B-1 INTRODUCTION

The ACI permits external computer systems to communicate with one or more robot controllers on a single RS232 link.

In its basic configuration, this protocol is used to transfer raw data either to or from the master device. The protocol allows any chunk of 8086 memory to be the object of the communication in a segmented addressing technique.

The protocol permits error checking and automatic transmission retries in order to establish a communication link.

B-2 FUNCTIONAL DESCRIPTION

The ACI is a master/slave protocol. All robot controllers are configured as slaves in the network. Any external compter would then have to be a master. Only a master can establish a communication. All the communication initiative must be taken by the master unit.

B-3 MASTER PROTOCOL

The master device must establish a communication link by specifying a slave device number as a target. It must then confirm to the target that it is indeed requesting a communication. After the link has been established, the master must then provide the slave with the information describing the data transfer that is about to take place. After all of the particulars concerning the communication has been transferred, the actual data is then sent. The data is transferred in packages of 128 bytes. Each 'block' of data contains its own start/stop characters and a checksum test byte to validate the data after transfer. Finally, it is up to the master to close the communication link with an EOT character.

A communication consists of four separate blocks. They are described in detail in the following sections.

B-4 ENQUIRY SEQUENCE

The master issues a simple three byte code to the serial line. This code is;

'R' , slave ID + 20h , ENQ

The master must then follow this sequence with two character times of no transmission. This will ensure that the slave has indeed read a valid enquiry sequence, and not some random chunk of another communication. When the slave has interpreted a correct enquiry, it will issue an appropriate response;

'R' , slave ID + 20h , ACK

When the master reads this response it has established the communication link to the target device.

The slave id number is any number between 0 and 7F hexadecimal. Using the ACI monitor command **@@RN**, the programmer can configure each robot controller with an appropriate slave id number.

Adding a value of 20 hex to the slave number provides security that the second byte of this string does not resemble a control code in any way. This way, the slave device will be able to distinguish an enquiry sequence and can establish communications quickly.

The master should attempt to contact the slave only a limited number of times before abandonning the communication. A failure to establish the communications with the slave after three attempts normally indicates a failure in the link. Either the baud rates are not set correctly at either end, or a physical problem exists with the interface wiring.

A delay of 2 character times should be included in the transmission of the enquiry sequence. This delay is placed between the second and third characters of the enquiry sequence. This time delay ensures that a three byte string from a data block cannot be misconstrued as an enquiry sequence. The master control must ensure that this time delay exists, or proper ACI operation cannot be guaranteed.

B–5 HEADER SEQUENCE

The header block consists of a description of the data transfer that will take place. The header block is broken down into the following byte description;

1	SOH	Start of header character
2	SLAVE ID + 20h	Slave identification number
3	MASTER ID + 20h	master identification number
4	READ/WRITE	data read or write selection
5	MEMORY TYPE	memory access type (see special codes)
6	NUMBER OF FULL BLOCKS	number of full blocks transferred
7	NUMBER OF BYTES IN LAST BLOCK	number of byte in last data block
8	MEMORY OFFSET LOW	target memory starting address
9	MEMORY OFFSET HI	
10	MEMORY SEGMENT LO	
11	MEMORY SEGMENT HI	
12	ETX	end of text character
13	LRC	longitudinal redundancy check

Table B-1

All data values are expressed in hexadecimal notation.

The slave ID number is the same that appears in the enquiry sequence.

The Master ID number must be 01 hexadecimal.

The Read/Write byte identifies what type of operation is to be executed. A write operation will transfer data from the slave device to the master. The read operation is the opposite. It transfers data from the master to the slave. The following codes are used:

READ	**01h**
WRITE	**00h**

The memory access type specifier identifies specific areas of memory that the data transfer can take place in. This eases the burden of programming on the programmer in many cases.

The elementary code '00h' is the general purpose memory access code. This code eneabled the programmer to read or write any byte in the 8086 memory space. This makes the command a very powerful and also a potentially hazardous tool to work with. CRS technical staff should be consulted when this command is used.

A list of the special codes available appears in a later section.

The next two byte in the header identify the amount of data to be transferred. This information is considered as the number of full data blocks to be transferred, and the number of bytes which remain in the last data block. The last data block cannot have 0 bytes in it. A full block contains 128 bytes of data. If the number of bytes to be transferred is an exact multiple of 128, then the last block will contain 128 bytes (80 hex). The logic to determine the number of blocks and the number of bytes in the last block is as follows;

```
let N be the total number of bytes to be transferred,
let B be the number of full blocks to be transferred,
let n be the number of bytes in the last block;

B = N / 128
n = N mod 128          ; (ie. the remainder of the division N/128)
if n = 0 then do
     n = 128
     B = B - 1
endif
```

The memory address identified the starting address of the data transfer operation. It defines (in Intel segment/offset format) where in the robot memory the data will be transferred to or from during this operation.

Header Transfer Error Detection

The LRC byte is the longitudinal redundancy check character . It is the sum of bytes 2 through 11 inclusive of the header block. The slave device sums the values of all received characters and will accept the header block only if it reads an LRC that is the same as its own computed value. If the LRC's do not match, then the header block is not accepted, and the slave will issue a **NAK** character in response so that the master control will know to re-try the header block transfer. The master can only have three re-try attempts before the slave device issues an error, and aborts the communication cycle. If the header block is accpted, the slave device will issue an **ACK** character as a response. The slave also expects the correct control codes **SOH** and **ETX** during the transmission, and any deviation from this pattern will cause an error, and the communication will be aborted. Timeout checks are made between the end of the enquiry sequence, and the accpetance of the SOH character, and also the length of time reuqired between sending an SOH and ETX character.

B-6 DATA BLOCK TRANSMISSION SEQUENCE

The data block transfer sequence is determined by the format specified in the header block.

In a data write command, the slave will begin to write the specified data blocks quickly after it issues an ACK in response to receiving the header block. It expects ACK characters from the master after each block is transmitted. Receiving a NAK will provoke a retry of the previous block. After the last block is sent and acknowledged, the slave will issue an EOT. It expects to receive a final EOT from the master, and then will close down the communication. The data blocks are configured in the following manner;

STX + [DATA BYTE #0] + ,.....,+ [DATA BYTE 127] + ETB + LRC

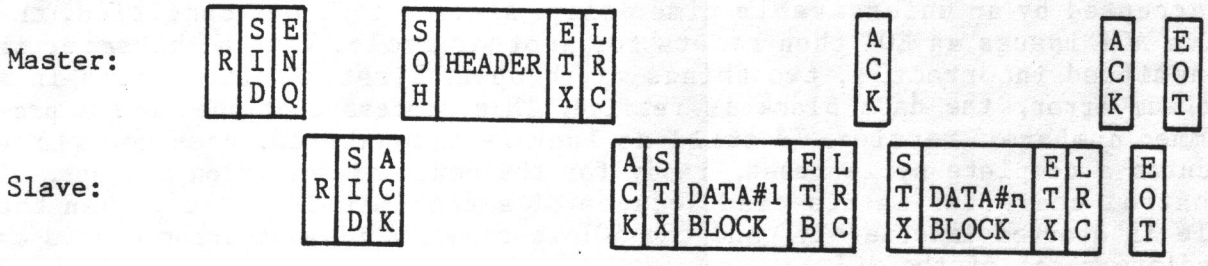
for the full data blocks, and

STX + [DATA BYTE #0] + ,.....,+ [DATA BYTE #n-1] + ETX + LRC

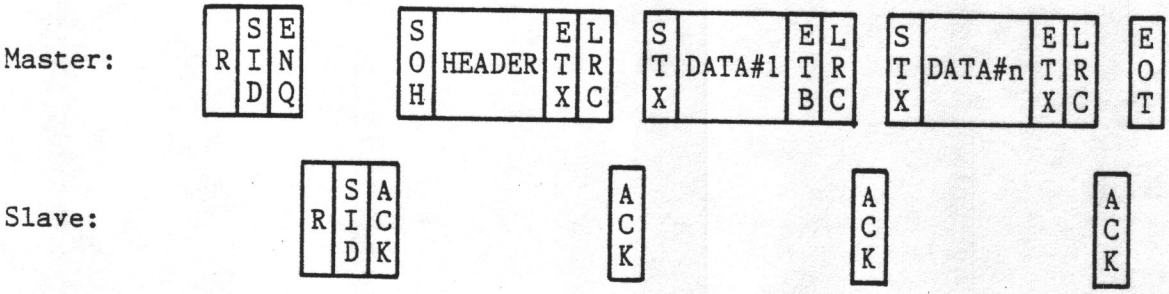
for the last data block.

The last data block uses an ETX character instead of an ETB character in order to signal the very end of the transferred data.

The following diagram illustrates the flow of data between the master and slave devices during a separate read and write cycle.



Similarly, in a read command, the slave will expect blocks of data after the header is aknowledged. It will issue a NAK if it detects an error in transmission. The master will issue an EOT after the last data block is read and acknowledged.



Data Transfer Error Detection

As in the header block transfer, the LRC byte is the sum of all data bytes in the block. Control codes STX, ETX or ETB are not included in the summation. An erroneous LRC check will cause the receiving end to issue a **NAK** character, which will force a re-try of the entire block. Only three re-tries are permitted, whereupon the recieving end should issue an **EOT** and abort the sequence. Timeout checks are made for reception of an STX after a complete header transfer, or a complete previous data block transfer, and also btween accpeting the opening STX of a data block transfer, and reading the ETX or ETB character. Once a data block has been accepted by the ACI in a read operation, the data block is transferred to the target memory address.

B-7 EOT SEQUENCE

The EOT is the final sequence, and is shown in the timing charts above. In the communication, the master ALWAYS has the last word, and the master ALWAYS has the ability to abort the communication. The slave will never knowingly abort the communication once it is started.

B-8 ERROR CHECKING AND RECOVERY

The ACI can handle a range of different communication errors. The two broad classifications of errors are transmission inaccuracies, and character time outs. Character time outs are used when a pending communication cycle has been interrupted by an unforgivable time delay. After a time out condition, the slave ACI issues an EOT then resets for another cycle. When a character is transmitted incorrectly, two things may happen. First, if it results in a checksum error, the data block is retried. This process continues for a programmed number of retries. If still no luck is encountered, then the slave executes a complete cycle reset, ready for the next communication attempt. If a control character is mis-read because of a transmission error, then the cycle is aborted immediately, and the ACI is reset. A timeout error causes an immediate reset of the ACI.

Error Number	Error Description, Cause, and recovery
1	Not used
2	No SOH character found to lead off the header sequence. The SOH character must be the first character of the header. Any other character will cause the ACI to terminate, and the communication cycle will be ended. The ACI will release an EOT character.
3	The ACI encountered a software error when reading in the header block. This is an internal software failure. Contact your CRS Plus representative. The ACI will be reset after this error.
4	A header retry failure was encountered. This means that the ACI tried four times to obtain the header, and was unable to do so due to repeated LRC failures. Usually, this is a sign of a bad interface wiring, or incorrect master control software. If you are using CRS Plus interface software for your host computer, contact your CRS Plus representative.
5	No ETX, or ETB was read at the end of a data block transfer. Remeber that the last data block must have ended with an ETX character. All other data blocks end with an ETB character. The ACI will immediately reset after detecting this error.
6	Not used
7	Not used
8	Data block read retry failure. The ACI terminated due to repeated LRC check failures when reading a data block from the Master.
9	Character timeout. The ACI expected a character, and did not receive one in the allotted time. The ACI will be reset. Check that the Master control did not hang up part way through a transmission, or that the correct number of characters were sent.
10	The ACI read a character other than an STX at the start of a data block. The ACI will be reset.
11	Data block write retry failure. The ACI was unable to deliver a data block to the Master without getting a NAK responce four consecutive times

Table B - 2 ACI Error Codes

Error Number	Error Description, Cause, and recovery
12	A character other than an ACK or NAK was read in response to a data block write. This error can occur on a spurious noise injection into the line, but is a rare error type. If this error persists, then the Master could be responding with an incorrect code. Contact your CRS Plus representative. The ACI will be reset.
13	Timeout error on transmission of a character. The ACI was not able to transmit a character in the allotted time. This timeout would have been caused by a failure in the DART chip (see hardware description) to present a transmit ready signal. The ACI will be reset after this error. Contact CRS Plus if the problem persists.
14	Bad target Id match in header block. The header block must contain the same slave device number that the enquiry sequence had, or this error will be generated. The ACI will be reset.
15	Not used
16	Bad character received when EOT expected. The ACI expects the Master to close all communication with an EOT character. This error is generated if this does not happen, and another character is received instead. Since the communication is already finished by this point anyway, the ACI will be reset.
17	A character other than an ETX was received at the end of a header block. This character must be delivered to the ACI immediately after the LRC character is sent.
18	Timeout on receiving the header SOH character. The Master control, after establishing communications with the slave during the enquiry sequence, has a limited time to send the following header block. See the table of timeout values. A timeout error always causes the ACI to reset.
19	Timeout of recieving the header ETX character. This timeout measures the time it takes to complete the transmission of the header block. Once the SOH has been received, the Master has only a limited time to transmit the whole header block. The ACI will be reset after this error.
20	Timeout on receiving the data block STX character. When the header block has been completed, and the Master has programmed a read cycle for the slave, the Master must deliver the first data block within a time period. Failing to do so will reset the ACI.

Table B - 2 ACI Error Codes

Error Number	Error Description, Cause, and recovery
21	Timeout on data block ETB/ETX character. Similar to error #19, the master has only a limited time to send the entire data block before this error occurs. The ACI will be reset.
22	Timeout on receiving a data block ACK/NAK character. After the slave has transmitted the data block to the Master, the Master must respond with the ACK or NAK character within the preset time limit, or the ACI will reset automatically.
23	Timeout on receiving the final EOT. The slave expects the Master to close thecommunication within a specified time limit. Failure to do so will set this error, and the ACI will be reset.
24	Unexpected EOT character. This error will be set if any expected control character turns out to be an EOT. The Master can close off the communication prematurely by sending an EOT instead of the accepted control code at any time in the sequence.
25	Not used
26	Bad special read code. The read code provided in the header is not supported.
27	Bad special write code. The write code provided in the header is not supported.
28	Bad special memory access code. The memory access code provided in the header is not supported.

Table B - 2 ACI Error Codes (cont'd)

B-9 ABORTING A COMMUNICATION CYCLE

Sending an EOT at any time that a normal control character is to be issued will be interpreted by the slave as a signal to abort the communication. This function may be required at times were an abrupt termination of the link is essential, for instance when attempting to quickly service another slave.

B-10 INTERFACE REQUIREMENTS

The ACI has been tested at baud rates up to and inculding 2400 baud. Faster baud rates may be possible, but are not recommended. Handshaking should be disabled, and parity should be turned off. A data byte size of 8 bits is required, with two stop bits. See the RAPL CONFIG command for setting up serial channel 1 to service the ACI link.

B-11 PREPARING THE ROBOT CONTROLLER FOR A COMMUNICATION CYCLE

The robot controller must be set up to accept the communication. This communication is executed through serial port 1. When this is active, then any robot references to device #1 will not respond, since this protocol takes priority when it is activated.

B-12 ACI MONITOR COMMANDS

The programmer can access the ACI software through a set of commands available at the terminal. These commands will allow the programmer to monitor any communications to and from the controller. The programmer can also enable or disable the ACI interface.

The commands are only available through the monitor level.

@@RI:
To force an initialization of the ACI interface. Any communication in sequence will be aborted. This may cause a loss of synchronization between the slave unit and the master control.

@@RN:
To select a slave device number for the robot controller. This value can be any value between 1 and 127. The value of 0 is not permitted.

@@RS:
Display the current status of the ACI software. The number of retries, the number of communication failures and the number of successful cycles will be displayed in a continuous fashion on the terminal device. This is a useful debugging tool.

@@RE:
Enable the ACI software. This will dedicate the serial port #1 to the ACI software. No other normal serial input will be allowed on this channel. Output is still allowed, but it will cause problems if an ACI cycle is concurrently running.

@@RD:
The ACI interface will be disabled. The serial port #1 will now be considered as any other programmable serial channel.

@@RH:
The last header which was read in will be displayed. This will indicate the type of the last (or current) conversation.

Character	Hex Code	Decimal Code
STX	02	02
ETX	03	03
ETB	17	23
EOT	04	04
SOH	01	01
ENQ	05	05
ACK	06	06
NAK	15	21

Table B-3 ASCII control codes for the ACI

Event	Timeout (in character times ++)
Enquiry timeout Timeout used to test for characters received during the enquiry sequence.	300
SOH timeout Time permitted between the recognition of a successful enquiry sequence, and the acceptance of the header SOH character	300
Header ETX timeout Time permitted between acceptance of the header SOH character, and the detection of the header ETX character.	300
ACK for data block Time permitted for waiting for an ACK or NAK from the master after the transmission of a data block from the slave.	300
STX data block timeout Time permitted for waiting for the data block STX from the master after a successful acknowledgment of either the header block, or the the previous data block.	300
Data block ETX/ETB timeout Timeout permitted for the complete data block, from acceptance of the STX to the recpetion of either the ETB or ETX character.	300
Final EOT timeout Time permitted between the acknowledgement of the last data block read and the final EOT response, or the last data block write acknowledge from the master, and its response to the slaves final EOT.	300

Table B-4 ACI timeout values

++ Timeouts are given in character times so that the baud rate does not affect the timeout strategy.

B-13 SPECIAL ACI ACCESS CODES

This chapter deals with the special functions available with the ACI version 2 software from CRS Plus. The extra functions located here explain how the system porgrammer can interface to the robot control to obtain information that was previously very difficult to obtain.

The functions explained here will permit the programmer to access robot I/O ports, as well as position and command registers.

The special codes are separated into special write codes (codes 40h to 4Fh) and special read codes (20h to 2Fh). These codes are placed in the access type specifier in the header block.

Special Write Codes

With the exception for code 47h, it is not necessary to specify an address with special write commands. It is important, however to specify the amount of data to be transferred. This is important in all cases.

40h

This special write command will return the most relevant addresses within the robot software. The addresses which are returned will enable the master control to access user memory directly, thereby permitting program upload and downloading. The use of this function will provide master control software which can operate independently from the robot software version. The buffer of information which is returned is in the form of memory pointers (4 byte, segment, offset format) . The lowest byte contains the least significant byte of the pointer. The addresses included in the buffer of information point to the items in the robot memory as described in table F-8, in Appendix F of the technical manual. The buffer of pointer information is the same as that used by the RAPL-BIOS facility, and is not reproduced here. With this special code, no address field need be specified, but since the pointer information consists of more than one full block, any portion of the data can be returned for use, starting with the first element.

41h

Send robot error codes. This special write code will send four bytes to the master indicating the health of the robot control. The buffer of four bytes includes, in this order:

1	AlarmStatus	00	indicates no alarmcondition
		01	indicates that an alarm exists
2	RAPL error code	NN	See the RAPL error manual
3	ACI error	00	No ACI error in effect
		01	ACI error is in effect
4	ACI error code	NN	See the ACI error list. This is the last ACI error detected.

42h

This command will force a write of the user input port images as captured by the RAPL operating system. The memory address in the header block need not be initialized, but the byte count must be specified. That is, it should contain a value of 8. See the technical description of the robot I/O space for a description of the inputs that can be read.

43h

This command will force a write of the user output port images as created by the RAPL operating system. The memory address in the header block need not be initialized, but the byte count must be specified. That is, it should contain a value of 8. See the technical description of the robot I/O space for a description of the inputs that can be read.

44h

This command will cause a write to the host computer of the position command data in the robot controller. The position command data exists as an array of 8 double integer values in memory. No address need be specified in the header block, but a data length of 32 bytes or less should be specified.

45h

This command will cause a write to the host computer of the actual position data in the robot controller. The actual position data exists as an array of 8 double integer values in memory. No address need be specified in the header block, but a data length of 32 bytes or less should be specified.

46h

This command will cause a write to the host computer of the end point of path data in the robot controller. The end point position data exists as an array of 8 double integer values in memory. No address need be specified in the header block, but a data length of 32 bytes or less should be specified.

47h

This command will transfer 8086 I/O input port values to the host computer. The I/O address is specified by the offset portion of the data address field in the header block. The segment field is not used. The number of bytes specified will result in a read of all ports from the specified address. This is a special command in that only one partial block of data can be transferred. The number of bytes in the last block will then correspond to the number of I/O ports to be scanned.

Special Read Codes

Special read codes allow the programmer to load specific areas of the robot memory with data.

20h

This special read command will load up to 128 bytes into the active command input buffer. In this way, and external computer can simulate a data terminal entry. Only text characters will provide a valid response. Any attempt to enter control codes as you would at the interactive level with the terminal will produce a RAPL error. No echoing will be noticed, since all normal echoes will go to the terminal.

Appendix B: Reprint of Technical Manual, Appendix G

TABLE OF CONTENTS

G-1 INTRODUCTION

RAPL software provides a list of pointers so that the user can access system parameters in RAPL using the ACI interface. See Appendix B for a complete description of the ACI interface. By providing this list of pointers, the programmer can be isolated from version changes in the RAPL software that could result in parameter address changes. This parameter list is accessible through special ACI write command 40 hex. A pointer to this list of pointers is also available at interrupt vector 60 decimal. The list of items is as follows.

Data found at the pointers indicated by this list are identified by type in the left hand margin: B for bytes, I for integers W for words, DI for long integers, R for real numbers, P for pointers and DW for long words comprise the data type specification. Arrays of types are shown with the appropriate array length. Remember, the items in the list are pointers. Some of these may point to the address of pointers (P type data) in the robot memory which then finally point to the data in question. A pointer to this list of pointers resides in interrupt vector 60 decimal.

G-2 NOTE FOR RAPL 5.00 AND LATER VERSIONS

Some system parameters have been replaced in this version. These obsolete parameters are indicated below. When the pointer list is read in the robot memory, these entries will appear as zero (NULL) pointers.

G-3 RAPL PARAMETER POINTER LIST

Data Type	Offset in List	Item Description (This pointer points to..)
P	0	the work space pointer - Pointer to the RAPL user space. Also the pointer to the RAPL program buffer. PROG_BUFF_PTR
W	1	the Program buffer size - the word register which tells us how many bytes of program storage has been allocated. PROG_BUFF_SIZE
W	2	the Program buffer count register - the word register which tells us how many bytes of program storage has been used. PROG_BUFF_CNT
P	3	the Program table pointer - the Pointer to the location of the program table in memory. PROG_TABLE_PTR
W	4	the program table size - the Word register which tells us how long the program table is in number of total program entry locations. PROG_TABLE_SIZE
P	5	the Variable table pointer - the Pointer to the variable table. VAR_TABLE_PTR
W	6	the Variable table size. Word register which tells us how big the variable table is in the total number of entries. VAR_TABLE_SIZE
P	7	the Location table pointer. Pointer to the location table. LOC_TABLE_PTR
W	8	the Location table size. Word register which tells us how long the location table is in the total number of table entries. LOC_TABLE_SIZE

Table G-1

Data Type	Offset in List	Item Description (This pointer points to..)
DI*8	9	the Calibration register pointer. Pointer to an array of 8 double integers which contain the robot calibration position. **CALIBRATION(8)**
B	10	the Calibrate position checksum byte. The byte addition of all valid calibration data for the 5 robot axes only. **CALIBRATE_CHECKSUM**
B	11	the 'robot is calibrated' check byte which is set after a calibration procedure has been run. **ROBOT_IS_CALIBRATED**
B*8	12	the Axis 'Begin Motion' array. An array of 8 bytes which signals each axis command generator to start a new motion. It is reset by the command generator when the path starts. **BEGIN_MOTION** ***** NOTE: DISCONTINUED FOR RAPL VERSION 5.00 AND LATER.**
B*8	13	the Axis 'Done' array. An array of 8 bytes which signals when each of the axes has finished a motion. Set by the command generator. **DONE** ***** NOTE: DISCONTINUED FOR RAPL VERSION 5.00 AND LATER.**
DI*8	14	the Actual Position array. An array of 8 double integers which are the absolute position registers of the robot feedback sub-system. Every 4 milliseconds, these values are updated with the incremental positions determined by the motor encoders. **POSITION**
DI*8	15	the Position Command array. An array of 8 double integers which represent the absolute position command to the robot motors. Any path generation algorithm can create these values, and the servo loop function will compare these registers to the actual position registers and will create the appropriate commands. **POSITION_COMMAND**
DI*8	16	the End point registers. An array of 8 double integers which define the end point of the current motion command in motor coordinates. If the robot is not moving, then these values equal the position command registers. **END_POINT(8)**
B	17	the 'Move type' parameter. This flag is set by the path generation selection. **MOVE_TYPE** The value can be: 1 - For joint interpolated moves, 2 - reserved 4 - manual mode. 5 - continuous path mode (and straight line moves)
B	18	the Number of active axes. Byte value with value from 1 to 8. **NUMBER_OF_AXES**
B*8	19	the Limp command array. Array of 8 bytes which selects the limp mode for each motor. In the limp mode, the servo loop function automatically overrides the command generation function, and will force the position command to be the same as the actual position registers. This supplies the motors with 0 voltage. **LIMP(8)** ***** NOTE: DISCONTINUED FOR RAPL VERSION 5.00 AND LATER.**

Table G-1

Data Type	Offset in List	Item Description (This pointer points to..)
DW	20	the Real time ticker. A double word quantity that is incremented each I/O update cycle (approximately 30 milliseconds). The value of this counter is zeroed only at a teach start situation, so it can accurately keep track of the total time that the controller has been switched on. REAL_TIME
B	21	the Hold request flag byte. A byte value that, when set to 1, will stop a RAPL path generation, and will cause all DONE flags to register a complete path. The end point registers of the path will be updated to the commanded position of the axes when all axes have been brought to a stop. In joint interpolated mode, the axes are decelerated to a stop. In straight line, the axes are stopped immediately. The hold request is reset when a new move is commanded. HOLD_REQUEST
B*8	22	the Axis Lock out array. Pointer to an array of 8 bytes which will selectively lockout individual motors from any subsequent motion commands. LOCKOUT(8) *** NOTE: DISCONTINUED FOR RAPL VERSION 5.00 AND LATER.
W	23	the Millisecond counter. A word quantity which is used to derive a millisecond delay from the system I/O clock update. Used to time the DELAY function in RAPL. MILLISECOND_COUNTER
R*8	24	the Acceleration array. An array of 8 reals which determine the acceleration of each motor during a joint interpolated motion. ACCELERATION(8)
R*8	25	the Transmission Ratio array. An array of 8 real numbers which equates the measurement units of the joint or motor with the corresponding motor pulse resolution. It is dangerous to change the first five values in this array, as it will change the robot transformation. The @XRATIO command changes the other three.
B	26	the Alarm Number. A byte value that contains the last RAPL error code to be generated. ALARM_NUMBER
W*8	27	the Rotary Resolution array. An array of 8 integers which specifies the motor encoder resolution in encoder lines per motor revolution. ROTARY_RESOLUTION(8)
R*6	28	the Tool Transform - an array of 6 real numbers that specify the current tool transform value. TOOL_TRANSFORM

Table G-1 (Cont'd)

Data Type	Offset in List	Item Description (This pointer points to..)
Struct	29	the Input block structure - the RAPL character input buffer area. The input buffers are specified in the following format: **input_block(3) structure (** **buffsize byte,** **in_ptr byte,** **out_ptr byte,** **buffer(128) byte)** The first two structure records keep track of data input from the two serial ports. The third is used as an intermediate buffer when executing from a program. 'buffsize' determines the size of the input ring buffer, up to 128 bytes. 'in_ptr' points to the next array element available for serial input, and 'out_ptr' marks the next character that can be released from the buffer, to be used by RAPL.
B	30	the Alarm status. A boolean byte value which specifies whether or not a RAPL error has occurred. The byte is reset when a new command is attempted. **ALARM_STATUS**
R*64	31	the Motor to Joint transform matrix. An 8 by 8 real matrix which transforms the motor coordinates into the joint coordinates. Since the robot structure is not mechanically de-coupled, several motors may have to move in order to provide a single joint motion. In a matrix equation, {j} = [J]{m}, where {j} is the joint row matrix, and {m} is the joint motor matrix.
R*64	32	Joint to motor transformation matrix. An 8 by 8 real matrix which determines motor coordinates from joint coordinates. It is the matrix inverse of item #31 above. In a matrix equation, {m} = [JJ]{j}, where {j} is the joint row matrix, and {m} is the joint motor matrix. The JJ matrix is the inverse of the J matrix identified by item #31.
W	33	the Work space size. A word value which specifies the total user area available. The user area can be represented as an array in PL/m-86 as follows. (User_Area based work_space_ptr)(work_space_size) byte;
B*3	34	the Extra axes calibrate checksums. A 3 byte array which contains the checksums for the calibrate position of each extra axis. **XCALIBRATE_CHECKSUMS(3)**
W	35	the Reserved work space size. This word contains the number of bytes which are to be set aside from the routine RAPL user memory. see section F-9 for a description of the RAPL user space. This memory area can then be used for 8086 machine code programs that can be loaded in from the terminal, or through the ACI. This area is also used for CTPATH data. **RESERVED_WORK_SPACE_SIZE**
B*9	36	the Digital Input Image buffer. A buffer of 72 bits which map the robot input space. See figure F-5. **DIGITAL_INPUT**

Table G-1 (Cont'd)

Data Type	Offset in List	Item Description (This pointer points to..)
B*7	37	the Digital Output Image buffer. A buffer of 56 bits which map the robot output space. Since the robot digital output ports cannot be read, then it is up to the software to remember what was issued to the output port before updating single bits of it, so that the remaining bits can be left as they were. See figure F-4. DIGITAL_OUTPUT
B	38	the Default character I/O device. The default character device is either 0 or 1 or 2. 0 and 1 represent the two serial channels. 2 represents a user program which is being used for character input. The default device is 2 only when using the controller without a terminal in the AUTOSTART mode. When a program is executing, the default device represents the serial channel that is to be used for any user input and output. DEFAULT_DEVICE
B	39	the Device Select. Identifies the current device being used for character input and output. DEVICE_SELECT
B	40	the Syntax Help function. This byte boolean value determines whether or not the Syntax building function will be activated or not. HELP_ON
B*8	41	the teach name template. An array of 8 bytes which contain the current teach name. Only the first 5 characters are used when the final point name is formulated. The rest is padded with underscore characters. TEACH_NAME_TEMPLATE
B	42	the teach name count. The byte value which is encoded into the final three character positions of the teach name when the point is finally stored. TEACH_NAME_COUNT
B	43	the Program pause flag. When a program is executing, setting this byte to a '1' value will stop the program after the next line is decoded. PROGRAM_PAUSE
B	44	the homing complete flag. This flag is set when the robot has been homed after a power up. Do not confuse this with item #11, which indicates whether or not the robot has been calibrated. This flag must be set in order that all move functions that access points will work. ROBOT_IS_HOMED
B*128	45	the string buffer. The string buffer is a buffer 128 bytes long, that is divided evenly into 4 32 byte areas. These correspond to the 4 string variables that are allowed by RAPL version 3.50 or later. STRING_BUFFER
B	46	the loss of feedback/collision detection byte. This byte indicates the status of this function. A boolean true means that the function is turned on. LOFB_CHECK
B	47	the flag indicating whether or not a program is completed or not. PROGRAM_COMPLETED
W	48	the word counter that indicates the number of repetitions that the current program has completed. PROGRAM_REPETITIONS

Table G-1 (Cont'd)

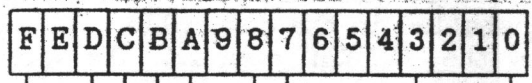
Data Type	Offset in List	Item Description (This pointer points to..)
W	49	the word register which indicates the number of repetitions of the program that have been requested. PROGRAM_REPEAT_LIMIT
B	50	the flag which indicates that an infinite looping of the current program has been requested. PROGRAM_LOOP_FOREVER
R*250	51	the 1000 byte buffer which contains the data for the current straight line move. Also used to contain the path parameters for a CPATH. STRAIGHT_LINE_BUFFER
B	52	the flag which indicates whether or not the trigger table has been enabled. TRIGGER_TABLE
B	53	the flag which indicates whether or not editing is permitted. EDIT_ENABLE
I	54	integer containing the global robot speed value. ROBOT_VELOCITY
R*6	55	the first element of the robot base shift registers. The shift registers are arranged in this order: X_BASE_SHIFT, Y_BASE_SHIFT, Z_BASE_SHIFT, O_BASE_SHIFT, COS_O_BASE_SHIFT, SIN_O_BASE_SHIFT
B	56	the flag which indicates whether or not a continuous path has been interrupted with the HALT command, typically in an ONSIG situation. CPATH_INTERRUPTED
DI*8	57	the 8 double integer registers used to save the path position were the robot was interrupted from its continuous path. HOLD_POSITION
B	58	the register which describes the configuration of the robot arm for all coordinate transformations. CONFIG
P	59	the pointer to the knots array for the current CPATH or straight line move. P_KNOTS
P	60	the pointer to the time array for the current CPATH or straight line move. P_TIME
P	61	the pointer to the acceleration array for the current CPATH or straight line move. P_ACCEL
P	62	the pointer to the temporary array for the current CPATH or straight line move. P_TEMP
P	63	the pointer to the knots array for the current CTPATH or straight line move. TP_P_KNOTS
P	64	the pointer to the time array for the current CPATH or straight line move. TP_P_TIME
P	65	the pointer to the acceleration array for the current CPATH or straight line move. TP_P_ACCEL
P	66	the pointer to the temporary array for the current CPATH or straight line move. TP_P_TEMP
R	67	the TIMER which increments as the continuous path proceeds. This timer registers is compared to the knot timer array in order to determine which path segment the path is currently in.

Table G-1 (Cont'd)

Data Type	Offset in List	Item Description (This pointer points to..)
W	68	the PATH_SEG which determines which path segment the path is currently in.
W	69	the number of knots that are in the CTPATH. TP_N_KNOTS
W	70	the number of axes which are included in the current CTPATH. TP_N_AXES
W	71	the number of knots that are in the last CPATH. N_KNOTS
W	72	the number of axes that are included in the last CPATH. N_AXES
B	73	the flag which indicates whether a CTPATH is loaded and ready to go. PATH_LOADED
W	74	the register which determines the maximum number of knots which will be used in the calculation of all straight line moves. ST_PATH_KNOTS
NOTE: THE FOLLOWING ARE AVAILABLE FOR RAPL VERSION 5.00 AND LATER		
B*8	75	8 character buffer which identifies the first location in a CTPATH, used to send the robot under joint interpolated mode to the start of the path. CPATH_START_NAME
B	76	Enable NULL positioning mode. In this mode, the finish criterion is not complete until each axis has moved to within a preset tolerance band of the end point. NULL_ON
B*8	77	The tolerance band for NULL positioning is set for each axis. Byte values allow for 127 pulse positioning error. NULL_TOLERANCE
B	78	Enable the trigger events during the next CPATH, CTPATH. TRIGGER_ENABLE
B	79	Enable outputs to be issued. Disabling the outputs is often useful when debugging a program without wanting to switch external devices. OUTPUT_ENABLE
B	80	Enables the ONSIG condition without changing the setup status of the command. ONSIG_ENABLE
B	81	Enable the arm power watchdog. Disabling will mean that the arm power will be switched off within 16 milliseconds. ARM_ENABLE
B	82	Enable the TRACE mode display. TRACE_ENABLE
B	83	Enable the FLASH function on the teach pendant LED. FLASH_ENABLE
B	84	Enable the check for loss of arm power. ARM_POWER_CHECK_ENABLE
B	85	Enable the teach mode with the prescribed template name and count number. TEACH_ENABLE
B	86	Enable the manual mode, under the currently active manual mode type. MANUAL_MODE_ENABLE
B	87	Enable the cartesian velocity continuous path control. This flag will ensure constant cartesian velocity control. CARTESIAN_VELOCITY_ENABLE
B	88	Enable the SLEW velocity mode for joint interpolation. This mode is faster than SPLINE mode, where the velocity shape has a parabolic characteristic. TRAPEZOIDAL_VELOCITY_ENABLE

Table G-1 (Cont'd)

Data Type	Offset in List	Item Description (This pointer points to..)
B	89	Stop any current motion and put the command generator in a suspended mode. **FEED_HOLD_ENABLE**
P	90	Pointer to the available user memory. This is a pointer to the reserved space. **MEMORY**
W*8	91	Array of 8 words which defines the status of the controller axes. This array is organized as a bit array. It replaces the old byte flag arrays: DONE, BEGIN_MOTION, LIMP, LOCKOUT. **AXIS_STATUS**. The bit allocation of this word array is shown below;



AXIS OK. This bit will be set to 1 when the axis card is healthy.

LIMP. bit will be set to ´1´ when the axis is under the limp mode.

LOCKOUT. When this bit is 1, the axis will be excluded from any future move commands.

AXIS ERROR. When this bit is set, the axis currently under test has an error condition.

MOTOR FAULT. When this bit is set, the axis card has detected a motor fault condition, and this condition has been recognized by the RAPL operating system

HOMED. When this bit is 1, the axis has been homed.

CAL. When this bit is set to 1, the axis has been calibrated.

BEGIN MOTION. When this bit is set, the RAPL command interpreter has requested a path start. This bit will be reset when the path does start.

DONE. When this bit is set to ´1´, there is no path in progress on the current axis.

NOTE: All unused bits are reserved by the RAPL operating system. Also, modification of these registers is strictly not recommended, as they are constantly updated by the RAPL communication software with the smart axis cards. Use these registers as status indicators only.

Table G-1 (Cont´d)

G-4 RAPL USER MEMORY ALLOCATION

Please note that addresses of pointers to items in the robot user memory can be determined from the pointer list described in Table G-1. All references to "the table" mean Table G-1.

The RAPL user memory is partitioned into 5 areas. These are listed according to memory order. User memory is pointed to by the PROG_BUFF_PTR parameter, item #0 in the table. Since this is the bottom of user memory, this pointer is also referred to as the work_space_ptr parameter.

The reserved memory area is the first item in the user memory. It can be reserved for any purpose so that RAPL program, location and data operations cannot affect it. If the expanded memory option is installed in the controller, This memory can be used to store path data generated by the "CTPATH" command. The work_space_ptr parameter is adjusted to point to the next byte beyond this area. All other RAPL user memory items are addressed relative to the work_space_ptr parameter.

Program Buffer

The program buffer, which is a contiguous array of bytes is stored next. Each program is stored immediately after one another. A '$' character separates one program from the next. The program buffer is addressed by the work_space_ptr parameter (#0) since it appears first in memory.

The program table is next. It can best be described by the following 'C' structure reference:

Program Table

```
struct p_table
     {
     char name[8];
     int index;
     int checksum;
     } prog_table_ptr[prog_table_size];
```

And the pointer to the program table (item #3 in the table) can be calculated to be:

```
prog_table_ptr = &work_space_ptr[prog_buffer_size];
```

Each entry into the program table takes 12 bytes. The PROG_PTR element defines the starting point in the program buffer for that particular program. The actual location of the program is then at:

```
prog_address = &work_space_ptr[prog_table_ptr[i].index];
```

The checksum byte is the addition of all the program bytes up to the final '$' character. The checksum is only a byte length addition, so the upper byte of the checksum word is undefined and should be set to '0'.

Location Table

The location table is next. It can be described by the following 'C' structure reference:

```
struct l_table
     {
     char name[8];
     float data[8];
     int checksum;
     } loc_table_ptr[loc_table_size];
```

For a precision point, the location table would be described as:

```
struct l_table
     {
     char name[8];
     long pp_data[8];
     int checksum;
     } loc_table_ptr[loc_table_size];
```

Thus any entry in the location table requires 42 bytes of data: an 8-byte name, 8 real or long integer fields (4 bytes each) and a checksum word. The checksum is the addition of all the bytes in the particular location. The checksum is only a byte length addition, so as with the program table, the upper byte of the checksum word is undefined.

The location table pointer can be calculated to be:

```
loc_table_ptr = &prog_table_ptr[prog_table_size];
```

Variable Table

The variable table is stored next. It can best be described by the following 'C' structure reference:

```
struct v_table
      {
      char name[8];
      float data;
      int checksum;
      } var_table_ptr[var_table_size];
```

An entry in the variable table requires 14 bytes: an 8 byte field for the name, a 4 byte real value, and the two checksum bytes. As with the program and location table entries, the checksum is the addition of all the bytes in the particular location. The checksum is only a byte length addition, so the upper byte of the checksum word is undefined.

The position of the variable table can be calculated as follows:

```
var_table_ptr = &loc_table_ptr[loc_table_size];
```

Trigger Table

The trigger table is an item in the RAPL parameter memory. It is a structure of 8 elements called "triggers" which define programmed output events which will occur as the robot arm moves through locations (knots) during a CPATH or a CTPATH motion. The trigger table saves the location name of the event, as well as the output number and desired state of the output at the event. If the output number field is a zero, then no trigger event is loaded. A negative trigger number indicates that the selected output will be triggered to a LOW state. A positive number will trigger the output to a HI state. Valid output numbers range from 1 to 40. The trigger table can be defined as the following structure:

```
struct trig_table
      {
      char name[8];
      int output_number_and_state;
      } trig_table[8];
```

Thus an entry in the trigger table takes 10 bytes: the location name field consisting of an 8-byte entry and the integer state value. No checksum is provided as this data is not retained when the robot controller power is removed.

The data in the trigger table is used in execution of the CTPATH or CPATH commands. As the path data is generated, the trigger table is scanned. If a location is called for which a valid entry exists in the table, an entry is made in the path data for that knot.

During execution of the path, RAPL scans two word values at each knot. Each contains a map of the lower 16 user output line numbers. A "1" in a bit position in the first word represents an output whose state changes at that knot. The second word contains the desired state of that output after the change. The change is made at I/O scan rates so that the state will change within approximately 40 milliseconds from the time the commanded position of the robot reaches the knot.

Continuous Path Memory Allocation

The CTPATH command utilizes the reserved memory to store most of the path information. The remainder of information is saved in RAPL parameters pointed to by: TP_N_AXES, TP_N_KNOTS, TP_P_KNOTS, TP_P_TIME, TP_P_ACCEL, TP_P_TEMP, PATH_LOADED. These parameter pointers are shown in Table G-1 (items #63 to #66, #69 and #70). A more detailed description of these parameters can be found in the detailed discussion of the CPATH control strategy in the RAPL manual.

The first step in accessing the CPATH data is to determine the pointer to the reserved memory. For RAPL version 5.00 and later, this item is #90 in the pointer list. This pointer is not immediately accessible by any item in the pointer list for RAPL version 4.99 and earlier, but a simple operation can derive it, since it is accessible by taking the pointer to the workspace (item #0) and by

subtracting the reserved work space size (item #35) which would act as a negative offset from the start of the user memory. The following code extraction, written in 'C,' will do just that;

```
.
.
char *reserved_ptr, *work_space_ptr;
int reserved_space_size;
int *tp_n_knots, *tp_n_axes;
.
.
reserved_ptr = &work_space_ptr[-reserved_space_size];
.
.
```

Once the starting address to the data has been determined, the data elements for continuous path are stored in the following order;

float tp_pknots[n_axes][n_knots];
The knots array is located at the start of reserved memory. The knots array is actually a two dimensional array, bounded by the number of knots and the number of axes as the two dimensions. The total array size is therefore; n1 = #knots * #axes * 4, in bytes.

float tp_ptime[n_knots]
The time array is a float array that marks the elapsed path time at each knot.

float tp_paccel[n_axes][n_knots]
The acceleration array contains real numbers marking the acceleration of each axis at each knot position.

float tp_ptemp[n_knots]
This array is used for temporary storage during the calculation of the path parameters. It is then used to store the trigger setup information for the path. When storing the trigger information, the format of the data are two 16 bit word masks for each knot. The lower order word contains the bits which determine the outputs that are to be controlled at that particular knot. A 1 bit will indicate that the corresponding output will be controlled. The higher order word determines the state to which the designated outputs will be placed. A 0 bit will turn an output on, and a 1 will turn it off.

With this information, all of the pointers can be formulated, as the following code extraction demonstrates.

```
.
.
.
float *tp_pknots, *tp_paccel, tp_ptemp;
.
.
tp_pknots = reserved_ptr;
tp_ptime = &(tp_pknots[n_knots][n_axes]);
tp_paccel = &(tp_ptime[n_knots]);
tp_ptemp = &(tp_paccel[n_knots][n_axes]);
.
.
```

When it is time to execute a CPATH, using the GOPATH command, the pointer values and all of the related parameters are loaded into the working registers (items #236 to #248, #284 and #288) if the path loaded flag indicates that there is path data available.

ACI Library Procedures:

The following is a list of library routines included in the TURBO PASCAL ACI library. All data types and global variables requiring definition are also included. A simple example routine is also included with the listings.

data_transfer (slave_device: byte;
MemType: byte;
MemOfs, MemSeg: integer;
AccessType: char;
bytes: integer);

This routine coordinates a data transfer. It is the basic call made for any transfer of data using the ACI. Depending on whether a read or write is requested, it calls one of the following routines: data_read_from_slave or data_out. Each of these two routines call the Init_comm, header and send_header routines to initiate the communication. All of these routines call upon the lower level communications procedures listed here.

data_read_from_slave (n_bytes: integer);
Coordinate the transfer of data from slave.

data_out (n_bytes: integer);
Coordinate the transfer of data to the slave.

Init_comm (slave_no: byte);
Initiate communications with a slave. Calls the ENQout procedure.

ENQout (slave_no: byte);
Send an enquiry string.

header (slave_no, MemType: byte;
MemOfs, MemSeg: integer;
RorW: char;
message_length: integer);

Generate the Header Block in preparation for transmission.

send_header (slave, Memtype: byte;
MemOfs, MemSeg: integer;
RW: char;
data_length: integer);

Send the Header Block. Calls the previous routine.

error_state (err: byte);
Set error status.

turn_on_interrupt;
Turn on the serial interrupt state.

turn_off_interrupt;
Turn off the serial interrupt state.

clean_aux:
Clean the input buffer and support.

send_aux (cha: char);
Send a character to the auxiliary port.

int_srv;
Service the serial on an interrupt basis.

init_8250 (rate: integer);
Set the baud rate for the RS-232 (COM1 device).

string_out (str: txt);
Send a s\tring to the aux device.

N_string_out (str: txt; N: byte);
Send a string of length N to the aux device.

Functions:

clr_to_sen: boolean;
Check for transmit ready.

aux_ready: boolean;
Test for aux port with a waiting character.

GET_aux: char;
Get a character from the aux port.

string_in (N: byte): txt;
Receive a string of length N from the aux device.

ACI Library Procedures:

The following is a list of library routines included in the TURBO PASCAL ACI library. All data types and global variables requiring definition are also included. A simple example routine is also included with the listings.

data_transfer (slave_device: byte;
MemType: byte;
MemOfs, MemSeg: integer;
AccessType: char;
bytes: integer);

This routine coordinates a data transfer. It is the basic call made for any transfer of data using the ACI. Depending on whether a read or write is requested, it calls one of the following routines: data_read_from_slave or data_out. Each of these two routines call the Init_comm, header and send_header routines to initiate the communication. All of these routines call upon the lower level communications procedures listed here.

data_read_from_slave (n_bytes: integer);
Coordinate the transfer of data from slave.

data_out (n_bytes: integer);
Coordinate the transfer of data to the slave.

Init_comm (slave_no: byte);
Initiate communications with a slave. Calls the ENQout procedure.

ENQout (slave_no: byte);
Send an enquiry string.

header (slave_no, MemType: byte;
MemOfs, MemSeg: integer;
RorW: char;
message_length: integer);

Generate the Header Block in preparation for transmission.

send_header (slave, Memtype: byte;
MemOfs, MemSeg: integer;
RW: char;
data_length: integer);

Send the Header Block. Calls the previous routine.

error_state (err: byte);
Set error status.

turn_on_interrupt;
Turn on the serial interrupt state.

turn_off_interrupt;
Turn off the serial interrupt state.

clean_aux:
Clean the input buffer and support.

send_aux (cha: char);
Send a character to the auxiliary port.

int_srv;
Service the serial on an interrupt basis.

init_8250 (rate: integer);
Set the baud rate for the RS-232 (COM1 device).

string_out (str: txt);
Send a s\tring to the aux device.

N_string_out (str: txt; N: byte);
Send a string of length N to the aux device.

Functions:

clr_to_sen: boolean;
Check for transmit ready.

aux_ready: boolean;
Test for aux port with a waiting character.

GET_aux: char;
Get a character from the aux port.

string_in (N: byte): txt;
Receive a string of length N from the aux device.

```
00001 /*
00002 ------------------------------------------------------------------------
00003
00004 Module : ACI00.C
00005 ------
00006
00007 Description :
00008 -----------
00009   This module will provide all of the ACI protocol support routines. The
00010   requirements for the ACI implementation can be divided into four procedure
00011   categories:
00012   i)   Hardware Implementation, (the list below specifies which routines must
00013        be supplied by the programmer for different implementations). These
00014        procedures are found external to this one.
00015   ii)  Utility routines,
00016   iii) Protocol level. These are the procedures which control and enforce the
00017        ACI protocol standard,
00018   iv)  Application Interface. These procedures provide quick and easy integration
00019        of the ACI library to an existing application software package
00020
00021   Type i) procedures are defined in IBM00.C, or must be provided by the system
00022   programmer.
00023
00024
00025 Additional Information
00026 ----------------------
00027  The RAPL Technical reference manual, noteably Appendix B discusses the ACI
00028  protocol in detail. This information is not reproduced here.
00029
00030
00031 Global Procedures Defined :
00032 -------------------------
00033
00034 Utilities
00035 ---------
00036
00037 void aci_nstrout( str , n )
00038 char str[];
00039 int n;
00040    Send a string of characters out. The string is pointed to by 'str', and the
00041    string is 'n' bytes long.
00042
00043 void aci_strin( tempstr,n )
00044 char tempstr[];
00045 int  n;
00046    Read a string of characters in from the serial interface. The number of characters
00047    to be expected is 'n', while the destination character array is pointed to
00048    by 'tempstr'.
00049
00050 int lobyte(i)
00051 int i;
00052    Will return the low byte of the integer.
00053
00054 hibyte(i)
```