8-0 INTRODUCTION

This section of the manual is intended for those that are familiar with Intel 16 bit programming practices. Memory elements defined here are defined as Intel memory items and the definitions of these can be found in the appropriate Intel literature.

System parameters are used to change the robots operating characteristics and to observe what the robot is doing at any particular time.

8-1 MEMORY MANAGEMENT

The CRS Robot System Controller uses the Intel 8086 segmented architecture scheme. The Controller contains three different physical types of memory;

1. Four (4) Kilobytes of Low Power volatile CMOS memory:
   Used for Scratchpad use, 8086 stack space, and the interrupt vector space.

2. Four (4) Kilobytes of Battery-backed CMOS memory:
   Used for System Parameters.

3. Eight (8) Kilobytes of Battery-backed CMOS memory:
   Used for User memory space. This memory can be expanded to 56 Kilobytes.

4. 256 Kilobytes of EPROM memory:
   Used for firmware requirements. This memory can be expanded to 512 Kilobytes for future firmware or custom requirements.

Data integrity is maintained by using checksums on all program, variable, location and path data accesses.

8-2 COMPUTATIONAL ACCURACY

The Robot System Controller software utilizes the Intel 8087 Math co-processor fully for all mathematical calculations. Real numbers are stored in memory as four (4) byte quantities, giving 9 digit precision. Where possible, calculations are made using the full eighty (80) bit (10 byte) capability of the 8087 device, yielding a more accurate result. Although mathematical accuracy may vary, depending upon the nature of the calculations, achievable accuracy will be better than one part per million.

8-3 PATH CONTROL - CLOSED LOOP CONTROL

The CRS Robot Controller has complete closed loop DC Servo Control for up to eight (8) axes. Gains can be programmable for a wide range of motion requirements. Depending upon the controller type, the servo loop can be closed with different algorithms.

A150 Controller

This is a proportional/differential gain controller. Only the proportional gain is accessible through the RAPL-II GAIN command. The closed loop control is achieved in a period of approximately 3.5 milliseconds. The proportional gain acts on the positional error of each joint of the robot. This error is the difference between the commanded and the actual position of the joint. The actual position is maintained at the servo loop rate. The positional command is updated by the appropriate command generator software, and then is fed through a single order low pass filter with a time constant of 56 milliseconds.

A250 Controller

This controller is based on a fully distributed control architecture, with individual joint controllers. Each joint controller performs a full PID closed loop algorithm in a 1 millisecond loop time, providing high performance. As in the A150 controller, the proportional gain acts upon the difference between the commanded and actual position. The differential gain provides a degree of damping by acting upon changes in joint velocity. The integral gain provides a static error compensation, by acting upon steady state positional errors. The A250 controller is pre-configured at the factory, and only small adjustments to the values should be made when necessary using the RAPL-II GAIN command.

8-4 PATH CONTROL - COMMAND GENERATION

The A150/A250 series of controllers have multiple path generation modes which can be summarized as three major modes of path control:

1) Joint Interpolated motion

MOVE, APPROACH, DEPART, MOTOR, JOINT, MI, MA are RAPL-II commands that create joint interpolated motion. In the A150 controller, the joint interpolated command rate is 4.5 milliseconds. In the A250 controller, the joint interpolated motions are performed at 1 millisecond intervals within the IAXIS axis cards. The Joint Interpolated motion guarantees that all joints to be moved will start and stop together. It also guarantees that maximum acceleration and decceleration will not be exceeded for each axis. Command updates occur at four milli-second intervals. Figure 8-2 shows typical acceleration, velocity and position profiles for a joint interpolated robot motion.

2) Straight line Motion

JOG, straight line MOVE, DEPART, APPROACH, JOG commands provide a straight line path in cartesian space. Cartesian velocity can be controlled by issuing the ENABLE CARTVEL command, which will process the speed command as a percentage of the maximum cartesian velocity (@MAXSPD command). Otherwise, speed control is based on maximum joint velocity. In Straight Line, command updates are performed at a slower rate (18 milliseconds for A150 controller, 16 milliseconds for the A250 controller). This mode of control is a specialized form of the PATH control described below. The "knots" are calculated by RAPL-II based on a linear interpolation between end points. The number of knots used for the straight line is normally 10, which gives reasonable accuracy to the path. If a higher processing speed is required, the number of knots may be reduced. The ROBCOMM package with its memory edit feature can be used for this. Acceleration and decceleration in this mode are based on world coordinates instead of individual joints. If wrist motions are required during the straight line path, then the wrist rotational speeds are continuous throughout the straight line path segment. Joint limit checking is performed before the straight line move, and if any joint exceeds the programmed limits, the command is immediately terminated, and an error message is displayed.

3) Continuous path

CPATH, CTPATH, GOPATH. Defining a path curve through space by selecting a number of intermediate points through which the robot will move. Again, the selection of the CARTVEL parameter will determine the mode of velocity control, but the algorithm always generates a path through the intermediate points.

The continuous path algorithm utilizes a cubic spline technique which will join the path points, called 'knots' in such a way as to make the velocity of the joints adjust smoothly between paths. The result of this is a motion which stops only at the end of the path. The algorithm will determine the parameters of the paths so that the robot moves through each of the knots. It does not relate to the programmer exactly how the ensuing path will appear. It is up to the programmer to provide sufficient points to define the path if precise path control is required.

The cubic spline algorithm requires the storage of path parameters for the continuous path. These parameters must be maintained until the path has been completed. Reserved memory is used to store the path parameters generated with the CTPATH command. A small buffer is maintained for use with the straight line moves and all CPATH commands. The CTPATH command can generate and store up to 8 pre-programmed paths; ready for instant recall by the GOPATH command.

The storage requirements for a continuous path are calculated here:

B is the number of bytes required for parameter storage,
K is the number of knots which define the path, and,
A is the number of axes that have been programmed.

$$B = 8 * K * (1 + A)$$

The expansion memory option must be provided for the CTPATH command to be accessible. To illustrate the memory requirements for the continuous path, up to 256 knots can be programmed using the CTPATH command. This would require memory of 10240 bytes for a full 8 axis implementation, according to the equation provided previously.

Due to the mathematics of the algorithm, the programmer must not use identical locations in consecutive elements of the path knot list. Doing so will cause a math error and processing of the path parameters will stop. The following statement will cause an error.

```
10 CPATH A,B,B,C
```

since the point B is used consecutively in the list.

For a large number of knots, the processing time of the continuous path parameters can be noticeable. This is not typically a detriment if good programming practices are adhered to. Typical applications use one path which can be calculated at the start of the application program. Subsequent calls to GOPATH will execute the programmed path with no delay.

8-4 PATH CONTROL - COMMAND GENERATION (Continued)

4) VIA path.

The VIA command defines a curve through space based upon a set of intermediate points that will be approximated, but the robot may not move through them, depending upon the acceleration constraints of the robot. The CARTVEL flag determines the mode of velocity control, as before. When CARTVEL is enabled, the path between points is generated as a straight line. With the A150 series, the VIA command is allowed only when CARTVEL is disabled. IA path commands are generated at 16 millisecond intervals for joint interpolated speed profiling, or 32 milliseconds with cartesian velocity profiling selected with the CARTVEL command.

Command generation can be halted using the RAPL-II "HALT ON <[-]Input#>" command and enabling the HOLD parameter. When an interrupt occurs, all joints deccelerate at the rate determined by the characteristics of the filter. When the interrupting signal returns to normal, all joints resume motion also at the rate dictated by the filter stage. This input should be treated as an emergency stop feature, and should not be used as a normal motion hold.

A150 Controller

All command generation output is digitally filtered to provide an update rate of 0.0035 seconds per update for each servo axis.

A250 Controller

Joint interpolated motion is handled entirely by the joint controllers. All other modes of path generation are provided by the motherboard 8086/8087 tandem processors. High levels of motion smoothness is provided by additional compensation by the axis cards, before the commands are sent to the motors. This compensation provides higher performance in terms of speed and smoothness.

8-5 INPUT/OUTPUT SCANNING

Digital input/output scanning takes place at approximately 40 milli-second intervals. This is the same interval used for the system clock. The 'ONSIG' and HOLD function input detection and the arm power check occurs at this rate as well.

8-6 MEMORY ALLOCATION

The robot controller has two types of on-line memory. The EPROM memory contains the system firmware, and is up to 512 Kbytes in capacity. The RAM memory is 16 Kbytes long, and is expandable to 64 Kbytes of total memory.

The firmware consists of two basic sections. It includes a monitor level structure which defines the hardware of the system. On top of this level, is the RAPL-II operating system which the programmer deals with. It handles the execution of RAPL-II programs, along with the generation of robot path coordinates for motion control.

The RAM memory itself is divided into several functional areas. These are described in Table 8-1. Most importantly, its uppermost segment is allocated to the user. It is referred to as User Memory. It is this section of memory which contains user programs, path storage, variable storage and location storage. The user memory occupies all but the lowest 8 Kbytes of RAM.

Command functions like ALLOCATE and NEW alter the contents of the user memory. These functions are described in detail in the RAPL-II Programming Manual. The FREE command will show the amount of memory currently used.

The User Memory is further divided into five main areas. These areas are used to store the program table, the programs themselves, robot locations, variables and path data. Each of these areas store data in different structures.

Since the five types of user memory can be allocated (partitioned) by the user, the system programmer should understand how these pointers work.

The Program Table

The program table is the directory of stored RAPL-II programs. Each robot program entry has three fields: the first field is an 8 byte character string which contains the program name, the second is a word quantity which contains the index in the program buffer to the start of that program, and the third is a word quantity which contains the program checksum. (The checksum is used to prevent gaurd against the memory being damaged or altered in an unexpected way.) Thus each entry in the table requires 12 bytes: 8 for the name, 2 for the index, and 2 for the checksum.

When a new program is created, there must be an empty space in the program table or an error will result.

The Program Buffer

The program buffer is a contiguous byte array which contains the user programs. The length of this buffer is set when the ALLOCATE command is entered. Its size depends on what is left of the user memory after allowing for Program, Location and Variable tables, as well as the Reserved memory.

The program buffer stores programs as sequential ASCII characters. Program information is entered by typing in through the user terminal, or downloading from a Master computer using the ACI interface and support software, such as the ROBCOMM software system.

The location of the start of a program is determined by RAPL-II from the index in the program table, relative to the program buffer pointer. Each program is terminated by a dollar-sign character (**$**) which acts as a seperator. Memory in the program buffer is altered dynamically whenever editing of a program takes place. The memory is "squeezed" up or down as lines are entered, modified, or deleted. Thus the index of all entries in the program table will have to change whenever a line is entered into a program located below the one changed. RAPL-II makes these changes unseen by the programmer. The program buffer can be subdivided into two sections immediately after an ALLOCATE command. The HIMEM command will partition an area away from the uppermost program buffer. This reserved memory will then remain untouched for special programming applications, or PCP's.

Variable Storage

Variables are located in the variable table. The length of this table is defined when the ALLOCATE command is entered. Like the program table, there are three fields for each variable entry in the table. The first is the eight character string used to store the name. The second is a four byte real value which stores the value of the variable. The third is a word quantity used to store the checksum.

Location Storage

Locations (both precision, and cartesian types) are stored in a location table containing ten fields for each location. The first is the 8-character name string. The next eight fields are four-bytes each used to store the components of each location. The RAPL-II controller can control up to 8 individual axes of motion. A precision point can use position information for up to eight axes. In this case, each of the eight fields store a double size integer value of each axis motor coordinate (in encoder pulse units). When a cartesian coordinate is stored, it uses the first six of these 8 fields to store a four byte real number representation of each world component. If a GANTRY command has been specified, the two remaining fields of the eight are used to store the joint values for the programmed extra axes. If a TRACK command has been specified, only one of these fields is used - the other field is stored as zero. The tenth field stores the checksum.

The configuration of the user memory can be described in terms of structures and their elements. The appropriate PL/m-86 definitions would appear as:

```
ProgramTable(ProgramTableSize) structure (
          Name(8)    byte,
          Index      word,
          Checksum   word) at (ProgramTablePointer)

VariableTable(VariableTableSize) structure (
          Name(8)    byte,
          Value      real,
          Checksum   word) at (VariablePointer)

PrecisionPointTable(LocationTableSize) structure (
          Name(8)    byte,
          AxisVal(8) dword,
          Checksum   word) at (LocationPointer)

CartesianLocation(LocationTableSize) structure (
          Name(8)    byte,
          Coord(8)   byte,
          Checksum   word) at (LocationPointer)

ProgramBuffer(ProgramBufferSize) byte at (ProgramBufferPointer)
```

Similar 'C' code definitions are:

```
       ProgramTable struct
       {
      char Name[8];
      int  index;
      int  Checksum;
       } *ProgramTablePointer;

       VariableTable struct
       {
      char  Name[8];
      float Value;
      int   Checksum;
       } *VariablePointer;

      PrecisionPointTable struct
       {
```

```
    char Name[8];
    long AxisVal[8];
    int  Checksum;
     } *LocationPointer;

    CartesianLocation struct
     {
    char Name[8];
    float Coord[8];
    int  Checksum;
    }  *LocationPointer;
     char ProgramBuffer[ProgramBufferSize];
        char *ProgramBufferPointer;
```

The checksums associated with each element of the structures consists of the arithmetic sum of each byte in the structure element. Only the low byte of the checksum is used. The checksum used in the program table is the checksum of the entire program that it references, from the first byte of the program, up to and including the '$' delimiter used to end a program.

Exact addresses for each of the system parameters mentioned above can be found in the memory map for the current version of RAPL-II. This information can be found in Appendix G.

8-6 MEMORY ALLOCATION (Continued)

Address (hex)	Description
FFFF:0000	Bootstrap address
F000:FFEF	Upper limit of firmware address space
C000:0000	Standard 256K firmware address boundary
8000:0000	Extended 512K firmware address boundary
8000:0000	Maximum Firmware address boundary
0000:FFFF	Optional RAM expansion memory upper limit
0000:3FFF	Standard 16K RAM memory upper limit
0000:2000	Approximate start of USER memory (may vary according to firmware release version, and other customer options) Refer to Appendix F.
0000:1FFF	Approximate top of CPU scratchpad space and system parameter space
0000:1200	Start of CPU scratchpad and system parameter space
0000:0FFF	Approximate Top of 8086 stack space
0000:0800	Start of 8086 stack space
0000:0400	Start of Volatile path data buffer
0000:03FF	Top of 8086 vector interrupt space
0000:0000	Bottom of memory

Table 8-1 Memory Map