Command/Response Protocols and Concurrent Software

Final Report

June 24, 1987

NASA GRANT NAG-1-670

W. L. BYNUM

Principal Investigator

Department of Computer Science

The College of William and Mary

Williamsburg, VA 23185

Project Monitor

Nancy O. Sliwa

Mail Stop 152D

NASA Langley Research Center

Hampton, VA 23665

I was fortunate to have the assistance of two graduate students from William and Mary, John McManus and Mike Mansfield, from June, 1986 to December, 1986. Although neither of these students received any stipend, their contributions to the grant were significant.

This report will be organized as an enumeration of what each of the three of us accomplished during the grant period.

John McManus

John developed a graphics package for the Symbolics in ZetaLISP and used it to implement a program to provide on the Symbolics a graphical simulation of a moving PUMA manipulator. He constructed the program in a modular way so that it could be used to simulate cooperating PUMA manipulators. John is a highly motivated, intelligent student who required little supervision or assistance.

Mike Mansfield

Mike's contribution consisted of splitting the existing program for controlling the parallel jaw end-effectors into separately assembled parts. In doing so, he discovered pecularities in the 8051 assembler and linker system that showed that the goal of separate assembly was impractical. Although this result was negative in nature, it was very valuable to the grant, because it was something that I would have had take the time to discover myself.

Bill Bynum

Working with Bob Glover, I integrated the existing controller program for the parallel jaw end-effectors into the operator interface. This places the end-effectors under software control. As a separate program, I modified previously written code to provide a graphical simulation of the end effector on the VS11 graphics terminal that is driven by changes made to the REALTIME data structure — that is, the graphical image faithfully mirrors the jaw position of the end-effector selected in the operator interface.

There was no source that corresponded to the object code of the existing controller program. Using existing source files, I hand-disassembled the object code of the existing controller program for the end effector to produce a source listing for the program. The source files that comprise the program are:

ISRL6::SYS$USERDISK[WILLMAR.GRIPPER]DMHAND.51
ISRL6::SYS$USERDISK[WILLMAR.GRIPPER]DMSYSTEM51
ISRL6::SYS$USERDISK[WILLMAR.GRIPPER]DMHANDMAC.51
ISRL6::SYS$USERDISK[WILLMAR.GRIPPER]DMINITSE.51
ISRL6::SYS$USERDISK[WILLMAR.GRIPPER]DMSENSORS.51
ISRL6::SYS$USERDISK[WILLMAR.GRIPPER]DMENCODE.51
ISRL6::SYS$USERDISK[WILLMAR.GRIPPER]DMERRORS.51

The file **ISRL6::SYS$USERDISK[WILLMAR.GRIPPER]DMHAND.LIS** is a listing of the assembly of the program. The object code produced by this program source is identical to the existing object code.

I wrote the code to incorporate the parallel jaw gripper into the CRITTER behavior network.

After Mike Mansfield completed his work in December, 1986, I rewrote the existing program to control the parallel jaw end-effector. The new program corrects deficiencies and errors in the existing program and provides significantly more functionality, reliability, and interactive debugging capability than the existing program. The new program has been thoroughly documented, both by comments in the program source and by a separate written report. A copy of that report is attached as an appendix .

A Parallel Jaw End-Effector Controller

Part of the Final Report

for NASA Grant NAG-1-670

W. L. Bynum

June 1987

Introduction

The purpose of this report is to document a new version of the program to control the parallel jaw gripper used in the Intelligent Systems Research Laboratory. This parallel jaw gripper was designed at the University of Rhode Island. The hardware and software that support the parallel jaw gripper were developed at NASA/Langley Research Center under the leadership of Jim Wise and Wallace Harrison.

The program that is the subject of this report follows the basic outline of the original program designed by Wallace Harrison. Others who have made substantial contributions to the development of the controller software are Jim Wise, Sixto Vasquez, Don Soloway, Karin Cornils, Gene Bahniuk, Frank Primiano, Taumi Daniels, and Mike Mansfield. Those responsible for the development and maintenance of the parallel jaw gripper hardware have been Jim Wise, Kevin Barnes, Art Hayhurst, Alan Williams, and Dave Poskevich.

The program now being used to control the parallel jaw end-effector has been remarkably stable and relatively free from errors, and has not been modified for a long period of time. However, changes to this program were required for the following reasons.

1. There is little documentation for the old program – how it works, how it was designed. The actual source for the old program appears to be lost. Although there are several versions of the source that produce object code that is close to the object code now being used, none of the versions are an exact description of the old program.
2. The old program is inflexible and cannot be easily modified to deal with end-effectors having different drive motor worm gear ratios. The characteristics of different drive motors and shaft encoders have been found to vary widely, and the old program has no built-in mechanism with which the user can determine the number of shaft encoder counts required to move the jaws from wide open to closed. This constant is used by the FORTRAN controlling program in the REALTIME system to convert back and forth between jaw openings in mm. and encoder counts.
3. There is no provision for interactive debugging in the old program, so that isolation of errors and the determination of their cause is a lengthy, time-consuming process. Changing one of the parameters of the program involves creation of a new object file with the assembler and linker, transferring this file to an EPROM, and installing the new EPROM in the CPU board of the controller cage.
4. With the old program, it is possible for the user to get the program in a state where maximum drive current is being steadily applied to the drive motor even though no jaw movement is occurring. Although this does not harm the drive motor, the heat that the drive motor generates is deleterious to other heat-sensitive components located in the end-effector housing, such as the force-torque sensor.
5. The part of the old program that presents the sensor information contains an error that results in the loss of one of the sensor values. There is no documentation of the arrangement of sensor bits.

The new version of the controller program corrects these deficiencies. The following changes have been made.

1. This paper, along with the extensive commenting in the program source, provides a thorough description of the program.

2. A command has been added to the program with which the user can change the speed of the drive motor to accommodate different worm gear ratios. Another command has been added with which the user can determine the number of encoder counts that correspond to the total jaw travel.
3. An interactive monitor has been added to the program with which a user can examine and modify contents of the on-chip RAM and suspend and resume the clock/timer interrupts. This allows the user to change the system variables that control the program. Because the program locations given in an assembly listing are shown in hexadecimal, the mode of numeric user interaction with the program has been changed from octal to hexadecimal.
4. A feature has been added to the program so that a state in which a consistent current being applied to the drive motor when no jaw movement is occurring can be detected and terminated.
5. A monitor command to debug the sensor system and the associated LEDs on the front panel of the controller cage has been implemented. A description of the location and meaning of the sensor bits appears in this document and has been added to the commenting in the program source.

The next section of this paper contains a brief description of the parallel jaw end-effector hardware and the Intel 8031 processor that is used to control the end-effector. The third section of the paper gives a general overview of the controller program, and the final section contains a complete description of the program's structure and design. There are three appendices: a memory map of the on-chip RAM, a cross-reference listing of the self-scheduling routines, and a summary of the top-level and monitor commands.

Features of the Gripper Controller Hardware

The reader may find it helpful to have a brief description of the technical features of the parallel jaw gripper and its associated hardware. The gripper has been described in detail elsewhere [1]. The reader interested in a detailed desciption is referred to that source.

Parallel jaw movement is achieved on the gripper through a four-arm linkage, two arms per jaw. The linkage for each jaw has an attached sector gear that is driven back and forth by a central worm screw. There is a shaft encoder attached to the motor driving the worm screw that is used to provide an indication of the position of the jaws. Each jaw has four *proximity* detectors, each implemented by three light-emitting diodes. In the three-diode set, two of the diodes are used as emitters and one diode is used to sense light reflected from the emitting diodes. There is a *crossfire* detector to determine when an object is between the jaws, there is an *overload* detector to determine when the maximum allowable load on the gripper has been exceeded, and there is a *limit* detector to sense when the jaws are open to their widest position.

The program that controls the gripper is written for a microprocessor in the INTEL 8051 family. The two members of this family that fit into the design of the CPU board used in the controller card cage are the 8031 and the 8751. These two processors have identical instruction sets. The only difference between them is the presence of 4-Kbytes of

on-chip EPROM on the 8751 for storing program code, whereas the 8031 has no on-chip EPROM and depends solely on the external program memory provided by the two 2-Kbyte EPROM sockets on the CPU board. Both processors share the following features [2,3]:

- 8-bit CPU
- on-chip oscillator and clock circuitry
- four 8-bit I/O ports
- 128-byte on-chip data memory (RAM)
- 64-Kbyte address space for external data memory (RAM)
- 64-Kbyte address space for external program memory (ROM or EPROM)
- two 16-bit timer/counters
- a five-source interrupt structure
- a full-duplex serial port

The circuitry on the CPU board limits external program memory to two 2-Kbyte EPROM sockets. At present, only one of the two sockets is needed, since the controller program is slightly shorter than 2-Kbytes. External data memory is used solely to address the I/O ports of the 8155 chip for sensor data and the port to the digital-to-analog converter controlling the jaw drive motor. The only data memory (RAM) accessed by the program is the 128 byte on-chip RAM.

The interrupt structure for the 8051 family has five levels, as shown in the following display:

interrupt	use	priority
external request 0	increasing encoder counts	highest
internal timer 0	system ready queue	
external request 1	decreasing encoder counts	
internal timer 1	unused	
internal serial port	terminal I/O	lowest

The encoder connected to the motor driving the worm gear is connected to the 8031 processor so that a rotation of the encoder in the direction of increasing encoder counts causes an "external request 0" interrupt and a rotation of the encoder in the direction of decreasing encoder counts causes an "external request 1" interrupt. The handlers for these two interrupts increment or decrement the memory location in the on-chip RAM that is used to hold the actual encoder count.

The "internal timer 0" is used as an autoload clock timer. A fixed value is loaded into the 16-bit timer 0 register. The register is decremented in each clock cycle of the processor. When the register reaches 0, the timer-driven interrupt occurs and the register is reloaded with the fixed value. This is a periodically occurring interrupt that is used to drive the software that maintains the operating system ready queue. How this is done will be discussed in the next section.

Another hardware feature that affects the design of the software is the method in which the worm screw motor is driven. The motor is driven through the 8-bit digital-to-analog converter port that is located in the external memory space of the 8031 processor. Values at this port indicate the amount of current being sent to the drive motor. These values are represented using excess-128 notation. That is, the value corresponding to the current sent to the drive motor is 128 more than the actual value. For example, the *null*, or quiescent, current corresponds to a port value of $128 = 0 + 128$. The maximum *positive* current corresponds to a port value of $255 = 127 + 128$, and the maximum *negative* current corresponds to a port value of $0 = -128 + 128$.

The value written to this port is latched, which has the effect that the motor continues to be driven with the last value written to the port until overwritten with a different value. This feature, along the wiring of the CPU circuit board used in the controller card cage, makes use of the 8031 processor preferable to the 8751 processor. When the "reset" switch is closed, the 8751 processor begins executing at location 0000 of its internal EPROM unless one of its control lines is raised. This control line is not raised on the CPU circuit board used in the controller card cage, because the board was designed for the 8031 CPU. When the 8751 processor is used and no program code is stored in its internal EPROM, there is a brief delay while the processor "executes" the 4 Kbytes of this internal EPROM before going to the controller program stored in external EPROM. In contrast, the 8031 processor goes to the external EPROM immediately, since it has no internal EPROM. During a reset of the processor, the bit pattern at the digital-to-analog port drives the worm gear motor until the jaw controller program stored in the external EPROM can take control and null the current. The result is a brief "spasm" of the jaws occurring at a reset when the 8751 is used on the CPU board that is entirely absent when the 8031 processor is used. Such a spasm could be undesirable if a processor reset became necessary when the end-effector was grasping an object. A similar spasm occurs with either processor when the controller cage is initially powered up with the motor switch on. This spasm is harmless, since the end-effector is not usually grasping an object at initial power-up.

The internal serial port is used to transfer serial character input from the RS232 port to the processor and to transfer output from the processor to the RS232 port. The 8031 processor actually uses two buffers for serial communication, one for transmitting characters and one for receiving characters, although both are accessed through the same on-chip RAM address. A serial interrupt occurs when either the transmitting buffer empties or the receiving buffer fills. The processor can determine what sort of action to take when a serial interrupt occurs by examining the appropriate bit in the serial port control register.

Overview of the Gripper Controller Software

Both the previous and the new control programs for the parallel jaw gripper have the same basic structure. This section discusses their common structure from a general, high-level point of view.

A program controlling the parallel jaw end-effector must perform several actions:

- Keep a record of the current jaw opening.
- Move the jaws, when needed.

- Record, on a regular basis, the status of the proximity, crossfire, overload, and limit sensors.
- Respond to the user's requests for jaw movement or jaw status information.

The opening of the parallel jaws is correlated to encoder counts of the shaft encoder driven by the worm screw motor through the external interrupts 0 and 1 as described in the previous section. The handlers for these two interrupts maintain a record of the current encoder counts.

Jaw movement can occur in either of two ways, a move to a commanded position or a manual "jog request" from the jog switch to open or close the jaws a small amount.

In a move to a commanded position, the controller program continually checks the actual encoder value against the target value and applies the corrective drive current to bring the actual encoder value to the target value. In a jog request, the controller program simply applies briefly a current to drive the jaws in the direction requested.

The status of the proximity, crossfire, and limit detectors is continually monitored by the controller program and stored in on-chip RAM.

Interaction with the user occurs through the serial port. The lower level of this interaction was described in the part of the previous section dealing with the serial port and the associated serial interrupts. The controller program must perform the conversion from the character-level interaction with the user down to the binary level at which the controller program must operate. The controller program must also determine from the character input supplied from the user what the user wants and respond accordingly.

From the above discussion, it appears that the controller program must do several things simultaneously, which is impossible, of course, since like most people, a single processor can only do one thing at a time. This apparent concurrency is accomplished by having the processor switch rapidly from one task to another so that all of the tasks appear to progress steadily to their completion.

The Ready Queue

The interleaving of task execution is the responsibility of the operating system kernel. It is implemented by use of one of the 8031 clock/timers as an autoload timer to generate a periodic clock interrupt. The handler for this interrupt maintains a *ready queue*, a list of subroutines waiting to execute. Each subroutine in the list has an associated *schedule variable* which holds the number of clock interrupts to occur before the subroutine executes. When the clock interrupt occurs and control passes to the interrupt handler, the interrupt handler checks through the ready queue and decrements the schedule variable of each subroutine on the queue. Any subroutine whose schedule variable is zero is removed from the ready queue and made ready to execute. On termination of the handler, the subroutine removed from the ready queue executes to completion, with occasional pauses due to interrupts.

To say that this process is repeated each time a clock interrupt occurs is a slight oversimplification. The clock interrupt handler only checks the ready queue on a fraction of the times that the clock interrupt occurs. This slows the rate of checking the ready queue to a point that fits the time dynamics of the gripper hardware. Currently, the

clock interrupt handler checks the ready queue every fourth clock interrupt — three clock interrupts out of four, the handler simply returns with no action. The frequency with which the clock interrupt handler checks the ready queue can be varied interactively by the user. The value for the clock interrupt handler frequency that the program uses is stored in a location of the on-chip RAM that can be modified by the user with the interactive monitor.

The frequency with which the clock interrupt handler checks the ready queue does not affect the system. Its behavior is essentially the same whether the clock interrupt value is at the minimum value of 01 (check the ready queue at every clock interrupt) or its maximum value of FF (check the ready queue only once out of every 255 clock interrupts). The explanation for this fact is that the clock interrupt frequency determines the *frequency* of events in the controller system in relation to *real time* and not the *order* of the events or their *relative frequency*. Even at the largest value for the period of the clock interrupt timer, the system events occur rapidly enough in relation to real time not to cause a problem. It is possible that if a 16-bit value were used instead of an 8-bit value to keep track of the period of the clock interrupt handler, the frequency of system events could be slowed to the point of adversely affecting system behavior.

Process Scheduling

The *schedule* subroutine of the operating system kernel places a subroutine on the ready queue and sets the schedule variable of the subroutine to the desired initial value. The subroutines in the parallel jaw controller that must perform a task repeatedly are written as *self-scheduling* routines. One of the last statements of such a subroutine would be to schedule itself into the ready queue to re-execute at some future time. This self-scheduling causes the subroutine to be invoked periodically. The self-scheduling occurs toward the end of the subroutine to avoid having two copies of the subroutine executing at the same time. At system initialization, the main program schedules all self-scheduling subroutines to get them started.

The new program uses three self-scheduling subroutines. The *sensor* subroutine continually reads the proximity, crossfire, and limit sensors and displays the information by turning on the appropriate indicators on the front panel of the controller cage. The *jaw movement* routine continually checks to see whether the jaws must be moved, and if so, moves the jaws. The subroutine first checks to see if the jog switch is active and drives the jaws accordingly. If the jog switch is not active, the subroutine drives the jaws to zero the difference between the commanded position and the current position. The *jaw watching* subroutine continually compares the present jaw position with the previous jaw position, sets a *stopped* bit flag if the two positions are the same, and then calculates the error between the target position and the present position.

The original program only uses the *sensor* and *jaw movement* self-scheduling routines. The *jaw watching* subroutine is the program unit in the new program that provides the capability of detecting when the jaws are stopped. The corresponding *sensor* and *jaw movement* subroutines in the two programs differ significantly.

In the new program, the schedule frequencies of the self-scheduling subroutines can be varied interactively. The schedule values for the self-scheduling subroutines are taken

from locations in the on-chip RAM; they not hard-coded into the object file. As a result, the user can vary these values using the interactive monitor.

This feature was added to give the user the flexibility to determine the optimal sizes of the scheduling intervals. This feature was necessary because the controller program did not perform correctly with the values initially chosen. Furthermore, without the ability to vary the values interactively, it was very difficult to determine what the values should be. This feature has been invaluable in getting the controller program to work as it was designed.

Tests have shown that the value of the *jaw watching* schedule variable is critical for proper behavior of the system. This routine should execute as frequently as possible. When the *jaw watching* schedule variable is large, the *stopped* bit does not accurately describe the state of the jaws because the bit can be ON even though the jaws are actually moving, and the error value between the targeted and actual positions is not valid.

The inaccuracy of the *stopped* bit is most crucial during initialization of the jaws. The jaws have previously been at rest, so the *stopped* bit is ON. As the jaws are driven open to their widest position, the *stopped* bit remains ON even though the jaws are moving, because the *jaw watching* routine, which would change the *stopped* bit, is waiting on the ready queue for its schedule variable to reach zero. The checks built into the initialization routine sense from the *stopped* bit that the jaws are stopped, even though they are actually moving, and abort the initialization.

The scheduling frequency of the *jaw watching* subroutine is also important because this subroutine computes the error between the target and actual positions. If the value of the schedule variable of the *jaw watching* subroutine is significantly larger than the value of the schedule variable of the *jaw movement* subroutine, then the value of the error between the target and actual positions used by the *jaw movement* subroutine to drive the jaws is not accurate, which leads to oscillation of the jaws around the target jaw position when a jaw movement command is given. This also dictates that the *jaw watching* subroutine should execute as frequently as possible.

Detailed Description of Program Structure

The new program is contained in ten files, all in the directory

ISRL6::SYS$USERDISK[WILLMAR.NEWGRIP]

on the ISRL MicroVax.

The details of the program structure will be organized around a discussion of the contents of these files. All addresses referred to in the code are given in hexadecimal.

File	Contents
HEXHAND.51	master file that "includes" the other files in order
HHDATASTR.51	all data structures used anywhere in the program
HHMACROS.51	all macros used anywhere in the program
HHVECTORS.51	places all interrupt jump vectors
HHTTYIO.51	terminal input/output subroutines
HHSYSTEM.51	operating system: ready queue, schedule, wait
HHINIT.51	initialization routines
HHSENSORS.51	read and display sensor values
HHMVJAWS.51	encoder interrupt handlers, jaw movement, jaw watching
HHCMDLOOP.51	main program, command loop, interactive monitor

HEXHAND.51

This file is a shell file that "includes" the necessary files into the assembly in the proper order. It contains no code.

HHDATASTR.51

This file contains all data structures used anywhere in the program. This includes the 128 bytes of on-chip RAM, as well as the external port addresses. The data structures are defined in *memory order* – which means that the file itself provides a memory map of the on-chip RAM. The 8031 processor has four banks of eight registers that occupy the first 32 bytes of on-chip RAM. Only the first bank is used in the program, which occupies bytes 0 through 7. The program data structures occupy the rest of the on-chip RAM from location 08 through 7F. The program data structures are, in order, the buffers for terminal I/O, the bit flags, the sensor bits, the ready queue, and the remaining program variables, such as the current encoder count and the commanded encoder count, and finally, the system stack. A memory map of the on-chip RAM is included in an appendix.

The system stack occupies the top part of the on-chip RAM, from location 54 through location 7F. This represents a significant change from the original design and was necessitated because the system stack location in the orignal design occasionally resulted in overwriting some of the flag bits when the stack grew larger than its expected size.

The file generates no object code.

HHMACROS.51

This file contains all assembly language macros used anywhere in the program. There are macros for augmenting the instruction set of the 8031 processor, such as word-sized (2 byte) arithmetic and compare-and-jump-on-equal, and macros for simplifying procedure call prologues and epilogues.

The file generates no object code.

HHVECTORS.51

This file contains the code necessary to place the jump vectors for all of the interrupts. The jump vector locations for the 8031 processor are as follows:

code memory address	interrupt type	used for
0000	power-on reset	system initialization and re-initialization
0003	external request 0	positive-going motor shaft encoder
000B	clock/timer 0	system clock autoload timer
0013	external request 1	negative-going motor shaft encoder
0023	serial port	terminal input/output

When the particular type of interrupt occurs, the 8031 processor executes the code at the memory address shown. This code is typically a jump to the appropriate interrupt handler.

HHTTYIO.51

This file contains the subroutines needed for terminal input and output, namely:

- the handler for the serial port interrupt. This handler actually does the low-level work of writing the output buffer to the terminal screen and storing input from the terminal keyboard into the input buffer.
- the routine that moves characters from the program to the output buffer.
- the routine that moves characters from the input buffer into the program and echoes them to the screen. All character comparisons in the program are made between upper case ASCII characters. This routine also coverts lower case ASCII characters into upper case before storing them in the input buffer and echoing them to the terminal. This feature relieves the user of having to reset his or her terminal before using the program, since it accepts lower case and upper case input equivalently.
- the routines that do character conversion and number conversion from binary used in the program into hexadecimal for display and from hexadecimal entered by the user into binary for use by the program. The original program used octal notation for input and output instead of hexadecimal. With the inclusion of an interactive monitor, the change to hexadecimal offers a considerable convenience to the user, since it relieves the user from having to convert back and forth from the octal terminal I/O to the hexadecimal used in the assembler listing.
- the high-level routine that writes strings to the terminal screen. This routine is used to write all messages to the user.

HHSYSTEM.51

This file contains the basic operating system subroutines:

- the interrupt handler for the clock timer interrupt. This routine maintains the ready queue and the associated schedule variables.
- the routine to initialize the operating system and 8031 processor variables.
- the *schedule* routine that adds routines to the ready queue. The behavior of this routine was described in the previous section.
- the *wait* routine. This routine is used in the initialization routine where it is necessary to pause briefly before proceeding — to wait for the worm drive motor to stop turning or to wait for the 8155 chip to reset. The wait is accomplished through clearing a flag bit, scheduling a subroutine that will set the bit when it executes in the future, and then entering a "busy wait" loop that terminates when the bit is set.

HHINIT.51

This file contains the two main subroutines that are used to bring the end-effector and its controller into a consistent initial state.

The *init* routine initializes the 8155 I/O chip, the sensor data bits, and the variables in the on-chip RAM that relate to encoder counts.

The *fullopen* routine drives the jaws to the position of maximum jaw opening. This is the position that corresponds to zero encoder count value that is set by the *init* routine. The full open position is detected by the limit sensor. The *sensor* self-scheduling routine sets the *limit* bit in the program when the limit sensor fires.

In the old program, if the limit sensor was inoperative, the *limit* bit would never be set and the program would continue to drive the jaws open. After the jaws reached the physical limit of jaw opening, the jaws would stop moving, but current would continue to be applied to the worm screw drive motor. This does not damage the drive motor but it *does* generate heat that is harmful to the heat-sensitive components in the end-effector housing, such as the force-torque sensors.

The new program checks to be sure that the jaws are moving during the *fullopen* initialization. If the jaws stop moving before the *limit* bit goes on, the initialization is aborted and control is passed to the interactive monitor. The new program successfully detects limit sensor failure.

HHSENSORS.51

This file contains the self-scheduling *sensors* routine that reads the current values of the proximity, crossfire, overload, and limit sensors, stores them in the proper place in the on-chip RAM, and displays the results in the LEDs on the controller cage front panel.

HHMVJAWS.51

This file contains three types of subroutines: the handlers for the encoder interrupts and the *jaw movement* and *jaw watching* self-scheduling subroutines.

Encoder counts are maintained by the two external request interrupt handlers. When the external request 0 interrupt occurs, the encoder count value stored in on-chip RAM is incremented, and when the external request 1 interrupt occurs, the encoder count value is decremented.

The Jaw Movement Subroutine

The *jaw movement* subroutine is one of the three self-scheduling subroutines used in the new program. This subroutine is responsible for two types of jaw movement: a move to a commanded position, or a "jog request" made by the manually-operated jog switch on the front panel of the controller card cage to open or close the jaws a small amount.

The jaws are also moved by other parts of the program during initialization and calibration. It is desirable to be able to temporarily disable the effects of this subroutine during those periods, so that the worm gear drive motor will not receive contradictory drive information. The *jaw movement* subroutine is enabled and disabled by means of a *drive* bit in the on-chip RAM. If the bit is set, then the subroutine executes normally, but if the bit is clear, then normal execution is bypassed. The *drive* bit is on only during a move to a commanded position.

Each execution of this subroutine consists of the following steps. First, the subroutine checks to see if either an open or close jog has been requested. If so, then the subroutine drives the motor in the appropriate direction and returns. If there is no active jog request, the subroutine checks the *drive* bit to see if it is on. If so, then the subroutine drives the jaws with a current that is equal to the difference between the target position and the current position. Then, the subroutine checks to see if the *stopped* bit has been set by the *jaw watching* subroutine.

If the *stopped* bit has been set, this means that the jaws may be close to the targeted position and the move to a commanded position is nearly over. It might seem that under these conditions the jaws should actually be at the commanded position. What happens in practice is that the jaws stop moving slightly before the commanded position is reached, and drive current should continue to be applied to the motor for a short period longer. This is accomplished by the use of an auxiliary counter variable that is loaded with a positive value prior to the beginning of a move to a commanded position. After the *stopped* bit goes on, this counter variable is decremented in each execution of the subroutine until the variable reaches zero, at which time the current to the motor is zeroed and the *drive* bit is cleared. This counter variable thus briefly delays terminating the drive current to the motor. Moreover, it prevents a high drive current from being applied to the drive motor indefinitely.

If the *drive* bit is off when the subroutine is entered, the subroutine checks to see if the previous execution of the subroutine involved a jog request. This is done by checking a different bit flag, the *dojog* bit, in the on-chip RAM that is set by this subroutine during a jog request. If the *dojog* bit is on when this check is made, then the value at the motor port is zeroed to stop the jog and the *dojog* bit is cleared. This step is necessary to overcome the previous value written to the port and stop the jog movement. If the previous execution

of the subroutine did not involve a jog request and the *drive* bit is off, then the subroutine leaves without any further action.

The Jaw Watching Subroutine

This self-scheduled subroutine continually sets or clears the *stopped* bit according to whether or not the jaws are stopped. In addition, it updates the "current" and "previous" encoder values and calculates the difference between the targeted and current encoder values.

Since the *jaw movement* subroutine watches the *stopped* bit carefully, the *jaw watching* subroutine should be rescheduled more frequently than the *jaw movement* subroutine.

HHCMDLOOP.51

This file contains the top-level of the main program. Control passes to this code at power-up and at power-on reset.

First, the schedule variables for the self-scheduling subroutines and the frequency with which the clock interrupt handler checks the ready queue are initialized. Then the jaw movement variables and the jaw opening of the gripper are initialized by calls to the HHINIT.51 routines. Finally, the program enters a command loop and begins accepting commands from the user. The prompt used for the main command loop is "CMD: ".

The possible commands that can be entered are:

Command	Description
C	display the range of encoder counts from the wide open to fully closed jaw positions
E	display error between target and actual positions
I	initialize the gripper and its data structures
M	enter the interactive monitor
P	move to a commanded position
S	display current sensor values
T	display target position
W	display current position
X	change speed of jaw movement

The S command displays the current states of the sensors in binary format; all other displays are given in hexadecimal.

C Display the range of encoder counts from the wide open to the fully closed positions. This command initializes the gripper as in the **I** command below, closes the jaws until they stop, displays the corresponding encoder count value, and re-initializes the gripper. The value displayed is slightly more than the actual total range of encoder counts because of looseness in the jaw linkage and compliance of the material used in the jaw surfaces. If either initialization of the gripper fails, control passes to the interactive monitor (see the **M** command below).

E Display error between target and actual positions. As mentioned in the hardware section, the error value is kept in excess-128. Since the error value is shown in hexadecimal, a value of "0080" corresponds to zero error, a value of "008F" corresponds to an error of 15 (the actual position is 15 encoder counts more than the target value),

and a value of "007A" corresponds to an error of –6. The error value is always equal to the actual position (the value displayed by the **W** command) minus the target position (the value displayed by the **T** command).

I Initialize the gripper and its data structures. This command initializes the gripper by initializing the on-chip RAM location dealing with encoder counts and then driving the jaws to the wide-open position. The wide-open position of the jaws is sensed by the limit sensor. The current value of this sensor is stored in the *limit* bit. If the initialization procedure notices that the jaws have stopped (the *stopped* bit is on) before the *limit* bit comes on, the initialization is aborted with an error message, and control is passed to the interactive monitor so that the user can determine the cause of the initialization failure (see the **M** command below).

M This command enters the interactive monitor. The command line prompt changes from "CMD: " to "* ". The following commands are accepted:

Monitor Command	Description
C	change contents of a location in on-chip RAM
D	display contents of a location in on-chip RAM
I	toggle clock/timer interrupts
L	test sensor LEDs
N	display contents of next location in on-chip RAM
Q	quit the monitor and return to main command level
S	display contents of SP, the stack pointer

C addr value

Change the contents of on-chip RAM location *addr* ($0 \leq addr \leq 7F$) to *value*. The address and new value stored at the address are displayed after the storage takes place.

D addr

Display the contents of on-chip RAM location *addr* ($0 \leq addr \leq 7F$).

I

Toggle the clock/timer interrupt on or off. This command is needed to be able to look at the ready queue or test the sensor LEDs. Disabling the clock/timer interrupt suspends all changes to the ready queue so that the user can use the *D* monitor command to inspect the ready queue and the associated schedule variables. If the clock/timer interrupt is not disabled, then use of the *D* command to inspect the ready queue leads to confusing results, since the ready queue is being changed by the clock/timer interrupt handler as it is being inspected.

L bit-pattern

Test sensor LEDs. This command allows the user to test each sensor LED individually. The full word (two-byte) *bit-pattern* is stored at the two on-chip RAM locations used to store the sensor information. The proximity sensor bits are stored at location 2D and the crossfire, overload, and limit bits are stored at location 2E. This command stores the full word bit pattern entered by the user at locations 2D and 2E and then displays the contents of location 2D. The action

of displaying the contents of the location just stored was added to make sure that the intended value for the sensor LEDs was actually being stored at locations 2D and 2E.

The following example of use of this command may be helpful. In the interaction shown below, the typing done by the user is shown in italics.

* *I*
* *L FBFE* 2D FE
* *N* 2E FB

The user has stored the bit pattern "FBFE" (in hexadecimal) or "1111 1011 1111 1110" (in binary) in the sensor bytes of the on-chip RAM. The *I* command was necessary to suspend the clock/timer interrupts, so that the self-scheduling *sensor* subroutine will not replace the values entered with the current sensor values. The value "FE" is stored at location 2D and the value "FB" is stored at the location 2E (the low order byte is stored at the lower address and the high order byte is stored at the higher address). The "FE" of the pattern turns on the crossfire LED, and the "FB" turns on the LED corresponding to the +X detector on the right jaw (see the main command level S command for a description of the locations of the sensor bits). The LED is ON if and only if the corresponding bit is zero. You will notice that, as a side-effect of the command, the contents of location 2D are displayed. If, as the user has done here, you want to see the contents of the next byte (the next sensor byte containing the limit, overload, and crossfire bits), you can use the monitor *N* command to display the contents of the byte at location 2E.

The following bit patterns can be used to test each LED on the front panel of the controller cage:

Bit Pattern	LED Location
FF7F	–X, left jaw
FFBF	+X, left jaw
FFDF	+Z, left jaw
FFEF	–Ya, left jaw
FFF7	–X, right jaw
FFFB	+X, right jaw
FFFD	+Z, right jaw
FFFE	+Ya, right jaw
FBFF	limit
FDFF	overload
FEFF	crossfire
F800	all LEDs ON
0000	all LEDs ON
FFFF	all LEDs OFF
07FF	all LEDs OFF

N

Display the next memory address and its contents. The address displayed is the address next to the most recently referenced address by the *C* or *D* commands.

When the monitor is entered, the default memory address is set to the current value of the stack pointer, SP.

Q

Quit the monitor and return to the main command loop.

S

Display the current contents of the stack pointer, SP. Even though the stack pointer is kept at location 81 in the on-chip RAM, the *D* command cannot be used to display its value. The *D* command is implemented using the MOV instruction with register-indirect addressing, and the 8031 chip restricts the range of this instruction to the values between 00 and 7F. The stack pointer cannot be accessed by the *D* instruction since its address is outside of this range.

Any other characters are accepted but ignored.

P Move to a commanded position. This command accepts a target encoder value from the user, stores it in the on-chip RAM, and sets the *drive* bit so that the *jaw movement* routine will start driving the worm screw motor to zero the difference between the current encoder value and the target value.

Encoder counts range from "0000" when the jaws are wide open to the value of approximately "D600" when the jaws are closed. Encoder counts are always non-positive and are represented in two's complement notation. Consequently, a value of "D600" really stands for the two's complement negative of "3A00". Therefore, the commands "PE47A" and "P-1B86" have the same effect, since "E47A" is the two's complement representation of the negative of "1B86". Either form of the command will be accepted by the command interpreter.

A comment is needed about use of the **P** command. The **P** command has been improved in the new version of the program. With the two's complement convention used for the number of encoder counts, any **positive** number of encoder counts can never be achieved. In the old program, supplying a positive target would cause the jaws to be driven closed and the position encoder counts would become increasingly negative. This resulted in an unrecoverable error situation since the encoder counts could never become positive. The new program rejects a positive encoder target as an invalid command.

S Display current sensor values. This command displays the current sensor values as a 16-bit binary string. These values are stored in locations 2E and 2F of the on-chip RAM. The leftmost eight bits of the binary string are the bits of location 2F and the rightmost eight bits are the bits of location 2E. The bits have the following meanings (bit 15 is the leftmost bit and bit 0 is the rightmost bit):

Bit Pattern	LED Location
15..11	unused - bit might be either 0 or 1
10	crossfire
9	overload
8	limit
7	right jaw, -X
6	right jaw, +X
5	right jaw, +Z
4	right jaw, -Ya
3	left jaw, -X
2	left jaw, +X
1	left jaw, +Z
0	left jaw, +Ya

The bit in the sensor bit string is "0" if and only if the corresponding sensor is *on*. For example, if, in response to the S command, the user receives the bit string:

1111101101101101

this would indicate that the following detectors are *on* (proceeding from left to right):

crossfire (leftmost 0)
right jaw, -X
right jaw, -Ya
left jaw, +Z (rightmost 0)

T Display target position. This command shows the last commanded position in encoder counts. When the jog switch is used to open or close the jaws, the target position is kept equal to the actual position, so that the jaws will not be moved back to the previously targeted position by the *jaw movement* subroutine.

W Display current position. This command shows the encoder counts corresponding to the current jaw opening.

X With this command, the user can change the speed at which the jaws open and close. When a combination of worm and sector gears is used in an end-effector with a significantly higher gear ratio than the ratio used in the original design, the maximum speed of the worm drive motor has to be reduced because the jaws move too rapidly. With the original worm/sector combination, a speed value of "7F" is used, but with the newer high-ratio worm/sector combination, a speed of "33" seems to move the jaws at approximately the same speed as the original gearing.

Conclusions

The program and documentation produced by this research comprise the latest step in the evolution of software to control the ISRL parallel jaw end-effector. As such, it represents the cumulative experience of a large group of people over a long period of time.

The capabilities of the controller program have been expanded considerably. The software has been thoroughly documented. The user interface has been extended to include an interactive monitor. The flexibility of the program has been improved by moving several

of the program parameters from the EPROM object code to storage locations in the on-chip RAM that can be modified with the interactive monitor.

Of all of the modifications to the existing program, the addition of the interactive monitor probably has the greatest significance. Assembly language programming is difficult, even in the best of circumstances. The information provided by the interactive monitor is invaluable in determining root causes for irregular or hard-to-explain program behavior. The program is interrupt-driven and the interrupts occur with a frequency related to processor clock frequency. Consequently, simulation of the interrupt behavior of the system through the 8051 software simulator is not possible. There are many interrupts that can be interleaved in arbitrary order — this represents a combinatorially explosive number of possible test cases. Therefore, the interactive monitor is the principal tool available to the programmer in testing the program and correcting errors. It was invaluable in this work.

References

1. M. J. Wise, Description of the End-Effector/Sensor System, Automation and Technology Branch Internal Report, NASA/Langley Research Center, Hampton, Virginia.

2. MCS-51 Familty of Single Chip Microcomputers – User's Manual, July, 1981, Intel Corporation, Santa Clara, California.

3. Microcontroller Handbook, 1986, pp. 7-1 through 8-85, Intel Corporation, Santa Clara, California.

Appendix A
On-Chip RAM Memory Map

On-chip RAM Address (hex)	Program Name	Description
00	REG0	R0, register 0
01	REG1	R1, register 1
02	REG2	R2, register 2
03	REG3	R3, register 3
04	REG4	R4, register 4
05	REG5	R5, register 5
06	REG6	R6, register 6
07	REG7	R7, register 7
08	OINPTR	address of characters entering the output buffer
09	OOUTPTR	address of characters leaving the output buffer
0A	OBYTCNT	count of bytes in output buffer
0B	–	output buffer
15	IINPTR	address of characters entering the input buffer
16	IOUTPTR	address of characters leaving the input buffer
17	IBYTCNT	count of bytes in input buffer
18	–	input buffer
2C	–	bit flags
2C.0	TBFULL	bit 0, 1 if transmit buffer full
2C.1	SEMA	bit 1, semaphore used by WAIT routine
2C.2	STOPPED	bit 2, 1 if jaws are stopped, else 0
2C.3	DRVON	bit 3, 1 if jaws are driven in servo mode
2C.4	DOJOG	bit 4, 1 if last jaw movement was a jog
bits 2C.5, 2C.6, 2C.7 are unused		
2D	–	proximity sensor bits
2D.0	–	bit 0, left jaw, +Ya sensor
2D.1	–	bit 1, left jaw, +Z sensor
2D.2	–	bit 2, left jaw, +X sensor
2D.3	–	bit 3, left jaw, –X sensor
2D.4	–	bit 4, right jaw, –Ya sensor
2D.5	–	bit 5, right jaw, +Z sensor
2D.6	–	bit 6, right jaw, +X sensor
2D.7	–	bit 7, right jaw, –X sensor
2E	–	other sensor bits
2E.0	–	bit 0, crossfire sensor
2E.1	–	bit 1, overload sensor
2E.2	–	bit 2, limit sensor
bits 2E.3 ... 2E.7 are unused		
2F	T	queue of schedule values, ready queue routines
35	Q	queue of addresses, ready queue routines
41	SYSCLKCNT	system software clock, check ready queue when it's 0
42	SYSCLKPD	reload value of SYSCLKCNT

Appendix A
On-Chip RAM Memory Map

43	MVJAWSPD	MVJAWS routine schedule value
44	SENSORPD	SENSORS routine schedule value
45	WTCHJWSPD	WATCHJAWS routine schedule value
46	ERROR_CNT	error between target and actual encoder counts
48	RATE	difference between current and previous encoder counts
4A	ENCODER_CNT	current encoder count
4C	COMMAND_CNT	target encoder count
4E	OLD_ENC	previous encoder count
50	STOPCNT	# times to bump jaws at end of commanded move
51	JSPEED	maximum absolute value of jaw speed
52	MXPJSPD	maximum positive jaw speed (in excess-128)
53	MXNJSPD	maximum negative jaw speed (in excess-128)
54 ... 7F	–	system stack

Appendix B
EPROM Addresses of Self-Scheduling Subroutines

EPROM Address (hex)	Subroutine Name	Schedule Value	RAM Addr. of Sched. Val.	Function
0295	SYSCLK	04*	42	clock/timer interrupt handler
04C2	MVJAWS	0C	43	move jaws, either jog or in differential servo drive
0473	SENSORS	0A	44	read and display sensor values
0541	WATCHJAW	01	45	check to see if jaws are stopped & calculate error value

* This value is the number of clock interrupts that occurs before the subroutine checks the ready queue.

Appendix C
Summary of Controller Commands

Top-level Controller Commands

Form of Command	Action
C	open jaws fully, close fully, report total encoder range
E	display current error value in excess-128
I	open jaws fully, initialize controller data structures
M	enter the interactive monitor (commands are listed below)
P *target*	move to *target* position (encoder counts)
S	display current sensor values as a binary bit string
T	display target position (encoder counts)
W	display current position (encoder counts)
X *speed*	change speed of jaw movement to *speed*

Interactive Monitor Commands

Form of Command	Action
C addr value	change the contents of on-chipRAM location *addr* to *value*
D addr	display the contents of on-chip RAM location *addr*
I	toggle clock/timer interrupt ON or OFF
L bit-pattern	test sensor LEDs (0000 → all LEDs ON, FFFF → all OFF)
N	display address and contents of next on-chip RAM location
Q	quit the monitor and return to the top level
S	display the contents of the stack pointer