern struct log yarc_log[2];` This structure contains the data to be logged from this interrupt period for one of the robots. It contains up to six values, which are user defined in the controller (see Section 6.2 below).

Put quite simply, it is the software developer's responsibility to write the body of the `user_29k` routine and within that body one must access the variables from the **how** structure, calculates the control law and outputs the results into the **chg** structure.

6.2 Development features

This section will highlight the data logger and other code available within the system that a developer may consider incorporating.

The data logger is a feature that allows the user to obtain real-time data from the system. It is easily implemented and can be tailor to suit many needs. The logger allows up to six variables per interrupt for each robot to be stored. The logger is begun on the AT-Super by setting the `logcode` of the **chg** structure to the value `LOG_ON` (defined in `tracs.h`) and filling up the `info` array of the **log** structure with the variables of interest. The PC-AT checks the logger code word and when it is set, the PC-AT stores the values from the `info` array into two (one for each robot) large arrays (1000 by 6) within the PC's program. Note that the PC's array is 1000 deep, so only 1000 sets of variables can be logged. The user program (on the AT-Super) must keep track of how many sets of variables have been sent to the PC-AT, it cannot exceed 1000. The AT-Super program stops the logging by setting the `logcode` of the **chg** structure back to zero. When the program is finished, the PC-AT writes the logged data to a file the directory `\pkg\rt\log`. The name of the file reflects the time and robot of origin. This directory should be emptied of old, unwanted log files, or else the system disk will quickly fill up. Note the flexibility of this feature, the user program decides when to start and stop logging data. It could be triggered by any event: the start of a new task phase, the saturation of a sensor or actuator, the reaching of a certain position, etc.

The software developer does not have to start writing code from scratch. There are many working files in various `\rt` subdirectories. The most important resources are mentioned here. The `\rt\lib` subdirectory has robot library routines, most notably the kinematics and inverse kinematics for each 6 DOF arm. Most applications use this library. This library also adopts a practical convention for storing transformations (that is prevalent throughout the lab). The files `pumadata.h`, `pumaforce.h`, and `mmcs.h` located in the `\pkg\control\h` subdirectory may also prove useful in developing code.

Additionally, it is common for the basic position control routines to have a joint level proportional-derivative (PD) controller embedded in `user_29k`. This can be used as is, but note that the routine reads gains from an external file named `\pkg\control\data\gains.dat`. (There is a variation of this file used for force control with the flat end-effectors.) The developer may wish to adopt the overall style from the applications, especially the way that various phases of the program task are enabled by the operator. The phases are also generally used as trajectory generators.

When making changes to existing code, do so judiciously. Remember, that the robots are being *directly* controlled by the code you are writing. Use safeguards as much as possible, such as software torque limits, error checking, software return status, etc. Make large, sweeping changes incrementally, verifying each step by running the program. When running new code, be very careful and observant of the system behavior (i.e., have your finger ready on the arm power off button.)

6.3 Compiling and linking

The programs are compiled with the MetaWare HiC compiler, invoked by the `ccomp` command [3]. The compiler flag `-f027` is commonly used to signal the compiler that the floating point coprocessor, the AMD 29027, is to be used. Also, the trigonometric functions are contained in a high speed library, `\yarc\hc\lib\ftrig.o` that should be linked. The code written by the developer resides in `user_29k.c`, `init_29k.c`, or is located in a subroutine that is called by either of these routines. The `main()` routine, however, is standard for all programs and is located in `\pkg\control\29k\rt_29k.o`. Thus, this is the first routine named in the compiling statement, such as:

```
ccomp -o filename.out -f027 \pkg\control\29k\rt_29k.o filename.c
\pkg\rt\lib\kinlib.o \yarc\hc\lib\ftrig.o
```

where `filename.c` contains all the user developed routines, including `user_29k.c` and `init_29k.c`.

The applications on the system make extensive use of the `MAKE` utility. It processes a "makefile" that has a record of what flags should be used and which routines and libraries should be compiled and linked. Additionally, the `MAKE` utility will check to see if the object code is up-to-date and if it is not, it will recompile the newer source files. (See the MicroSoft Code View and Utilities Manual [2] for more information.)

7 Running Application Software

This section will outline how to run programs on the system and it also contains a subsection which describes how to use the offline control code debugger.

Before any program can physically use the robots, they must be powered up and calibrated. (See the calibration section of Appendix C). Occasionally, the robots will lockup and not move at all when they are enabled. This is because the controller joint boards have unknowingly reset themselves. In this case the robots must be re-calibrated.

A program, whose name is `filename.out` is run by issuing the command `rt filename`. The program `rt` is the modified version of the AT-Super loader/servicer that creates the real-time environment and runs on the PC-AT. The program `filename.out` is created by the user to run on the AT-Super and contains the system's `rt_29k` code as well as the user defined `user_29k` code, the latter of which does the robot control. The robot arm power should be disabled before executing this command.

When this program starts, it will print out information that may or may not have significance to the operator, depending on how the `init_29k` routine is written. On many applications, this program will read data files (such as the PD gains file) and then print them on the screen to confirm that the correct data is being used. This type of output is only done in the initialization, because the real-time code cannot do printf's from the AT-Super. Eventually, the program asks if the operator wishes to change the sampling rate. Type "y" if this is desired, otherwise just hit any other key. It is suggested that the sampling rate not be changed unless the code is so long that it is unable to run at the default rate.

Next, the main menu is displayed. Read it carefully. The menu summarizes the acceptable inputs to the system (note they are all lower case) and the basic inputs are described below:

- **q** — This will command terminate (quit) the program and restore the system to MS-DOS. If the robots arm power is on, it will be turned off, however it is recommended that the arm power be turned off by the remote switch (or at the controller front panel) before issuing this command.

- **b** — This command will begin the passing of control loop actuator values to the actuators themselves. In other words, before this command is issued, the actuators are receiving zero power and the arm is actually limp. Turning the arm power on now will cause the arms to collapse under their own weight since the brakes will be off but no current is sent to the motors. If the arm power is turned on before this command is issued by mistake, exit the program and start over. Do **NOT** try to run the code now because the the trajectory generator may have been initialized at the robots configuration when the program started, and now the robots position is very different. If **b** is issued, there may very well be a quick, unexpected movement by the arms.

- **e** — This command will end the passing of control loop output values to the actuators and pass zero instead. Note that the robots will be limp, their brakes are off and no power is going to the motors. (See above.) This is used very infrequently, when it is desirable to limp and reposition the robots usually *after* a program is run.

- **p** — This command will enter a user defined phase of the task. This command has two additional keystrokes associated with it. The first key, immediately after the **p** discriminates between which robot should move to this phase, given by:

  - **0** — Robot number zero.
  - **1** — Robot number one.
  - **2** — Both robots zero and one.

  The next keystroke identifies which phase, from 0 to 5. Thus the command **p 2 0** will cause both robots to enter phase zero of the task.

The above commands control the software execution, but the operator has final say because s/he controls the arm power switch. This switch should *always* be in the operator's hand. The normal sequence for running a program is:

1. Verify arm power is off.

2. Type in the `rt filename` command.

3. Use the default servo rate (by hitting any key *but* "y" when asked if a change is desired.)
4. Type the **b** key at the menu prompt.
5. Turn the arm power on, closely monitor the robots, and if any unexpected behavior results, quickly turn off the arm power.
6. Run through the phases of the task using the **p** command above. Again, closely monitor the robots, and if any unexpected behavior results, quickly turn off the arm power.
7. Turn arm power off when task is complete.
8. Type the **q** key to quit the program.

After the arm power has been turned off, in steps 5, 6, or 7, do not turn it back on while the servo loop output values are being passed to the actuators. The program must be exited and the above sequence re-started to run it again.

Note that if the system severely malfunctioning, no keyboard entry may be possible, in which case, either hit `CTRL-BREAK` or reboot the PC. If the program displays the menu then exits immediately, the real-time synchronization is usually at fault. It is very likely the AT-Super was still calculating the previous servo period's output when that PC-AT expected it to be done. This could result from having an error in the user defined code or if the code is too lengthy and can not be calculated in time. The next section describes the off-line control debugger and timing utilities within the system that may be of use in this event.

7.1 Debugger

An off-line control debugger is located in `\pkg\rt\debug`. This debugger has two functions: it can time your `user_29k` routine or it can step through you control calculations, one servo period at a time. The indicated directory contains a program named `test.c` which acts like the `rt_29k` routine in TRACS, in that it contains the main routine and calls the user defined control routine. However, the PC is not in real-time mode while debugging, so the `rt` program is not used to run this software. Instead the `29k` program, the original AT-Super loader/servicer is used.

The `user_29k` routine is linked to the `test` routine by using the make files supplied. Note that you can link the `user_29k` object file directly, it does not need to be recompiled. The debugger is started by typing `29k filename`. The program gives you the opportunity to change any of the variables in the input structures, the normal inputs from the PC-AT such as robot encoders, sensor values, phase of task execution, etc. It will ask if you wish to time the routine or step through it.

The timer will query you for the number of trial iterations. It will then check the clock, call the routine the desired number of times, and then check the clock again. Thus, the time expended on the clock divided by the number of iterations gives the approximate time of calculation of the servo loop.

The stepper will process one servo loop and display the results from the output structure such as the power to the actuators. Then you will be able to change the input structure before the next step.

With these tools, you can sort through problems within the servo loop code. Note that write statements can be added to the servo code now since it is not in real-time mode, but then the routine will have to be re-compiled when switching between real-time and debugging modes.

8 Advanced Topic: Adding New Hardware

As the system evolves over time, the facilities to incorporate more sensors will be added to the system. The additional sensors may lead to new analog boards (both analog-to-digital (A/D) and digital-to-analog (D/A)) and communication boards (to a vision system, etc.). This section will describe what is involved to perform the integration from the TRACS software standpoint and give an example. This section does not provide any information that deals directly with the operation of a specific board. (It is the responsibility of the user to develop this type of software.)

The basic philosophy of sensor integration into TRACS is to have the PC-AT gather sensory input and to pass it to the AT-Super through shared memory as well as take output values from the AT-Super (via shared memory) to the sensors and actuators. The PC-AT must do all this during the interrupt servicing routine.

In the communication board integration case, this may pose a problem since serial communications that are done through polling at reasonable speeds require a polling frequency far in excess of the interrupt frequency. An interrupt driven communications board may be an option, but then the PC-AT will have multiple interrupts and becomes more complex.

The structure of the shared memory is critical in the applications. Without changing this structure, the system can have up to six sensory inputs and six outputs per robot. The structure can be modified to accommodate more and this will be shown in the example below.

8.1 Example: Adding a New Analog Board

This example will go through the steps involved in adding a new analog board to the system. Currently, the DACA board is used for analog inputs and outputs. It has two D/A's and 4 A/D's. For this example, let's assume the requirements are for 10 A/D channels of input and 3 D/A channels of output per robot.

The first step is to write the low level code that communicates with the board, which is beyond the scope of this document and is hardware dependent. There will probably be three routines, one that initializes the board, one to write to the D/A's and one to read the A/D's. The first routine, initialization, can be done before the real-time interrupt is started, but the latter two must be called from within the interrupt service routine. Therefore, these routines must follow the guidelines of the real-time interrupt code outlined in Section 4.

The 10 A/D channels per robot requirement is beyond the present system structure's capabilities, therefore the structure will have to be changed. This is not a terribly difficult task, but it will cause changes system wide. Most notably, any software running under the old structure will have to be recompiled and relinked with the new structure. But, since the MAKE utility has been used throughout the system, updating the old programs will not cause serious problems.

Currently, the main structures of TRACS are found in `\pkg\control\h\tracs.h` and is reprinted below:

```
/*
/* tracs.h
/*
 */

#ifndef TRACS_STRUCTS

#define NJNTS  6
```

```
#define ADCHANS 6
#define SUBTASKS 6

/* Data logger codes */
#define LOG_ON 1

struct how { /* Information from PC to 29K */

short enc[NJNTS], /* Encoder values */
adcr[ADCHANS], /* A/D converter values */
phase[SUBTASKS]; /* Phase of task (user defined) */
};

struct chg { /* Information from 29K to PC */

short dac[NJNTS], /* Motor DAC value */
status[SUBTASKS], /* Control status (user def.ed) */
log_code; /* Data logger code-word */
};

struct log { /* Information from 29K to PC */

float info[6]; /* data logger info (user-def)  */
};

#endif
#define TRACS_STRUCTS
```

In the above file, the **adcr** array of the **how** structure passes the A/D values to the AT-Super and the **status** array of the **chg** structure can be used to pass D/A values from the AT-Super. Both of these arrays have dimension six, while this is adequate for the D/A requirement, the **adcr** array dimension will have to be increased to ten. This is easily realized by the change:

```
#define ADCHANS 10
```

The next step is to run MAKE in the `\pkg\control\pc` and the `\pkg\control\29k` directories to make the new `rt` and `rt_29k` programs, respectively. The `rt` program is run from the DOS prompt and the `rt_29k` routine is the AT-Super real-time front-end that must be linked to all AT-Super control software that uses this new version of `rt`. This linking of the control software should be easily accomplished by running their respective MAKE routines. Thus, the needed changes are instituted.

Future changes could involve more than just widening an array, such as adding new variables to the above structures or adding new structures themselves. In the former case, adding new variables to the existing structures, the modification can be made to this include file and above procedure will still work as long as the variables are of the same type. In other words, the **chg** structure can only contain shorts and the **log** structure can only contain floats. In the latter case, adding an entirely new structure, the modification is more involved and should be done only it if no other solution exists. In this case, the program `\pkg\control\29k\rt_29k` will have to be well understood and editted. This is the AT-Super real-time front end. (See Section 5). In particular, this code performs the following:

- Allocate memory on the AT-Super for a mirror of the above structures because the AT-Super and the PC-AT data is not stored the same way and the data must be byte swapped between the two processors. The mirror structure is used by the AT-Super and the original shared memory structure is used by the PC-AT. The size of the allocated memory depends on the number and size of the structures.

- Initially zero out all the mirror structure data.

- Perform the actual byte swapping in exchanges to and from the mirror structure and the original structure. The byte swapping is different for shorts and floats, and that is why they are segregated between different structures. The exchanges and byte swapping is all done with pointers, which is why each structure must be homogeneous in data type.

References

[1] *Breaking Away From VAL.* Unimation Inc., 1982.

[2] *Code View and Utilities.* Microsoft Corporation, 1987.

[3] *High C Programmer's Guide.* MetaWare, Inc., 1989.

[4] *Microsoft C Run-Time Library Reference.* Microsoft Corporation, 1987.

YARC Systems Corporation

AT-Super User's Manual

Revision 1.3

TABLE OF CONTENTS

1. INTRODUCTION 1

1.1 Theory of Operation 1
1.2 Hardware Features 1
1.3 Software Features 3

2. INSTALLATION 5

2.1 Caution 5
2.2 Software Installation 5
2.2.1 System Software Installation 5
2.3 C Compiler Installation 6
2.4 FORTRAN Installation 7
2.5 Configuring 29K.EXE 8
2.6 Hardware Installation 10
2.6.1 Physical Layout 10
2.6.2 Physical Installation 10

3. SOFTWARE OPERATION 12

3.1 29K.EXE Program 12
3.1.1 29K.EXE Program Options: 12
3.2 C Compiler Operation 13
3.3 FORTRAN Compiler Operation 14
3.4 Compiler Batch Files 16
3.5 Debugger 16
3.5.1 Running The Debugger 16
3.5.2 Debugger Commands 17
3.5.3 Addresses 17

4. INTERFACE 18

4.1 Host Interface (HIF) Specification 18
4.2 AT-Super Memory Map 19
4.3 Control of the AT-Super 19
4.3.1 Control Port Register 20
4.3.2 Address Registers (Window Control) 20
4.3.3 Mail Box Register 21
4.4 Expansion Connector 22

APPENDIX 1: DIPSWITCH TABLES 23

1. INTRODUCTION

1.1 Theory of Operation

YARC Systems Corporation develops coprocessor products, such as the AT-Super. These computational engines provide mainframe level computing power using familiar host platforms/environments such as the IBM AT running MSDOS.

The coprocessor, in this case an Advanced Micro Devices Am29000, provides several times the computing power available from the Host system. The AT-Super utilizes the Modified Harvard Architecture inplemented by the Am29000 processor to provide separate Instruction and Data Memory. This separation allows the greatest performance in a coprocessing system.

To accomplish this, YARC provides coprocessors with sophistacated memory systems using the latest in memory technology. This memory is a local resource attached to the bus of the coprocessor and is therefore refered to as tightly coupled memory. This configuration allows maximum compute power while running in parallel to other processors in the system. The other processors can be either the Host processor, in this case an 80286, or additional coprocessor boards.

Each coprocessor communicates to the Host system through both I/O Ports and Memory Pages located in the dedicated page frame for the board. In this sense, the board looks much like an Expanded Memory Board to a PC. The Host (AT) loads a program to be run into the memory of the coprocessor board and issues a command to begin execution. Software executing on the coprocessor board will then notify the Host of any services it wishes to have provided by the Host. The Host system provides I/O support for the coprocessor, including file, console and printer I/O.

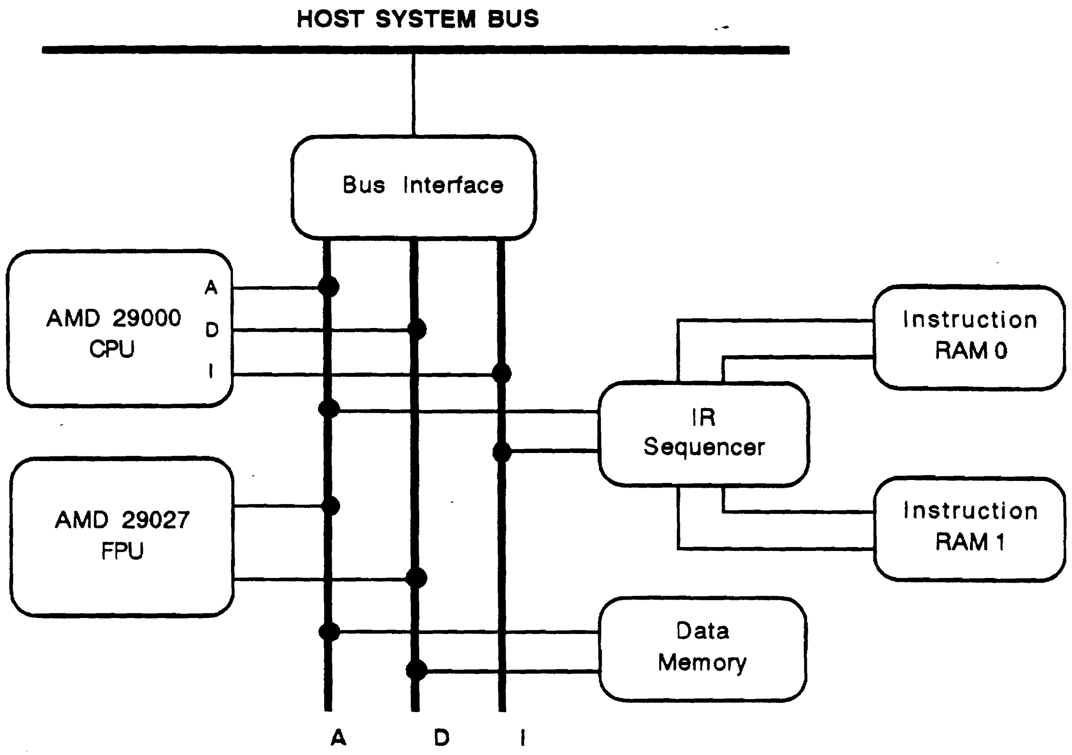
The AT-Super coprocessor board is capable of using Direct Memory Access (DMA) to operate as the master of the system. In this mode it can share slot memory and devices with the host and other coprocessors in the system.

1.2 Hardware Features

The AT-Super features an Advanced Micro Devices Am29000 processor with a socket for an optional Am29027 Arithmetic Accelerator. In addition, there are 2 Megabytes of Instruction Memory and 2 - 4 Megabytes of Data Memory.

The basic hardware diagram for the 29K boards is shown below. The card has three internal buses (Address, Data, and Instruction) in a configuration known as a Modified

Harvard Architecture. For a detailed description of this architecture, please refer to Trevor Marshall's article in Byte magazine (May, 1988).



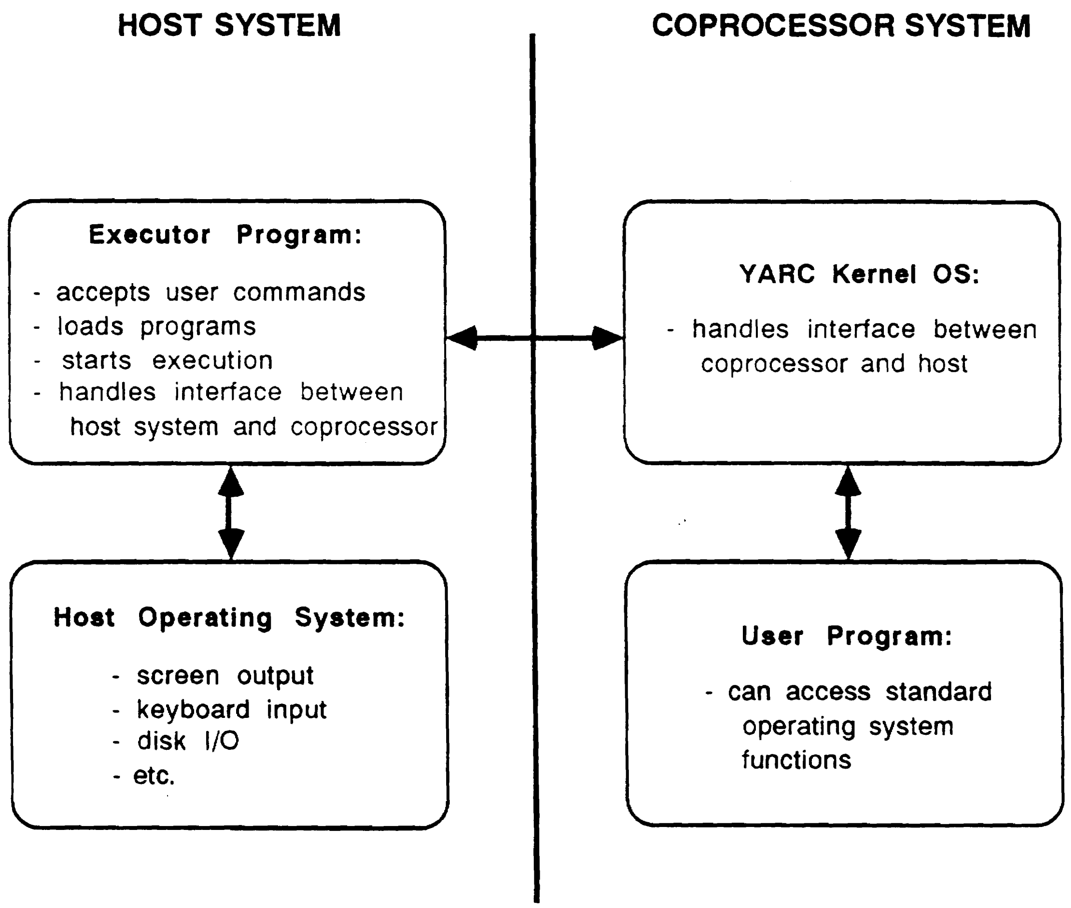
This architecture allows the CPU to perform simultaneous instruction and data fetches. The CPU first puts out the address of the instruction that it needs to fetch. This address is latched by the address control logic. During the time needed for the instruction to become valid on the instruction bus, the CPU can be performing a data fetch.

An additional design feature of the board which allows it to operate at high execution speeds is the interleaving of the instruction RAM. The instruction RAM occupies two different RAM banks which are interleaved so that one address points to one bank and the next sequential address points at the other bank. When the CPU fetches an instruction, the next instruction is also fetched. Instructions can therefore be fed into the CPU at a 40ns /cycle while using 80ns dynamic RAM. The memory architecture of the AT-Super allows it to acheive sustained execution rates of 17 MIPS, with 25 MIPS during burst mode.

The AT-Super may be expanded with daughter cards, which are plugged into the AMD 29027 socket, and then the 29027 is plugged into the daughter card. Daughter cards can be used for specialized operations such as driving printer engines or for data acquisition. YARC provides information for the design of daughter cards by the user. Additionally, YARC can be contracted to design daughter cards for custom applications.

1.3 Software Features

The design goal for the AT-Super product line is to provide the highest performance while minimizing the additional programming effort involved. Programs running on the coprocessor board have full access to standard host I/O functions. This is accomplished through an executor/loader program which translates requests for I/O operations from the board into the host-specific calls. A program running on the board does not "know" that it is running on a coprocessor board. The program can make all of the standard C or Fortran file or character I/O calls and these will be executed by the host. A diagram of the host/board software interaction is given below.



The user interacts with the AT-Super through a number of simple batch commands. For example, typing

```
        29K my_program.out
```

in response to the DOS prompt will cause the executable file "my_program.out" to be loaded onto the AT-Super and begin execution. From that point on, the program appears exactly as a DOS program, only executing many times faster. With multi-tasking operating systems, such as Microsoft Windows or DESQView, the user can operate the board as a background task.

Multiple coprocessor boards can be used for increased performance. While the boards are not specifically designed for parallel processing, they can be very useful for concurrent processing. For example, different stages of an image processing operation could be running on different boards, with the boards communicating through their bus-mastering capabilities. Likewise, multiple versions of a program can be run for increased throughput (e.g. different frames of a ray-traced animation could be performed on different boards).

2. INSTALLATION

The installation procedure for the AT-Super is described in this section. You should first install the software onto your development system. Next you should run the configuration program, which will configure your execution environment and inform you of the proper switch settings for the board. Finally, you should physically install the board into your PC system and run the test programs.

2.1 Caution

The AT-Super is a high-performance system and contains a number of static-sensitive devices. Ensure that you have discharged the static from yourself prior to handling the board. The proper way to do this is to touch the metal frame of your PC to ground yourself, and then touch the bag the board is shipped in (prior to opening it). This will discharge any static build-up on your person or on the board. Whenever you are going to handle the board, always ground yourself on your PC frame first.

2.2 Software Installation

Prior to installing the software that came with your system, it is suggested that you make working copies of all diskettes. Use these copies for the instatllation of your software, and keep the originals in a secure place as backups. You should first install the system software contained on the diskette labeled "Executor/Debugger". Next you should install the appropriate compiler software (if purchased).

2.2.1 System Software Installation

The "Executor/Debugger" diskette contains the following files and directories:

File	Function
YARCBOOT	29000 file which boots the AT-Super
29K.EXE	Executor program
CONFIG29.EXE	program to configure the system software
DEBUG29.EXE	debugger program
DEBUG29.DOC	notes on the debugger
README	a file with notes not in this manual
BROWSE.COM	a program to look at text files
README.BAT	displays "readme" file
DEMO\OSTEST.BIN	test of operating system
DEMO\DHRY.BIN	Dhrystones benchmark
DEMO\PI.BIN	program to calculate PI

DEMO\OPT1WHET.OUT	optimized Whetstone benchmark (single prec.)
DEMO\OPT2WHET.OUT	optimized Whetstone benchmark (double prec)
DEMO\WHET.OUT	standard Whetstone benchmark
DEMO\WHET.DOC	Whetstone documentation

Several directories are required to hold the various compiler and executor tools which you will be using with the AT-Super. The simplest installation method is to use the directories "**C:\YARC\bin**" ,"**C:\YARC\lib**", and "**C:\YARC\include**". Create these directories using the following commands:

```
MKDIR C:\YARC\bin
MKDIR C:\YARC\lib
MKDIR C:\YARC\include
```

All of the *.EXE files and the YARCBOOT file off of your system diskette should be placed in the "C:\YARC\bin" directory. The additional files on the diskette may be placed in any directory.

You should then modify you AUTOEXEC.BAT file so that the DOS system knows where to search for the various files that you have installed. First your PATH variable should include a reference to the "C:\YARC\bin" directory so that the .EXE files can be executed. As an example, the line should read:

```
SET PATH = C:\YARC\bin
```

You should also add a new system variable to your AUTOEXEC.BAT file which reads

```
SET YARC29K = C:\YARC
```

Which will direct YARC's system software to the correct location of the YARCBOOT file.

2.3 C Compiler Installation

All of the files needed for MetaWare High C on the AT-Super are contained on three diskettes. The installation procedure for the various files is given below.

Place the C Compiler Disk 1 in your "A:" drive. Copy "**A:ccomp.exe**" into a directory that is in your MS-DOS search path. Now copy the binary tools and libraries by typing:

```
COPY A:\bin\*.* C:\YARC\bin
COPY A:\include\*.* C:\YARC\include
```

Now place the C Compiler Disk 2 in your "A:" drive and type:

```
COPY A:\bin\*.* C:\YARC\bin
```

Now place the C Compiler Disk 3 in your "A:" drive and type:

COPY A:\lib\*.* C:\YARC\lib

This should have copied the following files to your destination disk:

C:\YARC\bin\HC1COM
C:\YARC\bin\HC2COM
C:\YARC\bin\LD29K
C:\YARC\lib\LIBS.A*
C:\YARC\lib\CRT0.O
C:\YARC\include\*.H

If you do not want to use the "**\bin**" and "**\lib**" directory names, **set 29KBIN** =(your binary directory) and **set 29KLIB**=(your library directory) and the compiler will search these directories if it cannot find files in the "**YARC29K**" sub-directories.

2.4 FORTRAN Installation

All of the files needed for Topexpress Fortran on the AT-Super are contained on a single diskette. The installation procedure for the various files is given below. Place the files in the directories shown below;

Binary files: (in "C:\YARC\bin")

FCOMP.EXE
F77PASS1.OUT
F77PASS2.OUT
LD29K
YARCBOOT

Library files: (in "**C:\YARC\lib**")

CIO.O
CRT0.O
F77LIB.O

If you do not want to use the "**\bin**" and "**\lib**" directory naming convention, you can place the binaries and libraries anywhere on your hard disk that you like. You must, however, set up several environmental variables so that the compiler can find them. These environmental variables used are **29KBIN** and **29KLIB**. The compiler will not append "**\bin**" or "**\lib**" onto a path used in this way; you must describe the exact path to your private directory. If the compiler cannot find files in the "**YARC29K**" directories:

- **F77PASS1.OUT**, **F77PASS2.OUT**, **LD29K**, and **YARCBOOT** are all searched for in the directory pointed to by "**29KBIN**".

- **CIO.O**, **CRT0.O**, and **F77LIB.O** are all searched for in the directory pointed to by "**29KLIB**".

2.5 Configuring 29K.EXE

The boards are shipped in a standard configuration, and there is often no need to change the settings. Special operations, however, may require modifications. The most common setting that is of interest is the one that sets the board's location in the I/O space of the PC. The board is shipped with its 16K window set at hex D0000. It may be necessary to move this window so as not to conflict with other boards, particularly if you will be using more than one AT-Super. Some of the settings can cause severe problems for the board, so do not modify any of the settings that you are not familar with.

The configurator is invoked by typing

```
config29 <cr>
```

The configurator reads in the "29k.exe" file and determines it's operating parameters. These may then be changed.

When the configurator is invoked the following begins scrolling onto the screen. The configurator first displays all of the current settings for the "29k.exe" file, and then one-by-one asks if you want to change these settings. For each option the one displayed in square brackets [ ] is the one currently selected. Pressing the enter key without typing an option will leave the currently selectec choice intact. After you have modified the settings in the "29k.exe" program, the configurator will display a diagram of the switch blocks on the board with the appropriate settings. Please verify on your board that the switch settings match the displayed settings on your screen.

The output from the "config29" program is shown below:

```
YARC 29000 Configurator V0.50. (C) 1989 Yarc Systems Corporation
Changes will be made to file 29k.exe
The executor will look for the YARC 29000 board at hex D0000.
I/O port base is set to hex 208.
DMA channel is set to 0.
Not using interrupts.
Debug off.
May become Bus Master.
Ignore HIF Notify calls.
Not using host interrupts.
Reset 29000 after each program ends.
Won't show each HIF call.
Memory not zeroed before loading each program.
```

```
Will use 29027 FPU if available.
29027 Mode = 0FC00000, Counters = 00000364.
Will use Branch Target Cache if 29000 Revision CA or greater.
Will show sign on message.
DW bit in 29000 configuration register OFF.

Search for 29000 memory at hex? [D0000]
Set I/O port base to hex? ([208],228,240)
Use Direct Memory Access? [N] Use Interrupts? [N]
Show sign on message? [Y]
Change debug options? [N]
Allow 29000 to become Bus Master? [Y]
"Break" on HIF Notify calls? [N]
Allow use of PC interrupts? [N]
Reset 29000 after each program ends? [Y]
Show each HIF call? [N]
Zero memory before loading each program? [N]
Use 29027 FPU if available? [Y]
Set 29027 Mode (high word) to hex? [0FC00000]
Set 29027 Counter (low word) to hex? [00000364]
Branch Target Cache (A) on, (B) off, (C) on for rev CA or greater?
[C]
Turn DW bit on in processor configuration register for rev D and
greater? [N]
```

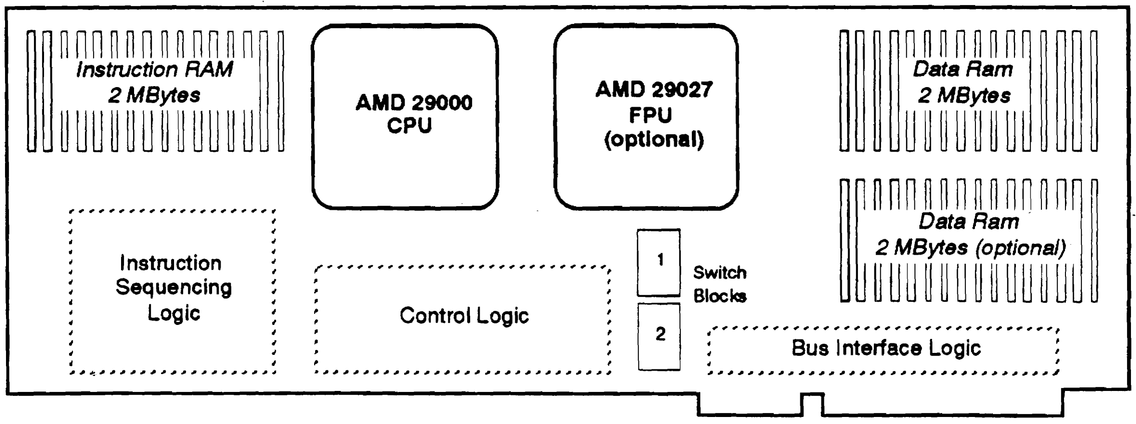
```
OFF  __________  ON   Switch Block 1
   1_____     _       Control Port Address
   2_____     _       Control Port Address
   3_____     _       16 Bit Transfers if On
   4_____     _
   5_____     _       Interrupt 10, Enabled if On
   6_____     _       Interrupt 11
   7_____     _       Interrupt 12
   8_____     _       Interrupt 15

OFF  __________  ON   Switch Block 2
   1_____     _       DMA Request 7, Enabled if On
   2_____     _       DMA Request 6
   3_____     _       DMA Request 5
   4_____     _       DMA Request 0
   5_____     _       DMA Acknowledge 7, Enabled if On
   6_____     _       DMA Acknowledge 6
   7_____     _       DMA Acknowledge 5
   8_____     _       DMA Acknowledge 0
```

2.6 Hardware Installation

2.6.1 Physical Layout

The upper left section of the board is dedicated to the Instruction Memory. The upper center contains the Am29000 and optional Am29027. The upper right contains the Data

Memory. The lower sections of the board contain the interface to the Host system. The physical layout is shown below:



2.6.2 Physical Installation

1. Turn OFF the power to your computer. Make sure no power goes to your system.
2. Remove the sheet metal screws on the back of your unit that hold the cover in place. Slide the cover forward being careful not to disturb any of the cables or wires in the system. Set the cover aside. Ground yourself on the frame of the PC.
3. Choose an open 16-bit expansion slot on the motherboard. A 16-bit expansion slot has two sets of edge connector slots. Make sure that adjacent boards do not interfere with the board or air flow to the board.
4. Remove the metal plate at the back of the computer that corresponds to that slot. To do this you will have to remove the retaining screw and pull up.
5. Carefully slide the AT-Super card down the card guides all the way into the expansion slot. Be sure the board is seated well into the slot. The board should not appear uneven, unbalanced, cocked or bent in any way.
6. Replace the retaining screw into the AT-Super plate.
7. Carefully replace the system cover, making sure that no cables or wires are disturbed.
8. Reinstall the retaining screws at the back of the cover.

9. Reconnect power and boot your computer with DOS.

3. SOFTWARE OPERATION

3.1 29K.EXE Program

The Loader/Executor program is the host-resident software that places software to be executed into the memory of the AT-Super and provides I/O support for programs running on the AT-Super.

The 29K.EXE Loader/Executor reads Am29000 binary executable files, loads them into the memory of the AT-Super and signals the AT-Super to begin execution. The AT-Super may require services to be performed by the Host processor, the AT. These services are provided by the Executor portion of 29K.EXE.

Most of the options for the Loader/Executor are correctly set by the file 29K.EXE. This allows the user to have the remainder of the command line to specify the program to execute and the options that should be passed to the AT-Super program.

The general form of the command line is as follows:

```
        29K {-loader_option} execution_file {-execution_file_options}
```

Only the 29000 program name (execution_file) and the options to be passed to it are required. All other options are either taken from the 29K.EXE file or simply default. As an example, to load an executable file "my_prog.out" down to the board and to initiate its operation, enter

```
      29k  my_prog.out
```

in response to the DOS prompt.

3.1.1 29K.EXE Program Options:

-d Turn on debug mode. Debug mode prints messages as to how far the program and loader are progressing. It is normally not used.

-v Turn on Verbose mode which prints messages regarding the size and location of what is loaded.

-n027 Suppress the board from using its Am 29027 FPU.

3.2 C Compiler Operation

The MetaWare High C compiler is invoked with the command

```
ccomp my_prog.c
```

This will invoke both passes of the C compiler, then link the file and produce the executable file "my_prog.out". There are a number of options available to the user for compilation. Some of the most common of these are shown with examples below. User input is shown in bold. File names should follow the standard C language convention of being less than 32 characters long. There should be no spaces or colons in the file name.

File extensions should be:

.c	C program source listing
.s	Assembly code source listing
.o	Object file
.out	Executable file

For a complete listing of compiler/linker options, please consult the MetaWare C Compiler manual.

Multiple Compilation

ccomp file_1.c file_2.c file_3.c

Result - compiles all three files, links them and creates executable file file_1.out

Suppressed Linking

ccomp file_name.c -c

Result - compiles the program file_name.c, and generates the object file file_name.o

Multiple Linking

ccomp file_1.o file_2.o file_3.o

Result - links the three object files and creates an executable file called file_1.out

Compilation and Linking

ccomp file_1.c file_2.o file_3.o

Result - compiles file_1.c, and then links all three object files and creates an executable file called file_1.out

Output Direction

ccomp file_old.c -o file_new.out

Result - compiles file_old.c and names the executable file file_new.out

YARC Floating Point Routines

ccomp file_name.c -f027 ftrig.o

Result - compiles file_name.c and replaces the standard single precision C routines with YARC's optimized routines

ccomp file_name.c -f027 dtrig.o

Result - compiles file_name.c and replaces the standard double precision C routines with YARC's optimized routines (both single and double object files can be specified

Link Information

ccomp file_name.c -m

Result - produces a section map of the program using Am29000 addressing

ccomp file_name.c -i

Result - produces an expanded section map of the program using Am29000 addressing

ccomp file_name.c -t

Result - produces a symbol map for each symbol in the program

ccomp file_name.c -a

Result - produces all section and symbol data (same as using both -i and -t options)

3.3 FORTRAN Compiler Operation

The FORTRAN Compiler is operated directly from the DOS command line. As an example, to compile the Fortran program "my_prog.f", the user would type:

```
        fcomp  my_prog.f
```

in response to the DOS prompt. If there is only a single source file, this command will compile the program "my_prog.f", and then link it creating an executable file named "my_prog.out". To load an executable program onto an AT-Super and begin execution, use the command 29K. For example, to run the program "my_prog.out", the user would type:

```
        29k my_prog.out
```

in response to the DOS prompt.

If there are multiple source files, compile each of them separately with the -c compiler option. This causes the linking to be suppressed and for an object module to be produced. The command

```
        fcomp  -c  my_prog.f
```

would result in the object module "my_prog.o" to be created. The fcomp command will also link multiple object files together. If the files "small.f", "medium.f", and "large.f" had all been compiled with the -c option, the command line to link them into an executable file would be

```
fcomp small.o  medium.o  large.o
```

The name of the executable file would default to the name of the first file in the list (in this case, "small.out"). For complete documentation on the FORTRAN Compiler operation, please consult the FORTRAN User's Manual.

3.4 Compiler Batch Files

If you are working on a project that has a large number of source or object files, you may wish to take advantage of the compiler batch file option. A batch file is simply a text file with compiler commands typed in it. Suppose that you have two files that you compile and then link with 4 object files and then direct the output to another file name. The compiler command for this operation can be written to a text file which we will call "proj". The contents of "proj" would be

file1.c file2.c file3.o file4.o file5.o file6.o -o new_file.out <ret>

To execute this batch file, after the compiler command type an @ followed by the name of the batch file. There should not be any spaces between the @ and the file name. For the example above, to execute the batch file, you would type

```
ccomp  @proj  <ret>
```

in response to the DOS prompt. Batch files also work with the FORTRAN Compiler.

3.5 Debugger

The debugger is a low level AMD 29000 debugger developed by YARC for the AT-Super product. It supports loading and executing a 29000 Common Object Format file (AMD HIFF standard), setting and removing breakpoints, software single step mode, disassembly of code in memory, and display of 29000 and 29027 floating point registers.

3.5.1 Running The Debugger

The debugger is started by typing

```
DEBUG29 program name <ret>
```

in response to the DOS prompt. The following should appear on the screen:

```
YARC AT-Super 29000 Debugger (V0.16) (c) YARC Systems, Westlake Village,
CA
Breakpoint at ADDR; (29000 address)            'B 810c'
Copy from ADDR, till ADDR, to ADDR;            'C 0 100 1000'
```

```
DIsasm from ADDR, [to ADDR];                          'DI 810c 81f0'
Dump from ADDR, [to ADDR], [cycles to repeat];        'DU 80002000 800020
Fill from ADDR, to ADDR, 32-bit VALUE;                'F 0 100 70400101'
Go to ADDR [ADDR ADDR ...]  (Trace OFF)               'G 810c 8118'
Kill breakpoint NUMBER;                               'K 0'
Procedure skip NUMBER;  (Trace ON)                    'P 0'
Quit                                                  'Q'
Register display [FPU, General, Local, Special] 'R','RF','RG', etc.
Substitute from ADDR;                                 'S 0'
Search Word from ADDR to ADDR for WORD              'SW 40400 42000 1234abcd'
Trace NUMBER;  (Trace ON)                             'T 10'
<return> go to next instruction (Trace Mode) or break point;
? shows this help message.
```

This help screen will appear if you type a ? at the prompt. When you are ready to run your code type a G and a breakpoint address or type a T to select Trace Mode. Your program will get control after the startup code (file crt0.o) is executed. Control will revert to the YARC 29000 Debug window before each instruction is executed (in Trace Mode) or before each break point. Then you may use any of the debugger commands on the help screen above. It is useful to have a link map of your program. Use the -a switch on the command line when compiling or linking your program to generate a link map.

3.5.2 Debugger Commands

A few of the debugger commands may need further explanation. The B command sets a break point at the address given but if no address is given a list of all currently set break points is shown. The K command needs the break point number, not the address, to remove a breakpoint.

The P command skips past any procedure calls, while the T command shows each instruction executed. Since this is a software debugger it is not possible to trace instructions when the DA (Disable All) bit of the processor status register is set, so most traps cannot be traced. If a P or T command is given without a number, the number last used with a P or T command is used.

Pressing the return key will either execute one more instruction, or a whole procedure if the instruction was a CALL and the P command was used last.

3.5.3 Addresses

29000 data memory starts at 29000 address 80000000 (hex). Currently, system data runs from 29000 address 0x80000000 to 0x80001fff, and user data starts at 0x80002000. User instruction memory starts at 29000 address 0x00008000 which is usually where the startup file crt0.o is loaded. Your program code usually starts at or after 29000 address 0x0000810c. Please see the Memory Map in the next section for more information.

4. INTERFACE

The AT-Super is usually operated as an engine for advanced scientific processing. The average user only needs to know how to compile, load, and execute programs, all of which have been explained in earlier chapters. Developers who are integrating the AT-Super into a package, however, will probably want additional information on how to directly control the AT-Super from within their own application. This section provides information on the interface between the host system and the AT-Super.

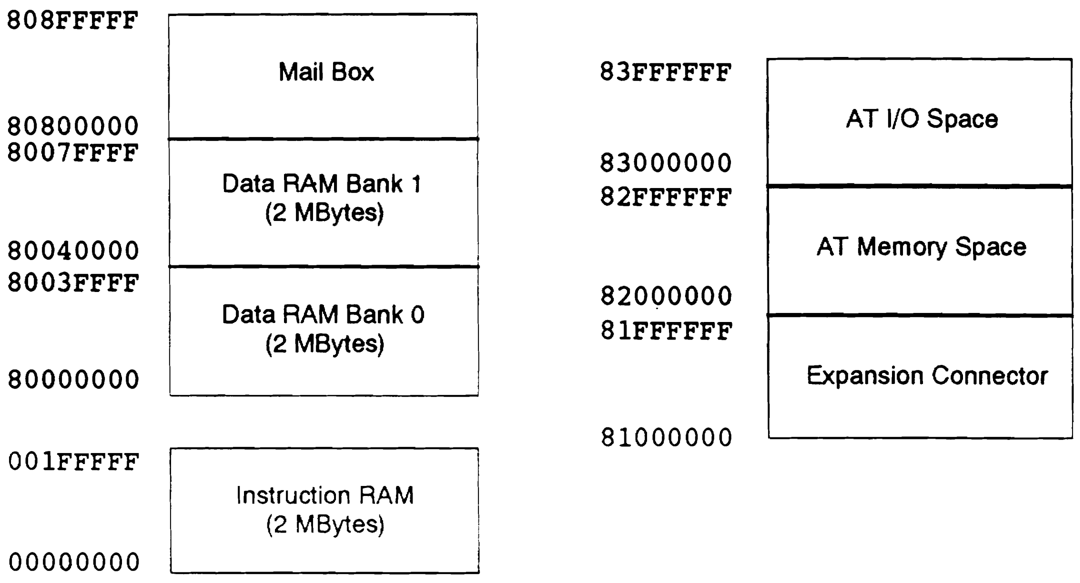
4.1 Host Interface (HIF) Specification

Advanced Micro Devices has developed a software specification for Am29000 products to provide better portability between execution environments. This specification, known as HIF, is currently in release 1.5 and is available from Advanced Micro Devices. This document is used as reference material for software developed for the AT-Super. Every attempt has been made to adhere to this standard and to provide binary portability from one HIF environment to another. However, as HIF is apparently an evolving standard, no warranty is provided as to HIF compliance or portability.

HIF, the Host Interface, defines a standard set of kernal services available to programs running on HIF compatible systems. The AT-Super provides these kernel services to provide the interface with the high-level language library functions, such as disk and screen I/O.

4.2 AT-Super Memory Map

The memory map of the AT-Super is shown in the diagram below. Not all adress lines are decoded for all memory locations, and so there are a number of places where "shadow" memory exists. For example, while the Mail Box is officially located in the low-order byte of location 8080000 Hex, it may appear at every memory location between 8080000 Hex and 808FFFFF Hex. The user should not attempt to utilize these "shadow" areas, since future versions of software or hardware may modify them.



4.3 Control of the AT-Super

The AT-Super is controlled by the host through four byte-wide I/O ports plus a 16K byte page of memory. The I/O ports are used in the control of the board, while the 16K byte block of memory is a "window" into the memory on the AT-Super. By modifying values in the I/O ports, this "window" can be made to address any memory location on the board.

The I/O ports are sequential and start at the base address specified when the board is congfigured (208 Hex, 228 Hex, or 240 Hex). The first I/O port is a control location. A read from this location gives information from the Serial EEPROM on the AT-Super. This EEPROM is not a user-accessible device. It is solely for the use of YARC Systems and its liscensed VAR's. Writing to this first I/O port allows resetting of the board and the handling of interrupts. The second and third I/O ports contain the high order bits of the address for the sliding window. The fourth I/O port is the Mail Box Register, which is used for communications between a program running on the AT-Super and the controlling executor/loader program running on the host.

4.3.1 Control Port Register

OFFSET	ACCESS	7	6	5	4	3	2	1	0
+ 1	Write Only	RST	DRQEN	IRQEN	B4	B3	B2	B1	B0

RST:	(RESET) A 1 in this postion will cause the AT-Super to be reset.
DRQEN:	(DMA Request Enable) - A 1 will allow the AT-Super to perform DMA operations to the Host (if Bus Mastering is enabled).
IRQEN:	(Interrupt Request Enable) - A 1 will enable interrupts from the AT-Super to the Host.
B4-B0:	Reserved for use by YARC Systems

4.3.2 Address Registers (Window Control)

Low-Order Address Bits Register

OFFSET	ACCESS	7	6	5	4	3	2	1	0
+ 2	Write Only	A21	A20	A19	A18	A17	A16	A15	A14

High-Order Address Bits Register

OFFSET	ACCESS	7	6	5	4	3	2	1	0
+ 3	Write Only	A31	A28	A27	A26	A25	A24	A23	A22

Note that in the High-Order Address Bits Register, the address lines A30 and A29 do not appear. By looking at the AT-Super memory map, it can be seen that these lines are "Don't Cares". Address line 31 controls whether the access is for the Instruction RAM (A31 = 0) or for the Data RAM (A31 = 1). As an example, suppose that we wish our memory "window" to point at the 16K block of memory between 8002C000 Hex and 8002FFFF Hex. We would set our address registers to point to location 8002C000 Hex. This is

```
1000_0000_0000_0010_1100_0000_0000_0000
```

in binary. Therefore we would write the value

```
1000_0000   ( 80 Hex)
```

to the High-Order Address Bits Register and the value

```
0000_1011  ( 0B Hex)
```

to the Low_Order Address Bits Register.

After these values have been written to the Address Registers, the 16K window will be located over the desired area. If the board was configured with the memory window base at D0000 Hex (the default), then any reads or writes from D0000 Hex to D3FFF Hex will actually be accessing the AT-Super memory from 8002C000 Hex to 8002FFFF Hex.

4.3.3 Mail Box Register

OFFSET	ACCESS	7	6	5	4	3	2	1	0
+ 4	Read Write	D7	D6	D5	D4	D3	D2	D1	D0

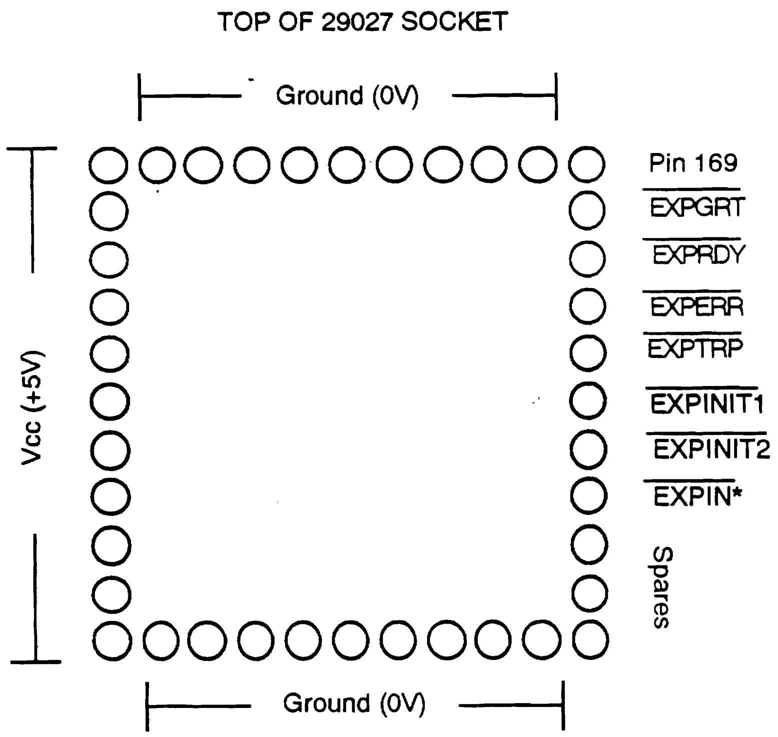
Communication between the AT and the AT-Super can occur via memory or a Mail Box port. This Mail Box is a byte wide register which, when written to, can cause an interrupt to the other side. In other words, if the AT-Super writes to the Mail Box, an interrupt will occur, if enabled, on the Host (AT). The interrupt is cleared when the Mail Box is read by the AT. If the AT writes a byte to the Mail Box register, an interrupt is sent to the Am29000. This interrupt is cleared by the Am29000 by reading the Mail Box register.

The Mail Box is used for communications between the board and the program on the Host which handles the HIF interface.

4.4 Expansion Connector

The AT-Super has an expansion connector that is an integral part of the socket for the Am29027. This connector provides a mechanism for custom hardware to be added to the AT-Super in a closely coupled manner like the memory system. The expansion connector is addressed in the Am29000 address space between 81000000 and 81FFFFFF.

The Am 29027 has almost all of the necessary signals from the system bus already present. YARC Systems has added an additional inner row of pins to the socket which carry the few additional signals necessary for a daughter card. The diagram below shows the signals available from this inner row of pins. If you intend to design a daughter card, please contact YARC Systems directly so we can provide you with an up-to-date set of PAL equations for the few PALs on the mother board which must be modified. In addition, YARC Systems services are available for the design of custom daughter cards for specialized applications.



* Note: EXPIN must be grounded if daughter card is present.

APPENDIX 1: DIPSWITCH TABLES

The dipswitch tables for the AT-Super are provided below. Note that the user should run the configuration program and then use the switch settings illustrated by that program. These tables are provided for completeness.

Switch Block 1:

Switch #	Function	Open (Off)	Closed (On)
1	I/O Address Select 1	see table	
2	I/O Address Select 0	see table	
3	Sixteen Bit Transfers	Disabled	Enabled
4	Motherboard Memory	512K	640K
5	Interrupt Request 10	Disabled	Enabled *
6	Interrupt Request 11	Disabled	Enabled *
7	Interrupt Request 12	Disabled	Enabled *
8	Interrupt Request 15	Disabled	Enabled *

* Enable only one Interrupt Request

I/O Port Address Settings:

I/O Port Address	Switch 1	Switch 2
disabled	On	On
240 Hex - 243 Hex	On	Off
228 Hex - 22B Hex	Off	On
208 Hex - 20B Hex	Off	Off

Switch Block 2:

Switch #	Function	Open (Off)	Closed (On)
1	DMA Request 7	Disabled	Enabled *
2	DMA Request 6	Disabled	Enabled *
3	DMA Request 5	Disabled	Enabled *
4	DMA Request 0	Disabled	Enabled *
5	DMA Acknowledge 7	Disabled	Enabled *
6	DMA Acknowledge 6	Disabled	Enabled *
7	DMA Acknowledge 5	Disabled	Enabled *
8	DMA Acknowledge 0	Disabled	Enabled *

* Only one DMA Request and its associated Acknowledge should be enabled.

B PC interrupt notes

This section contains additional notes to accompany Section 4 on the real-time interrupt.

B.1 Application Notes

1. Check the compiler flags in the makefile for intrpt.c and be sure they are compatible with the compiler flags of the current application. In particular, check the model size and four byte alignment flags. If they are incompatible, then either change the application's flags or re-compile intrpt.c with the application's flags.

2. Make sure that the user_tick_handler's maximum execution time is known and that the tick_freq is slow enough to allow for this maximum time.

3. There is a define in the include file pctimer.h that may or may not be machine dependent. In other words, if the timing of the interrupt is not at the frequency requested, check this value with the machine. So far, this value does work for IBM PC-XT, IBM PC-AT, and Jameco JE2019 (286 with turbo clock).

```
#define CLOCK_FREQ 1193180
/* verify! may be Machine Dependent */
```

B.2 Sample Listing

```
/*********************************************/
/*        prototype listing                  */
/*********************************************/
#include <stdio.h>
#include <conio.h>

/* declare routines defined in this listing */
void main(void);
void user_ctrl_break_handler(void);
void user_tick_handler(void);

/* declare routines defined in the intrpt software */
extern void init_rt( int tick_freq );
extern void tick_handler_switch( int on_or_off );
extern void rest_pc( void );

#define TRUE 1
#define FALSE 0
#define ON 1
#define OFF 0

/* create a global variable that the foregound  */
/* process will check to see if it should       */
/* continue to loop or exit.                    */
int force_exit = FALSE;

void main(void)
{
    int tick_freq;
    int ch;

    force_exit = FALSE;

    printf("Enter the desired freq. in hz: ");
    scanf("%d", &tick_freq );

    /* start up real time interrupt */
    init_rt( tick_freq );

    /* foregound checks force_exit before each loop */
    while( ! force_exit )
    {

/* Check for a user keyboard input.                  */
/* Allowable inputs are:                             */
/*    b - (BEGIN) to enable the calling of the user  */
/*        defined tick handler when tick occurs.     */
/*        NOTE: it is initially disabled.            */
/*    e - (END) to disable the user defined tick     */
/*        handler from being called when tick occurs */
/*    q - (QUIT) to quit the foregound, restore the  */
/*        and exit.                                  */
        if (kbhit())
        {
            ch = getch();
            if (ch == 'b')
            {
                /* enable user_tick_handler */
                tick_handler_switch( ON );
            }
            if (ch == 'e')
            {
                /* disable user_tick_handler */
                tick_handler_switch( OFF );
            }
            if (ch == 'q')
            {
                /* quit the foreground */
                force_exit = TRUE;
            }
        }
    }

    /* restore pc to original condition */
    rest_pc();
}


void user_ctrl_break_handler(void)
{
    /* it the program malfunctions, */
    /* this may be an easy out      */
    force_exit = TRUE;
    return;
}

void user_tick_handler(void)
{
    /* Do the user defined interrupt routine.   */
    /* This is where you can insert             */
    /* a trajectory generator,                  */
    /* a servo control law, etc.                */

    /* trajectory();    */
    /* servo();         */

    /* When finished, just return. */
    return;
}

/*********************************************/
/*        end of prototype listing          */
/*********************************************/
```

C TRACS Utility Programs

This section will describe various utilities offered in TRACS. These utilities are often critical to correct performance of the robots and the system as a whole. It is important that the user be familiar with them.

C.1 Encoder Calibration

The encoder calibration routine is located in `\pkg\p250\calib` and is named `calib.c`. This routine uses the joint processor boards to calibrate the robot's encoders. The encoders are relative, not absolute, and thus with each powerup or reset of the controller, the exact location of the joint motor is not known. The calibration routine will initialize the encoders to their correct value. The calibration routine uses the PC-AT in real-time mode and the robot joint controllers *only*, thus the AT-Super is not involved. The calibration routine takes the following steps:

1. The controller boxes should be on *but arm power should be off.*

2. Enter the command `calib` at the PC prompt.

3. It will query which robot to calibrate. Answer as follows:

   - **0** — calibrate robot zero
   - **1** — calibrate robot one
   - **2** — calibrate both robot zero and one successively.

4. The screen will inform you that the robot of interest is limped and that you may place it in the nest. This is done as follows:

   (a) With one hand, securely hold the robot of interest at its wrist joint.

   (b) With the other hand, turn on arm power *by pushing the flat black button on the controller front panel.* Do not use the remote switch to turn on the arm power because *both* robot arm will be powered and only one is being calibrated. The robot brakes are released and no power is being supplied to the actuators, thus it is in the limp state. If this is not the case, immediately hit the red arm power off button on the controller front panel.

   (c) With the robot limped, position it vertically straight in the right shouldered mode. Rotate joint 4 clockwise about its axis (given the axis is pointing up) until it meets a mechanical stop. Now rotate it back (counter-clockwise) approximately 90 degrees until the axis of joint five is parallel to the axis of joint two.

   (d) Rotate joint one clockwise until the axis of joint two is perpendicular to the nest plate located at the back of the robot base. Now bend joint three 90 degrees such that the end-effector approaches the nest. Continue approaching the nest by coordinating the bending of joints two and three.

   (e) Flip joint five so the end-effector is facing the nest. Before inserting the calibration end-effector into the nest, rotate joint six clockwise (about the axis from the wrist) until it meets a mechanical stop, then rotate it back (counter-clockwise) slightly such that the end-effector completely matches the nest fixture. Now it can be inserted into the nest. If the process is done correctly, the robot can move out from the nest with only a rotation on joint two.

5. Now that the robot is nested, hit the `enter` key of the PC to continue the calibration process. The program will move the end-effector out of the nest by rotating joint two. Then it will search each joint successively for the index pulse on the encoder. The principle behind the calibration is that the nest position forces the encoders to a known position, plus or minus half a motor revolution (usually a few degrees at the joint). Finding the index removes this uncertainty, and the robot location is known completely.

6. To verify the calibration is correct, the robot then will be parked in the vertically straight position. The operator can visually determine the routine's success because if it has errored, the misalignment will be by a complete motor revolution (usually a few degrees at the joint). Occasionally, the program will not continue from here as it should because the final park position has not been met and is in fact short by a few encoder counts. This results because the joint friction is high enough and the error is small enough that the control law running on the controller joint boards does not request a large enough current to move the joint. This situation can be resolved by gently touching the robot so the joints move slightly.

7. The calibration for the robot of interest is finished, the program will either exit or proceed if calibration is desired for both robots.

The calibration routine's repeatability is very stable, however the calibration values for the encoders were obtained by external measurements and verifications within the workspace. Often, repair work done to the robots involves disconnecting the joint shaft from the motor shaft. When this happens, a new initial calibration value for that joint (and any related joints, in the case of the wrist coupling) must be found. There is software within the system that can assist in this endeavor by placing the endeffector at known points within the workcell and measuring the error between the actual and expected encoder values. This method is not exact but quite error prone due to positioning errors, backlash, etc. But, it is none the less a starting point. The `slow` utility may also be employed to produce small changes in the encoders and to visually determine the effects (see section C.3 below).

C.2 Limp

The program `\pkg\p250\limp` is used to reposition the arms. Functionally, the program outputs zero current to the actuators and releases the brakes when the operator turns the arm power on. The program will ask which robot to limp, and the arm power should not be on until this question is answered. The limp program does not use the AT-Super nor the real-time environment, it simply communicates directly to the Unimation joint controllers.

C.3 Slow

The program `\pkg\p250\slow` provides to the operator direct access to the Unimation joint boards from the keyboard. Since this program operates at the lowest level, the user must be very familiar with the joint board operation. This information is documented in the *Breaking Away form VAL* manual [1]. To summarize, the joint boards send and receive information in two forms: either a command word and then a data word, or a command word and then a vector of six data words. The `slow` program allows the user to directly type in the command and data. (The commands are documented in the aforemention manual.) The program also enables the user to choose either robot and to do both types of communication (word or vector).

C.4 Maintenance

The main software maintenance involves periodically backing up the hard disk. Secondary maintenance includes purging files from the `\pkg\rt\log` directory and having users keep their personal subdirectories as compact as possible.

The backup of the hard disk is accomplished with the FASTBACK package and is done as follows:

1. Go to the root directory (type: `cd \`).
2. Start the package by typing **`fb`**.
3. At the menu, use option F9 to load the set-up file: **`tracs.fb`**.
4. Use the arrow keys at the menu to choose the *Start Backup* field and press **`return`**.
5. At the top of the new menu, select *Estimate Backup* and press **`return`**. Check the total number of kilobytes of memory needed. Approximately 2 megabytes will fit on a single floppy due to compression techniques. Verify that you have enough formatted floppies on hand before beginning.
6. At the top of this menu, select *Run Backup* and press **`return`**. Insert the first floppy and the backup shall begin. When the first floppy is full, it will chime and ask you to remove it and to insert the next floppy. The floppy disk drive light will *not* go out during the entire process, so do not wait for it, just simply follow the instructions on the screen and change floppies when directed to do so.
7. When the backup is complete, choose *Quit* at each menu until the package exits back to the DOS prompt.

Store the backup floppies in a safe area, perhaps one copy should be maintained outside of the lab. This process should be done once every two weeks, and even more often if a lot of development has occurred on the system within a short period of time.