A Real-Time Microprocessor-Based Laboratory System

© Irvin Shizgal

McGill University

School of Computer science

In partial fulfillment of the requirements for the Master of Science degree.

August 1992 ©

Abstract

A simple semaphore based operating system for the Motorola M6800 microprocessor is described. It is a stand alone system using fixed priority scheduling, designed as a controller in a series of neurophysiological experiments in a psychology laboratory. The overall scheme calls for one microprocessor controlled station for each experimental animal (rat) with an LSI-11 (equipped with floppy disks) choosing experiment parameters and recording the resulting data. Inter-processor communication is performed via serial links between each M6800 based system and the LSI-11 master.

Resume

On decrit un systeme simple a base de semaphore pour le microprocesseur Motorola M6800. Ce systeme est concu pour le controle d'une serie d'experiences neurophysiologiques dans un laboratoire de psychologie; il est capable d'operation individuelle, utilisant un programme de priorites fixes. L'arrangement d'ensemble prevoit un poste commande par microprocesseur pour chaque animal experimental (rat); un LSI-11 (muni de disquettes souples) effectue le choix des parametres de l'experience et enregistre les donnees qui en resultent. La communication inter-processeur est assuree par un jumelage en serie entre chaque systeme a base de M6800 et la machine maitresse LSI-11.

Acknowledgements

I would like to acknowledge the assistance of Prof. Peter Shizgal of Concordia University, without whom the project upon which this thesis is based would not have existed.

I would also like to thank Prof. G. Cormack for his comments on chapter five, and, of course, my supervisor Prof. G. Ratzer.

Contents

Chapter 1

INTRODUCTION

The purpose of this project is to provide a computer system (hardware and software) to automate a series of neurophysiological experiments using rats as experimental subjects. A large amount of data are needed to allow inferences to be drawn about some of the structures (and properties of these structures) found in the rats' brains. The decision to undertake this development effort was that of associate professor Peter Shizgal of Concordia University's Psychology Department to facilitate his research and that of his students.

1.1 Reasons for developing the system

1.1.1 Experimental paradigm

The type of experiments being undertaken involve the direct electrical stimulation of discrete regions within a rat's brain, while observing the behaviour of the rat ([GSY81]). The stimulation is of a sort that the rat finds rewarding, and which it can trigger itself by pressing on a bar in its cage. Sequences of pulses are applied to various sets of implanted electrodes, and stimulation parameters are adjusted while keeping track of the rat's bar-pressing activity

in order to test various hypotheses concerning neural structures.

1.1.2 Justification for computerization

The way these types of experiments were carried out in the past is as follows:

A student would sit before several (typically two) cages containing rats, and an instrument panel containing switches for electrode selection, pulse timing, pulse current ...etc., and a counter for accumulating bar presses over each trial. The student would 1) set up the next stimulation parameters, 2) trigger a priming stimulation to condition the rat, 3) wait for the duration of an experimental trial (usually one minute), and 4) at the end of the trial period record the number of bar presses and stimulations delivered. This sequence would be repeated hour after hour with stimulation parameters changing gradually in order to determine the threshold of effective stimulation and to trade parameters off against each other.

This manual sequence is extremely slow, tedious, and error prone. Also, since data are gathered so slowly, it is sometimes necessary to use the same rat in tests spanning many days, thus increasing the probability of an electrode becoming dislodged, an infection, or some

other mishap rendering the rat, and perhaps even the data gathered to that point, useless. The computer system eliminates the tedium and inaccuracy, can test as many as eight rats at once, and apply more complex algorithms for computing successive sets of stimulation parameters in order to reduce the number of trials required to determine a threshold.

1.2 System overview

The system is conceptually partitioned into three levels. In level I is the master computer overseeing all in-progress experiments, its hardware, and software. Level II consists of eight microprocessor systems (one per rat cage), their software, and interface hardware. The third and final level consists of the rat cages themselves, and all special purpose hardware associated with them.

1.2.1 Level I (LSI-11)

1.2.1.1 Level I hardware

The level I processor is a Digital Equipment Corporation LSI-11 with dual double density eight inch floppy-disk drives, running the RT-11 operating system. A CRT terminal acts as the system console, and an LA-120 Decwriter is available for hard copy. As well as interfaces for the above devices, the LSI-11 is equipped with two DLV-11J quad serial interfaces for communications with the eight level II processors.

1.2.1.2 Level I software

The LSI-11 software is itself partitioned into two major levels. The upper level software has to actually run the experiments - i.e. download the level II processors, decide on parameters for each trial, collect and log data, and provide reports to the experimenter/operator. The lower level software supports communication with the cage controller processors. The LSI-11 software was developed (and in part is still being developed) under RT-11 in Macro-11 and FORTAN by David Morton.

1.2.2 Level II (M6800)

1.2.2.1 Level II hardware

The level II hardware consists of eight processors based on the Motorola M6800, each equipped with 8k bytes of static RAM, an M6821 (peripheral interface adapter), an M6840 (triple programmable timer), an M6850 (asynchronous communications interface adapter), and a dual 12-bit digital to analog converter board. The 6821 PIA is used to interface various devices on the cage, the 6850 ACIA provides communication with the master, the 6840 timers are used to maintain a system time and generate pulses, and the digital-analog converters control pulse current magnitude.

1.2.2.2 Level II software

The level II processors must be able to handle communications from level I (receive commands and parameters, send data generated by trials), respond to rat activity in the form of lever presses, deliver very accurate stimulation pulse sequences, and keep track of passing time (a trial can be minutes long). For this purpose a general semaphore-based multi-tasking operating system was written for the M6800. This system was designed, implemented, and debugged by the author, and is the primary focus of this paper.

1.3 Major design choices

1.3.1 Level I

The choice of RT-11 for the master LSI-11 was mainly dictated by the fact that it was easily available and documented in sufficient detail. There were also other RT-11 users within the Concordia community whose experience would be available if problems should be encountered with the system.

The decision to encapsulate the communications portion of the LSI-11 software arose naturally from the fact that its function would remain fundamentally the same no matter what type of higher level experimental driver were added. In all experiments the level II processors must be loaded, and messages from/to them must be queued until processed/sent.

1.3.2 Level II

1.3.2.1 Generality

The justification for investing the extra resources involved in writing a general purpose operating system, rather than one designed to directly implement the experimental paradigm to be handled (although it is quite debatable whether this would, in fact, be simpler) lies in the fact that the system is research oriented. This implies that future experimental procedures will likely depend, in part, on the results of previous experiments carried out with the system. This being the case, it is likely that a specially designed set of programs would have to be re-written from scratch as soon as the experiments to be performed changed in any non-trivial detail. Such a re-write requires more time and technical expertise to perform than re-designing the experiment tasks given the facilities provided by the operating system.

1.3.2.2 The hardware/software functional split

One of the more difficult choices which arose early in the design phase was the question of how much work would each of hardware, software, and OS software contribute to the actual pulse generation and timing. To understand this clearly it is necessary to explain in more detail the format of the stimulation pulses.

At the lowest level, pulses are delivered in pairs called the "C" (for "conditioning") pulse and the "T" (for "test") pulse (if a single pulse only is desired, it is arbitrarily designated a C pulse). The pulse widths and the C-T interval could conceivably be as small as some tens of microseconds, and therefore must be very accurately timed - so much so that an instruction execution time is significant (ranging from 2 usecs. to 12 usecs. - with instructions of over 7 usecs. being comparatively rarely used).

These C-T pairs are combined into trains, and a sequence of trains comprises a stimulation. The resolution of the C-C interval must be less than 100 usecs., while the resolution of the inter-train delays can be as coarse as tens of milliseconds. A typical stimulation is long enough to watch directly on the face of an ordinary oscilloscope (typically, between 0.1 and 2.5 seconds).

It was decided that for reasons of reliability, accuracy, and elegance the C-T pairs were best generated by hardware. The rejected option in this case was the use of specially timed instruction loops - with different pieces of code required for different timing ranges. For this purpose custom hardware was developed (the prototype by the author, later improved by Peter Shizgal).

For the next interval, the C-C interval, it was clear that the operating system overhead would be too great to achieve the necessary resolution using a system call. A timed delay loop with interrupts disabled was settled on as acceptable.

For both of the largest intervals - the inter-train delay and the experiment trial duration - the time resolution provided by the operating system was acceptable.

1.3.3 Level III

Level III was specified by Peter Shizgal, and was dictated by previous experience with manual experimental setups, and instrumentation requirements. This equipment included the rat cages, the switches, lights, and other devices built into them, voltage controlled constant current amplifiers for generating pulses, electrode switchers, etc., The cages, distribution panels, etc. were constructed by students, and the electronic design and construction was done by William Mundl, the technical officer for the Psyohology Department of Concordia University. This level of the system is viewed as a black box by the higher levels.

Chapter 2

OPERATING SYSTEM SOFTWARE

2.1 System design

2.1.1 Style of system

The operating system supports multiple processes, with process synchronization being handled by semaphore primitives. Both for aesthetic reasons, and to simplify maintenance, reduce development and implementation time, and reduce user learning time, consistency and simplicity were the primary goals when designing the system.

2.1.2 Data structures

Each process within the system is associated with a process block which holds its priority, context (in the form af a stack pointer), and wake-up time (not always used). There are a fixed number of process blocks within a region of memory reserved for system use, and this number determines the maximum number of processes which can exist at any one time. Each of these blocks is identified by a (fixed) number which serves as a process identifier. A process can be in one of four states: (1) dead (inactive)- the process block is free, (2) blocked- the process cannot run until some event occurs, (3) ready- the process can run as soon as the

CPU is available, and (4) running.

A running process has its context installed in the machine registers, and its process block pointed to by a system variable (PBCRNT). All processes in the ready state are kept in the ready queue, sorted in order of priority (so that the next process to run is at the top of the list). A blocked process can be either in a semaphore queue (again sorted according to priority) awaiting an explicit event signal, or in the sleep queue awaiting its wake-up time.

Like process blocks, there are a fixed number of semaphore blocks available in an area of memory reserved for the system. Each of these blocks is referenced by a number which is always used to refer to the associated semaphore in system calls. Semaphore blocks contain a count (the number of abstract resource units currently available), and a queue pointer.

2.1.3 System services

The following paragraphs outline the services which the operating system kernel provides to user processes.

2.1.3.1 Synchronization

The synchronization primitives are referred to as P (wait) and V (signal) as they were originally designated by Dijkstra ([Dijk68]). Each call passes to the system the number of the semaphore to be used. Semaphore numbers must be assigned during application system design, since the kernel does not keep track of which semaphores are in use.

2.1.3.1.1 P (wait)

The "P" operation requests a single unit of resource. If the semaphore count is non-zero when the call is performed, then the count is decremented by one, and the caller is allowed to resume execution. A caller will never lose the CPU as a result of performing a "P" operation on a semaphore with a non-zero count.

If the semaphore count is zero when a "P" operation is performed on it, the calling process is inserted into the semaphore's queue in order of process priority. (In the case of more than one process of the same priority, ties are broken on a first-come

first-served basis.)

2.1.3.1.2 V (signal)

The "V" operation produces (releases) a single unit of resource. If the semaphore queue is empty, then its count is incremented and the caller cannot lose the CPU (at least not as a result of the call). If the queue is not empty, then the process at the top of the list will be able to run. Since this process may be of equal or greater priority than that of the caller, both processes must be entered into the ready queue so that the dispatcher can decide which should run next. The waiting process is inserted into "ready" first, since if both are of equal priority the one which was waiting should be scheduled first.

2.1.3.2 Process birth and death

Except during initialization, processes always come into existence by an already existing process generating a "start" system call. The parent process must supply a context for the new process, in the form of a stack pointer, and a priority, which will stay with the new process for life. The id-number of the new process is returned to the parent, if the 'birth' is successful. The id-number must be saved if the offspring is to eventually be killed, or some higher level message passing protocol added to the system.

A process is destroyed by some process (perhaps itself) performing a "kill" system call specifying the process id-number of the process to be destroyed. It should be noted that the family tree of processes within the system is not stored. Thus a process which creates several others can be destroyed without affecting its offspring.

To make process suicide possible without any special arrangements for communicating id-numbers, a system call ("who") is provided which returns the process id-number of the caller.

2.1.3.3 System time and the sleep queue

The system maintains a four-byte count of the elapsed time since system initialization. Since this is intended to provide comparatively large scale intervals, the time base is supplied by an external 10 msec. oscillator to all (eight) M6800 CPU's in the system. The purpose of maintaining a system time is to allow processes to place themselves in a dormant state, possibly for a long time (from minutes to days). Consequently, the arguments provided for a "sleep" system call are a two-byte count and a four-bit exponent. Thus, the farther in the future is the wake-up time, the lower will be the ultimate resolution of the actual time of reactivation. This is not seen

as a serious limitation, since the sytem time itself will drift due to the accumulation of small delays in servicing clock interrupts. This portion of the system has gone through two distinct versions involving a complete redesign for reasons outlined below.

In the initial system design, one of the programmable timers supplied on the M6840 chip was used as a free running oscillator which generated interrupts at regular intervals. At each interrupt, the system time was updated, the sleep queue checked for processes whose wake-up time had come, and these would be then moved to the ready queue. This provided a system time of the same accuracy as the crystal time base for the CPU (since a free running oscillator re-starts automatically without intervention by a service routine).

This timing system worked effectively from an operating system viewpoint. However, upon testing stimulation pulse generation, it was discovered that the frequent interrupts generated by the clock caused an unacceptable jitter in the timing of C-C intervals (these are timed by program loops). Disabling interrupts during pulse trains would have resulted in a possibly large, and generally unpredictable number of lost clock interrupts. Both types of error were unacceptable.

The following solution was devised. Instead of running the programmable timer freely, it would be run as a one-shot. Now, each time the timer is re-started, it is set for the minimum of (1) the maximum time interval possible, and (2) the time to the next process wake-up time. The system must keep track of the last interval programmed into the timer hardware, and when a clock interrupt occurs the system time must be updated by this amount, and the next interval computed. If a "sleep" call occurs during an interval, the timer must be stopped, its counter read, and the elapsed portion of the interrupted interval computed in order to update the system time.

If this time system had been run using the CPU time base (1 mhz.) then the maximum interval would have been only 65 msecs.- so it was decided that a longer external time base was needed for the programmable timers used for the system clock. A time grain of 10 msecs. was decided upon, and therefore an external 10 msec. time base was provided for the programmable timers. This gives the current version of the system the ability to run for as long as (approximately) 10.9 minutes before having to service an interrupt for the system clock.

The cost of this technique is that some small amount

of time is lost every time a clock interrupt occurs, or when the clock interval must be artificially ended because of a "sleep" system call. This causes the system time to gradually fall behind its correct value, and this error will increase linearly with the frequency of clock interrupts and "sleep" calls. Fortunately, this error is not great enough to cause any serious concern, especially over the magnitudes of intervals required for these types of experiments (generally on the order of minutes).

2.1.4 System characteristics

2.1.4.1 Scheduling

As mentioned above, all processes are assigned a fixed priority at birth. Whenever a process is inserted in the ready queue, or a queue associated with a semaphore, it is inserted after the last process on the queue of the same or greater priority. Thus, processes of greater priority are always served before those of lower priority, and processes of equal priority are dealt with on a first-come first-served basis. These rules of scheduling are applied identically to allocation of the CPU and semaphore resources.

With queues for semaphores and the CPU kept sorted in this manner, allocation of resources becomes simply

a case of removing the first process in a queue and allocating it the resource (either by inserting it in the ready queue if the resource is a semaphore, or making it the current process if the resource is the CPU). So that there are no exceptions to this allocation scheme an idle process is (must always be) provided to consume unused CPU time. When the system reaches this state (i.e. the idle process is running) it can only be changed by an interrupt allowing some waiting process to start again.

The priority is contained in a one-byte number. A process of priority of 0 is dead (i.e. the process block is free). Priorities of 1..254 are 'ordinary' priorities, where a greater number indicates greater priority. It was decided that it would be useful to have the highest priority designate 'infinite' priority, and so a priority of 255 is considered 'non-interruptible' and the dispatcher will automatically mask further interrupts when assigning the CPU to a process of this priority. The prime reason for this decision is that it makes it possible to have the interrupt poller, the program which polls the interfaces to see which ones require service, run as an ordinary process. This improves system consistency by reducing the number of exceptions to the scheduling rules.

2.1.4.2 Interrupts

The M6800 processor does not provide any hardware for prioritizing or vectoring interrupts. All devices share a common interrupt request line normally held at +5 volts (a logic "1"). An interrupt is caused by a device grounding this common line, at which point the processor stacks all registers, and jumps to the location it recovers from the top two bytes of its address space (usually contained in read only memory). The determination of the cause of the interrupt must then be done by code which interrogates device status registers. This iterrogation is usually referred to as "polling", and it is only at this point that priorities are normally introduced (either implicitly by the polling order, or explicitly as in the system under discussion).

Both for aesthetic reasons, and practical ones (i.e. simplicity of design, maintenance, and learning), it was decided to translate hardware interrupt signals into software semaphore signals as early as possible. This was done by having an interrupt poller process run as an ordinary system process, at infinite priority. Thus, when an interrupt occurs, all that is necessary is to make it seem as though the running process had signalled the semaphore assigned to the interrupt

poller process. So simple is this to do, that it takes only two machine instructions to achieve.

At this point, the interrupt poller can interrogate each device capable of generating an interrupt and signal a special handler process for each device. Device priority is explicitly controlled by the general priority mechanism.

As pleasing as this system is aesthetically, it is quite possible that a device requiring high speed interrupt service cannot wait for the system overhead involved in running the interrupt poller and system dispatcher (twice).

A high-speed device can be checked and serviced before passing control to the system, storing details of the interaction in a set of memory locations. The code executed before the operating system's priority mechanism is imposed and the memory locations used can be conceptually considered to be another layer of device interface, providing a 'virtual device'. The higher level portion of the handler for the device in question (i.e. the normally scheduled handler process) can then use the assigned memory locations as though they were additional device registers. As an example of this, suppose some sort of input device presents unlatched data to a set of input lines for only a very

brief time after it generates an interrupt. When this device generates an interrupt, the data can be read from the input lines and stored in memory, along with a flag indicating that this has been done, before signalling the interrupt poller. The interrupt poller could then interrogate this software flag as though it were the device interrupt flag, signal the appropriate device handler, and turn off the flag. The handler could then move the data into whatever buffer was designated to hold it.

Although this technique can improve the time required to service a device, it cannot increase the average data rate that the system can handle (in fact it slightly decreases it). This type of flexibility must be provided within the system, however, especially considering the unpredictable and sometimes haphazard hardware arrangements that can arise in laboratory situations.

2.1.5 Kernel processes

Apart from the code required to implement the system calls ("p", "v", "start", "sleep", "who", and "kill"), and the dispatcher, several processes must be considered an integral part of the system kernel. Two of these are the idle process, and the interrupt poller, whose functions are discussed above. The two other system processes of interest are the clock interrupt handler, and the initialization process.

At an interrupt generated by the programmable timer used to maintain the system elapsed time, the clock process updates the system time, and activates all processes in the sleep queue whose wakeup time has come, by moving them into the ready queue. The current system time is obtained by simply adding the last time interval to the old time value. Sleeping processes which should be reactivated are discovered by scanning the sleep queue. Since this queue is kept sorted in order of increasing wakeup time, it is not always necessary to scan the entire queue. If the first process in the queue should be activated, then it is deleted from the sleep queue and moved to the ready queue. This is repeated until the first process in the queue is one which is not ready to reawaken, or the queue is empty, at which point the job is done. The next interval is then computed according to the same

algorithm used by the "sleep" call, and the timer restarted.

The priority of the clock process is 254- i.e. the highest priority less than infinity. Since the clock process can be interrupted, the interrupt is explicilty masked when manipulating system data structures. This is a compromise between being forced to wait until the clock handler finishes its job completely before accepting another interrupt, and risking damage to important system data structures.

The remaining important system process, the initialization process, is, as its name implies, responsible for starting the necessary application processes, and initializing semaphore values. The system initialization sequence is as follows:

(1) Important system variables on page zero are initialized.

(2) All process blocks are marked as free, and all semaphore counts set to zero.

(3) All queues are set to null.

(4) IRQ and SWI interrupt vectors are set to transfer control to the proper locations.

(5) The initialization process is started

with a priority of infinity (i.e. 255).

The initialization process must then start the interrupt poller and clock handler, configure the system for the user application, and commit suicide, since it is no longer needed.

2.2 System implementation

2.2.1 Development & maintenance

The system was originally developed on Concordia University's Control Data Cyber 176 computer using the Motorola supplied cross assembler. Testing and debugging was performed on one of the M6800's. Later in the project when an Apple II microcomputer system became available, and because of difficulties with the Cyber system, support was transferred to the Apple, using a cross assembler written for the purpose by the author.

The Apple computer with one disk drive has proven perfectly adequate for supporting the operating system. A 9600 baud serial line has been installed to link the Apple to the master LSI-11 computer. Object files generated on the Apple are transferred to the LSI-11, which in turn downloads the M6800 systems.

2.2.2 Debugging

The firmware used to load, test, and debug the OS software is the "Mikbug" monitor supplied by Motorola. The monitor provides routines to examine and change memory, set breakpoints, accept downloaded programs, and send "uploaded" dumps of memory regions. The read only memory containing the firmware also traps restart interrupts, and vectors NMI (non-maskable interrupts), IRQ (interrupt requests), and SWI (software interrupts) through locations in normal read-write memory so that they can be trapped by software other than the monitor. Although the breakpoint facility was somewhat useful in testing and debugging indiviual routines and isolated pieces of code, it proved virtually useless in testing the running system since its use of the software interrupt as a break instruction conflicted with the system's use of the software interrupt as the mechanism for making system calls (system calls are done by loading a function code into the A-register and generating a software interrupt). This made debugging very difficult until the system was runnable. The primary debugging techniques in the early stages were the use of carefully constructed test cases, analysis of the system state via examination of memory contents, and thinking.

Chapter 3

APPLICATION CONFIGURATION

3.1 Inter processor communications

Each M6800-based rat cage controller must communicate with the master LSI-11 computer to receive instructions and parameters, and to report results. Since all processors involved in the system are within several feet of each other within the same room, direct interconnection of RS-232 serial ports of the respective computers produces reliable communication on the hardware level, with a low probability of error. It was decided, nevertheless, that some error checking should be built into the communication, and so a simple packet system was designed for the purpose.

Low overhead, in both the implementation time and usage of resources when running, was and is important- so the packet system has been kept quite simple. A packet consists of:

1) A start character (arbitrarily chosen to be an ASCII colon).

2) A count of the number of data characters in the packet (0..16).

3) The data.

4) A one-byte checksum such that the sum of the count, all data bytes, and the checksum will be zero (mod 256).

Packet transmission is accomplished by a simple protocol:

1) The processor wanting to transmit a packet sends a "ready-to-send" character.

2) The sender waits until the receiver responds with a "ready-to-receive" character.

3) The sender then transmits the packet and waits for an acknowledgement.

4) If the sender receives a positive acknowledgement, then the transfer is complete; if the sender receives a negative acknowledgement, the procedure begins again at step 1.

Until the transfer is complete, the process at whose request the data is being sent remains blocked. The data rate was established by trial and error, and it was found necessary to introduce a small inter-character delay in the LSI-11's transmission to allow time for overhead (the 6800 is not known for its speed).

3.2 The experiment

3.2.1 Format

A single experimental trial consists of:

1) A priming stimulation intended to bring the rat into a known state before the trial proper starts.

2) A period during which the rat's lever is armed and it can obtain stimulation by pressing the lever. During this period both lever presses and stimulations must be counted (these may be different, since a lever press during a stimulation does not trigger another stimulation).

3) A period during which stimulation is not available.

The parameters of the pulse train used for priming generally differ from those used for self-stimulation. Therefore two sets of parameters are necessary for each trial. A set of parameters for a stimulation (either priming or self-stimulation) is handled as a block, and refered to as a "stimulation parameter block" (abreviated SPB).

3.2.2 Commands

An experiment is driven by commands issued by the LSI-11. A command consists of a single byte code, followed by any data required for the performance of the command. The cage controller system always acknowledges commands by repeating the command being acknowledged to the master system, followed by any data which the command requires be sent.

There are currently five commands in the repertoire, described in the following paragraphs.

3.2.2.1 The "null" command

The null command does nothing other than acknowledge execution. It is useful in verifying that the system has not crashed or deadlocked, and proved useful in debugging the LSI-11 end of the communication system.

3.2.2.2 The "accept" command

New stimulation parameter blocks are sent by issuing an "accept" command, followed by the priming SPB, and the self-stimulation SPB. Both are stored internally in a common area to be referenced implicitly by other commands.

3.2.2.3 The "echo" command

As an additional verification, the master can request that the SPB's stored by a cage controller be repeated to it for comparison to the originals. This command causes both SPB's to be sent out from the M6800 exactly as they were received. This facility also makes it easier for the LSI-11 to recover from a crash without losing the last experiment (since the cage controller remembers the parameters for it).

3.2.2.4 The "report" command

This command causes two two-byte numbers representing the lever press count, and the stimulation count to be reported. These counts are the data gathered from a trial.

3.2.2.5 The "cancel" command

Should a need arise to cancel a trial (perhaps the rat is having a seizure) before its time elapses normally, the cancel command has the effect of terminating the trial.

3.2.2.6 The "begin" command

The command to begin a trial has the most complex action, and so has been saved for last. All other commands execute in a comparatively short time, and therefore are implemented as simple subroutines of the executive process. The running of a trial, however, is sufficiently complex and lengthy to warrant a process (or, rather, a set of processes) of its own. This allows the executive process to remain responsive to commands while a trial is in progress. It is in this way that the current trial can be cancelled.

There are two semaphores associated with a trial. One, named "SMSGO" is used to trigger a stimulation sequence. The second, named "SMSLCK", ensures that

stimulations are atomic. (A cancel command is the exception to the atomicity of stimulations, and can cause a halt in mid-stimulation.) These are discussed further in the descriptions of the trial processes below.

3.2.2.6.1 The lever interrupt handler

The lever interrupt handler is activated whenever the rat presses down on the bar in its cage. It then consults a pair of status flags which specify which of three actions it is to take:

1) It can ignore the lever press.

2) It can increment the lever press count, but not trigger a stimulation.

3) It can increment both the lever press count and the stimulation count and trigger a stimulation.

3.2.2.6.2 The trial process

The trial process is responsible for sequencing and timing events within the trial (distinct from events within a stimulation). It selects the approriate SPB for each step, triggers priming, and for the trial duration it enables lever press and stimulation counting, and goes to sleep, leaving the stimulation lock (SMSLCK) open so that the stimulation process can be triggered by the lever interrupt process. At the end of the trial interval, it disables further stimulation and lever press counting, and acknowledges the "begin" command to the master.

The acknowledgement of the command to begin a trial signals to the master that the trial is complete and the data may now be examined. Its job finished, the trial process kills the stimulation process, and itself.

3.2.2.6.3 The stimulation process

Once it is triggered, and permission to stimulate unlocked, the stimulation process generates the sequence of pulse pair trains specified by the (currently selected) parameter block (SPB). It is responsible for configuring the hardware involved in pulse generation, counting and timing pulse pairs and trains, and keeping count of lever presses during the critical C-C interval, when interrupts are masked.

Stimulation is triggered via the SMSG0 semaphore from only two places: 1) the trial process (for priming), and 2) the lever interrupt process (if the appropriate flag is set- recall that the lever interrupt process's ability to act as a trigger can be disabled). The semaphore SMSLCK serves to synchronize the trial process to the end of a stimulation.

Chapter 4

SOME FINAL COMMENTS

4.1 The 6800 processor

The experience provided by this project in the use of the 6800 processor emphatically underlined some of its weaknesses. Several important capabilities are missing from the 6800- some so obvious that their omission is puzzling.

Chief among these is the missing ability to push and pop the X register (the sole index register) on the processor stack. It was also noted that the form which indexing takes in the 6800 makes it difficult to address a location using a computed offset from the index register (the ability to specify one or both of the accumulators as an offset to the index register would have been useful).

Motorola must have been aware of the weaknesses of the 6800, for they fixed these problems, and added many other features in their next major eight-bit processor- the 6809. The 6809 architecture is extremely attractive, but still misses one feature the lack of which was felt in this operating system. This is the ability to have the hardware transfer control to an interrupt processor without the need for polling. This

would require that each hardware interrupt have associated with it an address and a priority, and was probably not possible with the technology available when the 6800 was designed, though a common feature on larger computers.

4.2 System design

In general, the operating system seems to achieve the goals of cleanliness and simplicity of design and the author is satisfied with it.

The configuration of processes running the application is also satisfactory. It has been changed several times usually in small ways, with one larger organisational change. This larger change was the decision to run a trial process distinct from the command executive process, thus allowing the addition of the "cancel" command.

The weakest part of the entire system is the packet communication subsystem. In retrospect, and in view of the amount of effort spent on inter-computer communications in general ([Tan81]), some of the difficulties which this subsystem can encounter could have been forseen- but they were not. It is possible, given the right conditions, for the existing communication system to get itself into serious trouble. For example, an incorrect data byte count can

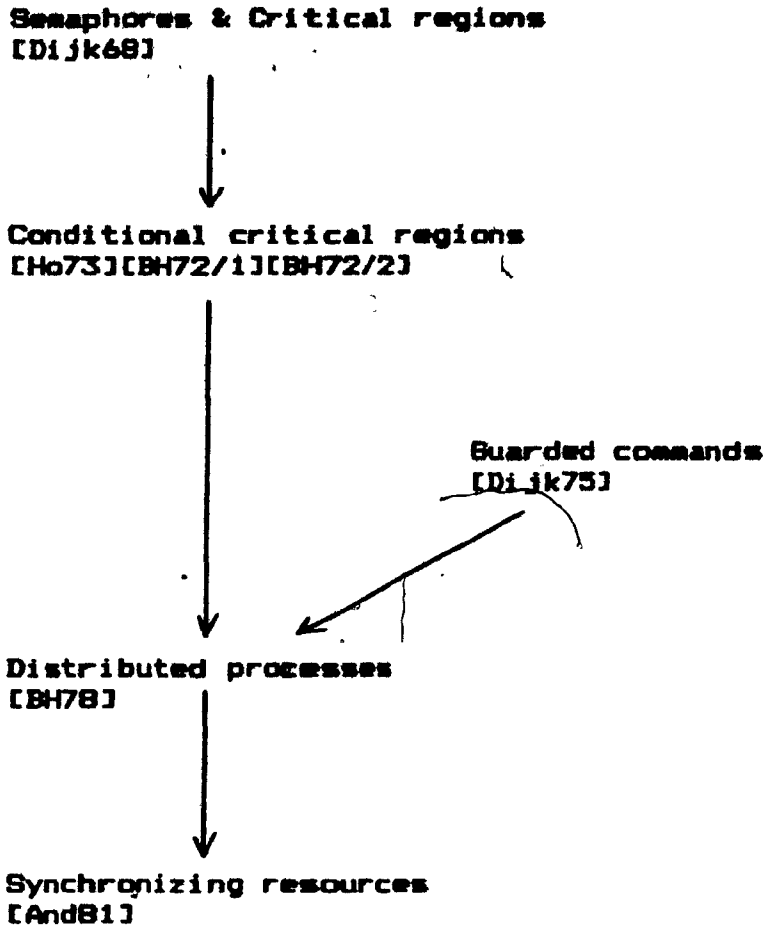
result in the receiver waiting for some characters which will never arrive. If another packet is then sent soon after the first, the two computers can get out of phase with their packet transfers- resulting in, at the very least, a lost packet. What is more, this problem cannot be eliminated completely by simply timing out packets, since the time to the next packet might be less than a reasonable time-out interval. The same type of problem can arise if several characters are lost due to overflow of the character buffer in the interface chip. (The chip can indicate that a character was lost- but not how many.) This problem could reasonably be expected to eventually occur (it has not happened yet, to anybody's knowledge) since interrupts are masked during C-C intervals.

The establishment of more reliable communications would be made easier if the RS-232 standard lines were more completely implemented in the hardware. As it is only the three most essential lines (transmit data, receive data, and signal ground) were used in connecting the computers. The lines which establish a character by character handshaking protocol would have been invaluable.

Although the overall system has not yet been used in full scale experiments (it will eventually include eight rat cages), it is currently being used with

two rats while experimentation takes place with how to make the best use of the system. New algorithms (for the selection of stimulation parameters based on the rat's behaviour), which were too complex to run manually, can now be tried to speed up the gathering of useful data.

The system works- and the people who have spent many hours staring at a rat pressing on a bar in a small cage are happy about that.


Semaphores & Critical regions
[Dijk68]
Conditional critical regions
[Ho73][BH72/1][BH72/2]
Guarded commands
[Dijk75]
Distributed processes
[BH78]
Synchronizing resources
[And81]

Chapter 5

A BRIEF REVIEW OF CONCURRENT PROGRAMMING

The purpose of this section is to sketch the most important landmarks in the development of multi-tasking systems design (of which the subject of this thesis is an example) in the opinion of the author. The diagram opposite presents the apparent ancestral relationships of the programming concepts chosen for discussion.

5.1 Semaphores & critical regions

The most elementary, and the earliest, techniques are the semaphores and critical regions, both presented in [Dijk68]. Although not identical, since critical regions directly implement only the mutual exclusion aspect of semaphores, these two concepts are of much the same level of sophistication. It seems most likely, however, that semaphores were originally conceived with the mutual exclusion problem in mind, then abstracted to a more general signalling concept. Both can be implemented quite simply, but have some serious limitations in use, and tend to result in a complexity analogous to that caused by the "goto" construct in sequential programming. Both are elegant in their simplicity (the prime requirement for being included in this review).

5.2 Conditional critical regions

The next step, taken by Hoare ([Ho73], [BH72a], and [BH72b]), was to attach a boolean expression to a critical region, describing the condition(s) which should be true for the critical region to begin (end) execution. This adds the signalling ability of the abstract semaphores to the critical region structure-but in an improved and simplified form. One can quite easily program, using the boolean expression of a critical region, tasks which require considerably more complexity to accomplish with only the semaphore primitives (although this can always be done, since it is possible to simulate conditional critical regions using semaphores). It also results in a program text which is more readable, whose behaviour is easier to verify, and it maintains the structured form of the critical region.

5.3 Monitors

The monitor concept originated by Brinch-Hansen ([BH73], [Ho74]) makes another step by suggesting the encapsulation of procedures which manipulate a shared resource or set of resources. In this scheme all communication and synchronization between customer processes is done via these common monitors, and no shared variables can exist outside of a monitor. Mutual exclusion is enforced within a monitor, in as much as a monitor may only be executing on behalf of one process at any instant (other processes may be suspended- their status to be re-evaluated as soon as some monitor procedure completes). Hoare proves, in [Ho74], that monitors can be simulated with semaphores and vice-versa (thus the improvement over semaphores is one of understandability and verifiability rather than one of power).

5.4 Distributed processes

In Brinch-Hansen's next major contribution, the concepts introduced in [BH78] ("distributed processes"), a new influence is apparent- that of Dijkstra's guarded command programming constructs ([Dijk75]). These constructs, although not originally intended for concurrent programming applications, proved such an elegant way of introducing non-determinism into sequential programming, that it now seems quite natural, in retrospect, that they should be applied to the concurrent programming case where non-determinism is inherent.

The fundamental entities described in [BH78] are processes. Displaying a surface similarity to monitors in the sense that all interactions between processes are done via procedure calls (and their parameters) and that shared variables are encapsulated within a fundamental entity- the difference is nevertheless great. The flow of control which passes through a monitor is always that of one of its customer processes, i.e. the monitor is not a process in its own right, but only a sort of common meeting ground. In contrast, Brinch-Hansen's distributed processes communicate directly by calling each other's procedures, while within the code for a process the functions of synchronization and signalling are

achieved by a combined adaptation of both guarded commands and conditional critical regions (embodied in the "when" and "cycle" statements). The result is very aesthetically pleasing, and, due to the relative independence of processes, is particularly well suited to systems involving a number of loosely-coupled processors (i.e. processors which do not share a common memory).

5.5 Synchronizing resources

The final contribution to be discussed here is that of Andrews in [And81] (it might be useful to mention that this paper has an extensive and thorough bibliography). Andrews re-introduces variables shared among several processes, but uses the concept of a "resource" to encapsulate a set of processes and common variables to which all processes within the set have free access. Communication (other than via shared variables within a resource) is again accomplished via calls, however here the calling mechanism provides for the case in which the calling process does not wish to wait for its call to be processed before proceeding (this is accomplished by using the keyword "send" instead of "call", and is clearly similar to the message passing primitives used in some operating systems).

Synchronization is performed by the "in" statement, which again strongly resembles Dijkstra's guarded command statements. It is within the guards of the "in" statement that calls (of either type) are accepted. An interesting addition here is that as well as entries and boolean synchronizing expressions, the guards of "in" statements can also contain explicit scheduling information. An arithmetic expression can be specified with respect to which multiple pending calls of an operation can be ordered. Andrews' proposal seems to provide considerable flexibilty, while still remaining readable.

5.6 Conclusion

Clearly, in this brief review of concurrent programming concepts many peoples' work has been omitted. Any such summary, however, requires a selection, and a process of selection must necessarily, and indeed should, reflect the opinion of the selector. The above selection of concepts and ideas seem, to this author, the most elegant proposals for usable concurrent programming languages.

Bibliography

[And81] Andrews, G. R., "Synchronizing resources", ACM Transactions on Programming Languages, Vol. 3, No. 4, 405-430, Oct. 1981

[BH72a] Brinch-Hansen, P., "Structured multi-programming", CACM, Vol. 15, No. 7, 574-578, July 1972

Introduces conditional critical regions.

[BH72b] Brinch-Hansen, P., "A comparison of two synchronizing concepts", Acta Informatica 1, 190-199, Springer-Verlag, 1972

A comparison of critical regions with conditional critical regions.

[BH73] Brinch-Hansen, P., "Operating systems principles", Prentice-Hall, Englewood Cliffs, New Jersey, July 1973

Introduces the concept of monitors.

[BH78] Brinch-Hansen, P., "Distributed processes: a concurrent programming concept", CACM, Vol. 21, No. 11, 934-941, Nov. 1978

[Dijk68] Dijkstra, E. W., "Cooperating sequential processes" Programming Languages, F. Genuys (Ed.), Academic Press, New York, New York, 1968

Introduces the concepts of concurrent statements, semaphores, and critical regions.

[Dijk75] Dijkstra, E. W., "Guarded commands, nondeterminacy and formal derivation of programs", CACM, Vol. 18, No. 8, 453-457, Aug. 1975

[GSY81] Gallistel, C. R., Shizgal, P., Yeomans, J. S., "A portrait of the substrate for self-stimulation", Psychological Review, Vol. 88, No. 3, 228-273, 1981

A survey paper describing and justifying neuro-psychological experimental techniques.

[Ho73] Hoare, C. A. R., "Towards a theory of parallel programming", Operating Systems Techniques, C. A. R. Hoare and R. H. Perrott (Eds.), Academic Press, New York, New York, 1973

Introduces the concept of conditional critical regions.

[Ho74] Hoare, C. A. R., "Monitors: an operating system structuring concept", CACM, Vol. 17, No. 10, 549-557,

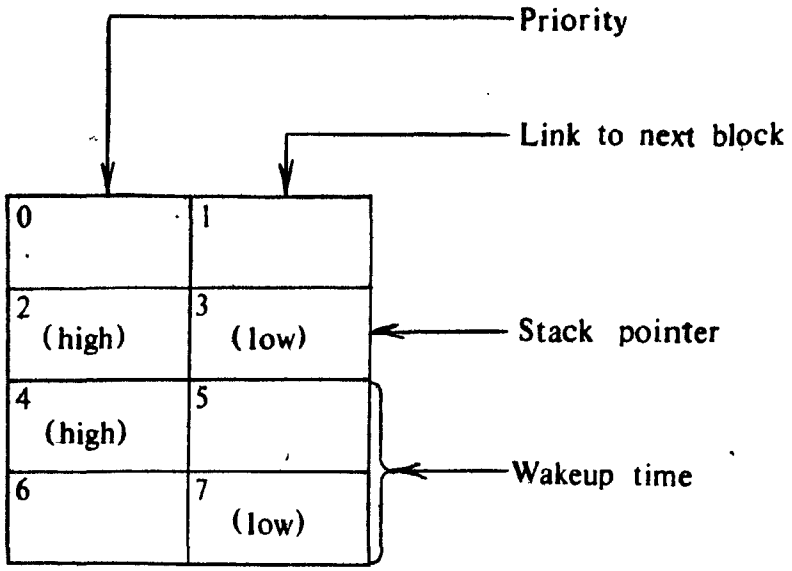
Oct. 1974

[Tan81] Tanenbaum, A. S., "Network protocols", ACM Computing Surveys, Vol. 12, No. 4, 453-489, Dec. 1981

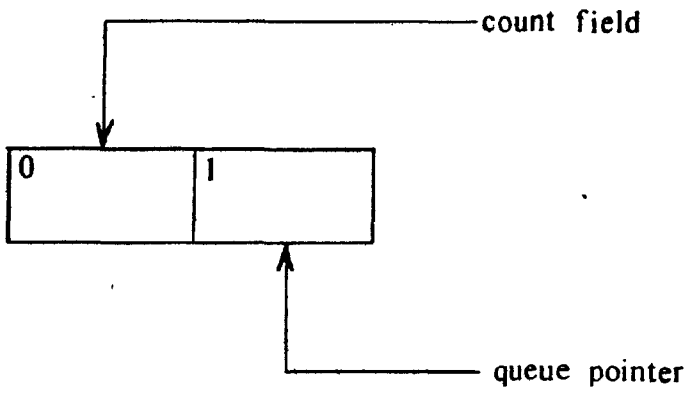
A survey of network protocols using the ISO/OSI model.

Appendix A:

Illustrations



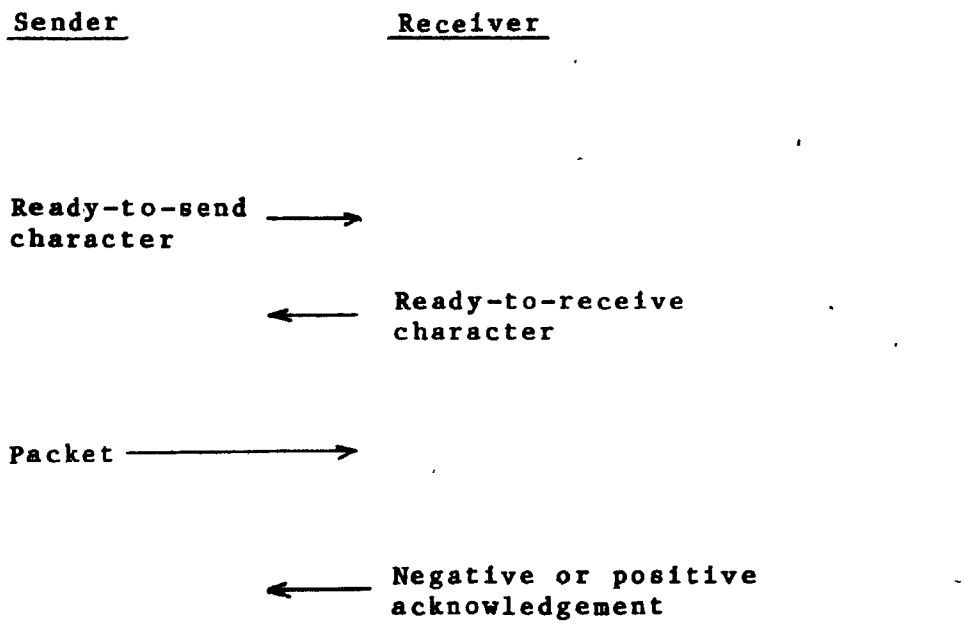
Process block



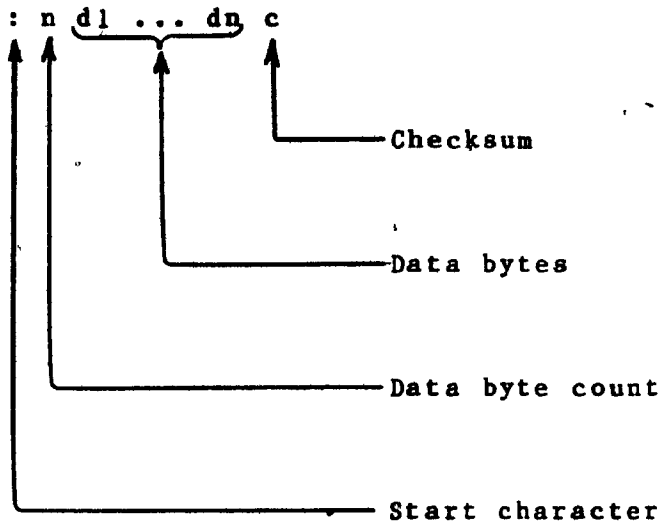
Semaphore block

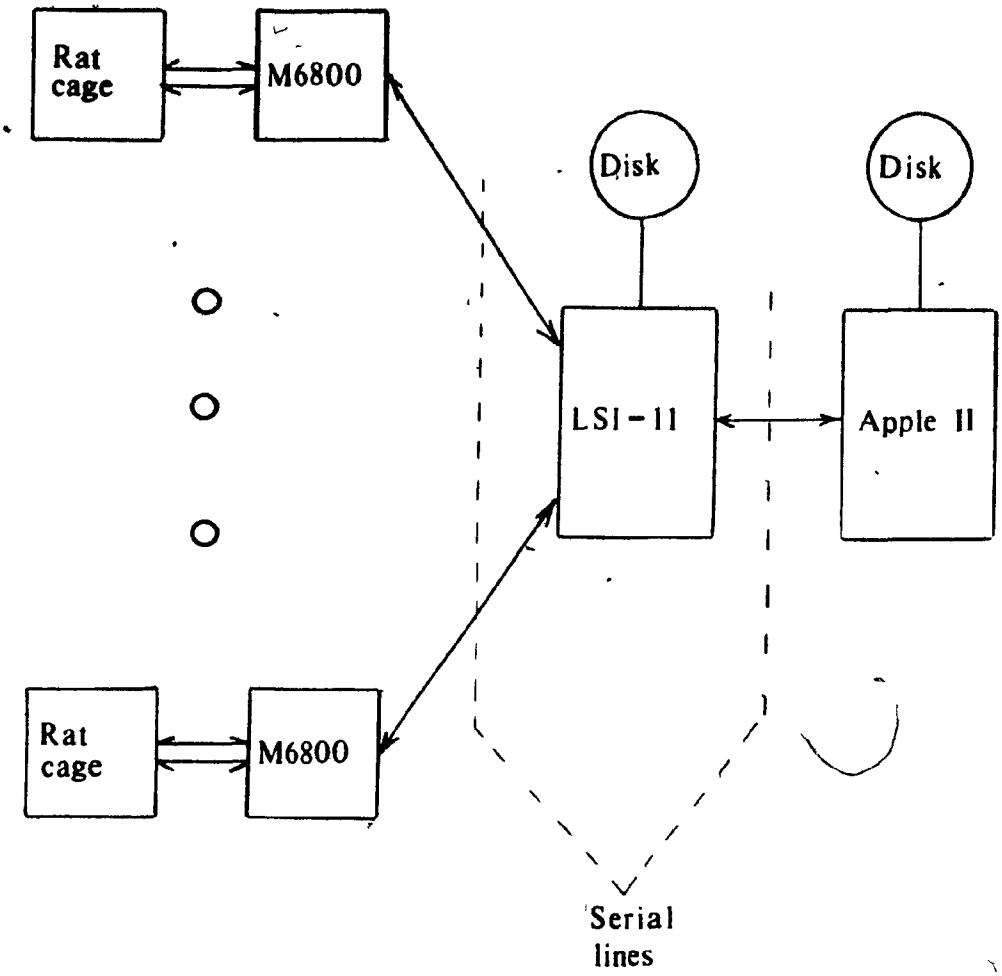
Appendix A: Illustrations

Packet transmission protocol



Packet format





System configuration

Appendix B:

Source code listing

```
* FILE: PAGE0
*
****************************************
* OS KERNEL FOR THE M6800 UPROCESSOR
*
*              I. SHIZGAL
*
****************************************
*--------EQUATES
****************************************
*
* SEMAPHORE ASSIGNMENTS
*
* #0 IS PERMANENTLY ASSIGNED TO THE INTERRUPT POLLER
* #1 IS PERMANENTLY ASSIGNED TO THE SYSTEM CLOCK
SMRCVR  EQU     2       RECEIVER HANDLER - GO
SMXMTR  EQU     3       XMITTER HANDLER - GO
SMRCVF  EQU     4       RECEIVER - INPUT BFR FULL
SMRCVE  EQU     5       RECEIVER - INPUT BFR EMPTY
SMXMTF  EQU     6       XMITTER - OUTPUT BFR FULL
SMXMTE  EQU     7       XMITTER - OUTPUT BFR EMPTY
SMSLCK  EQU     8       STIMULATION LOCKOUT SEMAPHORE
SMSGO   EQU     9       START STIMULATION SEMAPHORE
SMPIA1  EQU     10      PIA0-CA1 INTERRUPT SIGNAL
SMPIA2  EQU     11      PIA0-CB1 INTERRUPT SIGNAL
SMPCKO EQU      12      PACKET OUTPUT ACCESS LOCK
*
*
*
PRIO    EQU     0       DISPLACEMENT OF  PRIORITY  FIELD  IN  PROC.
BLOCK
LINK    EQU     1       DITTO - FOR LINK FIELD
SP      EQU     2       DITTO - FOR STACK POINTER FIELD
*
* LOCATIONS OF PROCESS BLOCK AND SEMAPHORE BLOCK MEMORY
* BOTH ARE $100 BYTE FIELDS, AND MUST START AT A $100 BOUNDARY
* THERE ARE 32 PROCESS BLOCKS, EACH 8 BYTES, NUMBERED 0..31
* THERE ARE 128 SEMAPHORE BLOCKS, EACH 2 BYTES, NUMBERED 0..127
* PROCESS BLOCK: BYTE 0: PROCESS PRIORITY (1..255), 0=>DEAD
*        BYTE 1: LINK USED WHEN BLOCK IS IN A QUEUE
*              THIS IS THE LOW-ORDER ADDR., SINCE THE
*              HIGH BYTE IS THE SAME FOR ALL
*        BYTE 2-3: PROCESS'S STACK POINTER
*        BYTE 4-7: WAKEUP TIME
* SEMAPHORE BLOCK: BYTE 0: SEMAPHORE COUNT
*        BYTE 1: SEMAPHORE QUEUE ($FF => NULL)
*              USAGE SAME AS LINK FIELD OF PB
PBS     EQU     $100
SBS     EQU     $200
*
*
* IMPORTANT ADDRESSES IN MONITOR'S RAM
IOV     EQU     $380    LOCATION OF VECTOR TO USER IRQ ROUTINE
```

```
SWI1    EQU     $38A    LOCATION OF VECTOR TO MAIN SWI ROUTINE
SWI2    EQU     $38C    LOCATION OF VECTOR TO USER SWI ROUTINE
*
*
*
* HARDWARE REGISTER DEFINITIONS
*
*       ACIA
ACIAS   EQU     $EC14   ACIA STATUS-CONTROL REGISTER
ACIAD   EQU     $EC15   ACIA DATA REGISTERS
*
*       6840 TIMER
*       EACH CHIP USES 8 ADDRESSES
TIMERA  EQU     $EC18
*
*       PIA
*       EACH CHIP USES 4 ADDRESSES
PIA0    EQU     $EC10
*
*
*
* DIGITAL TO ANALOG CONVERTER(S)
DAC1    EQU     $ED00
DAC2    EQU     $ED02
*
*
**************************************
*----------PAGE ZERO
**************************************
*
        ORG     0
*
        JMP     $800    INIT. SYSTEM
GETCH   RMB     3       VECTOR TO CHAR INPUT ROUTINE
PUTCH   RMB     3       VECTOR TO CHAR OUTPUT ROUTINE
*
* JMP VECTORS FOR SYSTEM ROUTINES
INSERT  RMB     3
DELETE  RMB     3
*
PBAREA  RMB     1       HIGH BYTE OF PBS
PBFREE  RMB     1
SBAREA  RMB     1       HIGH BYTE OF SBS
SBFREE  RMB     1
PBCRNT  RMB     1       POINTER TO PB OF CURRENTLY RUNNING PROCESS
READY   RMB     1       ROOT OF READY QUEUE
TIMER   RMB     1       ROOT OF TIMER QUEUE
*
TMPX    RMB     1       TEMPORARIES USED BY SYSTEM
TMP1    RMB     1
TMP2    RMB     1
TMP3    RMB     1
TMP4    RMB     2
```

```
*
SYSTIM  RMB     4       SYSTEM ELAPSED TIME CLOCK
LASTDT  RMB     2       LAST TIMER INTERVAL
*
*
*
*
* CONTROL REGISTER CONTENTS FOR DEVICES.
*
*       6840 TIMER CONTROL REGISTERS
*
CR1A    RMB     1       CONTROL REGISTERS FOR
CR2A    RMB     1       FIRST 6840
CR3A    RMB     1
*
*
*
*       ACIA STUFF
ACIASV  RMB     1       ACIA CONTROL REGISTER CONTENTS
ACIAVD  RMB     1       ACIA VIRTUAL DATA REGISTER
*
*
*       PIA CONTROL REGISTERS
PIA0RA  RMB     1       PIA 0, REG A
PIA0RB  RMB     1       PIA 0, REG B
*
*
* CHARACTER INPUT AND OUTPUT BUFFERS
* FOR ACIA HANDLERS
RCVBF   RMB     1
XMTBF   RMB     1
*
*
*
* DAC REGISTERS
DAC1R   RMB     2
DAC2R   RMB     2
```

```
* FILE: STACKS
*****************************
        ORG     $400
*****************************
* ENTRY POINT/PRIORITY TABLE FOR
* PROCESS INITIALIZATION
*
XIDLE   RMB     2
        FCB     1       PRIO
XINITG  RMB     2
        FCB     255     PRIO
XINTPO  RMB     2
        FCB     255     PRIO
XCLOCK  RMB     2
        FCB     254     PRIO
XXMTR   RMB     2
        FCB     200     PRIO
XRCVR   RMB     2
        FCB     201     PRIO
XEXAMI  RMB     2
        FCB     190     PRIO
XCMND   RMB     2
        FCB     50      PRIO
XTRIAL RMB      2
        FCB     60      PRIO
XSTIM   RMB     2
        FCB     60      PRIO
XLVR1   RMB     2
        FCB     200     PRIO
*
*****************************
* IDLE PROCESS STACK
        RMB     6
IDLESP  RMB     8
*
*--------------------
*INITIALIZATION PROCESS STACK
        RMB     20
INITSP  RMB     8
*
*--------------------
* INTERRUPT PROCESS STACK
        RMB     20
INTPOS  RMB     8
*
*--------------------
* CLOCK PROCESS STACK
        RMB     20
CLOCKS  RMB     8
*
*--------------------
* ACIA XMITTER PROCESS STACK
        RMB     20
```

```
XMTRSP  RMB     8
*
*---------------
* ACIA RCVR PROCESS STACK
        RMB     20
RCVRSP  RMB     8
*
*---------------
*EXAMIN PROCESS STACK
* STACK
        RMB     30
EXAMSP  RMB     8
*
*---------------
* END OF SYSTEM PROCESS STACKS
*
* BEGINNING OF APPLICATION PROCESS
* STACKS
*---------------
* COMMAND PROCESS STACK
        RMB     30
CMNDSP  RMB     8
*
*---------------
* TRIAL PROCESS STACK
        RMB     30
TRIALS  RMB     8
*---------------
* STIMULATOR PROCESS STACK
        RMB     30
STIMSP  RMB     8
*
*---------------
* LEVER1 PROCESS STACK
        RMB     20
LVR1S   RMB     8
*---------------
* LEVER2 PROCESS STACK
        RMB     20
LVR2S   RMB     8
*
*---------------
* STACK FOR EXAMIN'S SON
EXMSON  EQU     $800-8
```

```
* FILE: KERNEL
**************************************
*----------SYSTEM CODE
**************************************
*
*
        ORG     $800
*
*-----------------------------------
* SYSTEM STARTUP CODE
*
*
* INITIALIZE SYSTEM ROOT BLOCK
        LDAA    #PBS/256
        STAA    PBAREA
        LDAA    #SBS/256
        STAA    SBAREA
        LDAA    #$FF
        STAA    READY   READY:=NULL
        STAA    TIMER   TIMER:=NULL
        CLRA
        STAA    SYSTIM  CLEAR SYSTEM TIME
        STAA    SYSTIM+1
        STAA    SYSTIM+2
        STAA    SYSTIM+3
*
* CLEAR ALL PROCESS BLOCKS
        CLRA    LOOP COUNTER
        LDX     #PBS    POINTER TO AREA
INIT2   CLR     0,X
        INX
        INCA
        BNE     INIT2
*
* CLEAR ALL SEMAPHORE BLOCKS
        LDAB    #128    LOOP COUNTER
        LDX     #SBS    START OF SEMAPHORE AREA
        LDAA    #$FF    NULL POINTER CONSTANT
INIT1   CLR     0,X     CLEAR SEM.COUNT
        STAA    1,X     SEM.QUEUE:=NULL
        INX     NEXT
        INX
        DECB    DONE?
        BNE     INIT1
*
* INITIALIZE USER VECTORS FOR IRQ AND SWI
* IN MONITOR'S SPACE
*
        LDX     #IRQINT
        STX     IOV
        LDX     #SWITRP
        STX     SWI1
        STX     SWI2
```

```
*
* START UP INITIALIZATION PROCESS
        LDX     #PBS    USE PB# 0
        LDAA    XINITS  STACK POINTER
        STAA    SP,X
        LDAA    XINITS+1
        STAA    SP+1,X
        LDAA    #255    GIVE INTITIALIZATION HIGHEST PRIORITY
        STAA    PRIO,X
        LDX     XINITG  INITIALIZE PC
        STX     INITSP+6
        CLRA    A:=POINTER TO PB# 0
        LDX     #READY
        JSR     INSERT  INSERT IT IN READY QUEUE
        JMP     DISPAT  AND GO!
*
*
*
XINITS FDB      INITSP
*-------------------------------
* SWI TRAP INTERPRETTER FOR SYSTEM CALLS
* ALL SYSTEM CALLS ARE MADE VIA THE "SWI" INSTRUCTION.
*  INPUTS:  A  =  CODE  SPECIFYING  WHICH  SYSTEM  CALL  IS  BEING
PERFORMED.
* OTHER INPUTS AND OUTPUTS ARE SPECIFIED INDEPENDENTLY FOR EACH
* SYSTEM CALL.
SWITRP  CMPA    #6
        BCS     *+3
        RTI
        ASLA    FUNCTION CODE IS IN A - WANT TO USE AS TABLE INDEX
        ADDA    XSWITB+1
        STAA    TMPX+1
        CLRA
        ADCA    XSWITB
        STAA    TMPX
        LDX     TMPX    X:=ADDRESS OF ROUTINE ADDRESS
        LDX     0,X
        JMP     0,X
*
* SWI SYSTEM ROUTINE ADDRESS TABLE
XSWITB FDB      SWITAB
SWITAB  FDB     XP
        FDB     XV
        FDB     XSTART
        FDB     XSLEEP
        FDB     XWHO
        FDB     XKILL
*
*
*
*-------------------------------
* DISPATCHER
* -RESUME TOP PRIORITY PROCESS IN READY QUEUE
```

```
* -THIS ROUTINE IS ALWAYS 'JMP'ED TO, NEVER CALLED
* (NO PARAMETERS)
*
DISPAT  LDX     #READY
        JSR     DELETE  DELETE TOP PB FROM READY
        STAA    PBCRNT  INSTALL IT AS THE CURRENT PB
        STAA    PBFREE
        LDX     PBAREA  X:=ADDR OF BLOCK
        LDS     SP,X    RESTORE PROCESS'S STACK POINTER
        PULA
        LDAB    PRIO,X  GET PRIORITY
        INCB
        BEQ     DSPTR IF IT IS 255 THEN DISABLE INTERRUPTS
        ANDA    #$EF    ELSE ENABLE THEM
        PSHA
        RTI
DSPTR   ORAA    #$10    DISABLE INTERRUPTS
        PSHA
        RTI
*
*
*
*-------------------------------
* INSERT
* ROUTINE FOR SYSTEM USE.
* INPUTS: A = POINTER TO PB TO BE INSERTED
*       X = ADDRESS OF ROOT POINTER OF QUEUE
*       INTO WHICH THE PB IS TO BE INSERTED
*       IN ORDER OF PRIORITY.
* OUTPUTS: NONE
* A,B PRESERVED
*
XINSER  EQU     *
        ORG     INSERT
        JMP     XINSER
        ORG     XINSER
*
        PSHB    SAVE REGISTERS
INS3    PSHA
        STX     TMPX
        LDAB    0,X     IF
        INCB    QUEUE IS EMPTY
        BEQ     INSR1   THEN GO DO INSERTION
* ELSE COMPARE INPUT PB'S PRIORITY WITH THAT OF TOP ELEMENT
        DECB
        STAB    PBFREE
        LDX     PBAREA
        LDAB    PRIO,X  B:=TOP ELEMENT'S PRIORITY
        STAA    PBFREE
        LDX     PBAREA
        LDAA    PRIO,X  A:=INPUT PB'S PRIORITY
        CBA
        BLS     INSR2   INSERT HERE ONLY IF INPUT PRIO> TOP
```

```
*        ELEMENT'S PRIO
* INSERT INPUT PB AT THIS POINT
INSR1   PULA    A:=INPUT PB ADDRESS
        LDX     TMPX    X:=QUEUE ROOT ADDRESS
        LDAB    0,X     B:=POINTER TO TOP ELEMENT
        STAA    0,X     MAKE INPUT PB NEW TOP ELEMENT
        STAA    PBFREE
        LDX     PBAREA
        STAB    LINK,X  COPY OLD TOP ELEMENT POINTER INTO
*        INPUT PB'S LINK FIELD
        PULB
        RTS     AND RETURN
INSR2   LDX     TMPX    ADVANCE ROOT POINTER TO LINK FIELD
        LDAB    0,X     OF TOP ELEMENT
        STAB    PBFREE
        LDX     PBAREA
        INX     X:=ADDRESS OF NEW ROOT
        PULA    A:=POINTER TO INPUT PB
        BRA     INS3    THIS IS REALLY A RECURSION
*
*
*
*-------------------------------
* DELETE
* ROUTINE FOR SYSTEM USE.
* INPUTS: X = ADDRESS OF QUEUE ROOT POINTER
* OUTPUTS: A = PB POINTER OF DELETED (IE. FIRST)
*       QUEUE ELEMENT
* B,X PRESERVED
*
XDELET  EQU     *
        ORG     DELETE
        JMP     XDELET
        ORG     XDELET
*
        PSHB    SAVE REGISTERS
        STX     TMPX
        LDAB    0,X     GET QUEUE ROOT
        STAB    PBFREE
        LDX     PBAREA  X:=POINTER TO TOP ELEMENT
        LDAB    LINK,X  B:=LINK FIELD OF TOP ELEMENT
        LDX     TMPX    RECOVER X
        LDAA    0,X     A:=POINTER TO TOP ELEMENT
        STAB    0,X     UPDATE QUEUE ROOT
*        (ROOT=LINK FIELD OF TOP ELEMENT)
        PULB    RECOVER B
        RTS
*
*
*
*-------------------------------
* P
* SYSTEM CALL.
```

```
* INPUTS: A = 0
*       B = SEMAPHORE # : 0..127
* OUTPUTS: NONE
* ALL REGISTERS PRESERVED
*
XP      ASLB    CONVERT SEM# TO OFFSET
        STAB    SBFREE
        LDX     SBAREA  X:=SEM ADDRESS
        TST     0,X     IS SEM COUNT=0?
        BEQ     PFAIL   YES - CALLER MUST WAIT
        DEC     0,X     NO - DECREMENT SEM COUNT
        RTI     AND RESUME CALLER
PFAIL   LDAA    PBCRNT
        STAA    PBFREE
        LDX     PBAREA  X:=ADDRESS OF CURRENT PB
        STS     SP,X    SAVE CALLER'S STACK
        LDX     SBAREA
        INX     X:=ADDRESS OF SEM QUEUE
        JSR     INSERT  ENQUEUE CALLER IN SEM QUEUE
        JMP     DISPAT  AND DISPATCH
*
*
*
*--------------------------------
* V
* SYSTEM CALL.
* INPUTS: A = 1
*       B = SEMAPHORE # : 0..127
* OUTPUTS: NONE
* ALL REGISTERS PRESERVED
* IF THE PRIORITIES OF THE CALLER AND THE
* TOP PROCESS IN THE SEMAPHORE QUEUE ARE
* EQUAL, THEN THE PROCESS WHICH HAS BEEN
* WAITING WILL BE THE ONE TO RUN FIRST.
*
XV      ASLB    CONVERT SEM# TO OFFSET
        STAB    SBFREE
        LDX     SBAREA  X:=ADDRESS OF SEM
        LDAA    1,X     IS SEM QUEUE EMPTY?
        INCA
        BNE     VGO     NO
        INC     0,X     YES - INCREMENT SEM COUNT
        RTI     AND RESUME CALLER
VGO     LDAA    PBCRNT
        STAA    PBFREE
        LDX     PBAREA  X:=ADDRESS OF CURRENT PB
        STS     SP,X    SAVE CALLER'S STACK
        LDX     SBAREA
        INX     X:=ADDRESS OF SEM QUEUE
        JSR     DELETE  POP TOP BLOCK FROM SEM QUEUE
        LDX     #READY
        JSR     INSERT  AND INSERT IT IN READY
        LDAA    PBCRNT
```

```
        LDX     #READY
        JSR     INSERT  INSERT CALLER IN READY
        JMP     DISPAT  AND DISPATCH
*
*
*
*-----------------------------------
* START
* SYSTEM CALL.
* INPUTS: A = 2
*       B = PRIORITY TO BE ASSIGNED TO NEW PROCESS
*       X = STACK POINTER FOR NEW PROCESS
* OUTPUTS: CARRY = 0 => SUCCESSFUL
*       CARRY = 1 => NO FREE PB'S AVAILABLE
*       IF SUCCESSFUL, A = NEW PROCESS'S ID#
* ALL ELSE PRESERVED
*
XSTART  LDAA    PBCRNT
        STAA    PBFREE
        LDX     PBAREA  X:=ADDRESS OF CURRENT PB
        STS     SP,X    SAVE CALLER'S STACK
        LDAA    #32     THERE ARE 32 PB'S
        CLRB
SRCHLP  STAB    PBFREE
        LDX     PBAREA  X:=ADDRESS OF NEXT BLOCK
        LDAB    PRIO,X  GET BLOCK'S PRIORITY
        BEQ     STRTSU  IF ZERO THEN THIS ONE IS FREE
        DECA    ANY MORE BLOCKS TO TRY?
        BEQ     STRTFA  NO=>FAILURE
        LDAB    PBFREE  YES--ADVANCE TO NEXT BLOCK
        ADDB    #8
        BRA     SRCHLP
STRTFA  PULA    FAILURE--SET CALLER'S CARRY
        ORAA    #1
        PSHA
        RTI
STRTSU  LDAB    PBFREE
        LSRB    COMPUTE ID#
        LSRB
        LSRB
        PULA    SUCCESS--CLEAR CALLER'S CARRY
        ANDA    #$FE
        PSHA
        TSX
        STAB    2,X     RETURN PROC. ID# IN CALLER'S A
        LDAA    1,X     GET PRIORITY FROM CALLER'S B
        LDX     PBAREA
        STAA    PRIO,X  STORE IN PROC. BLOCK
        TSX
        LDAA    3,X     A:=SP (HIGH)
        LDAB    4,X     B:=SP (LOW)
        LDX     PBAREA
        STAA    SP,X    STORE SP IN PB
```

```
        STAB    SP+1,X
        LDX     #READY
        LDAA    PBFREE
        LDAB    PBCRNT
        JSR     INSERT  ENQUEUE NEW PROCESS IN READY
        TBA
        LDX     #READY
        JSR     INSERT  ENQUEUE CALLER IN READY
        JMP     DISPAT  AND GO
*
*
*
*-----------------------------------
* SLEEP
* SYSTEM CALL.
* INPUTS: A = 3
*       B = TIME EXPONENT (LOWER 4 BITS)
*       X = TIME INTERVAL COUNT, X>1
* OUTPUTS: NONE
* ALL REGISTERS PRESERVED
* FUNCTION: THE CALLER IS PUT TO SLEEP FOR (TIME COUNT
*       SHIFTED LEFT EXPONENT TIMES) CLOCK TICKS.
*
XSLEEP  LDAA    PBCRNT
        STAA    PBFREE
        LDX     PBAREA  X:=ADDRESS OF CURRENT PB
        STS     SP,X    SAVE CALLER'S STACK
        TSX
        LDX     3,X
        STX     TMP2    STORE INTERVAL IN LOW ORDER WORK SPACE
        CLR     TMP1    CLEAR HIGH ORDER WORK SPACE
        CLR     TMPX
        ANDB    #$F     MASK SHIFT COUNT TO 4 BITS
        BEQ     SLP7    UNTIL SHIFT COUNT=0
SLP2    ASL     TMP3    DO
        ROL     TMP2    SHIFT TIME COUNT LEFT
        ROL     TMP1
        ROL     TMPX
        DECB
        BNE     SLP2    OD
* SLEEP PERIOD MUST BE >= 2
SLP7    TST     TMPX
        BNE     SLP3
        TST     TMP1
        BNE     SLP3
        TST     TMP2
        BNE     SLP3
        LDAA    TMP3
        CMPA    #2
        BCC     SLP3
        RTI
* UPDATE SYSTEM TIME
* SYSTEM TIME:=SYSTEM TIME+(LASTDT-TIMER COUNT)
```

```
SLP3    LDAA    TIMERA+2
        LDAB    TIMERA+3
        PSHB
        PSHA
        LDAA    LASTDT
        LDAB    LASTDT+1
        TSX
        SUBB    1,X
        SBCA    0,X
        ADDB    SYSTIM+3
        ADCA    SYSTIM+2
        STAB    SYSTIM+3
        STAA    SYSTIM+2
        BCC     SLP4
        INC     SYSTIM+1
        BNE     SLP4
        INC     SYSTIM
SLP4    PULA
        PULB
* ADD COUNT TO CURRENT TIME TO GET WAKEUP TIME
* AND COPY RESULT INTO PROCESS'S PB
        LDX     PBAREA  X:=PB ADDRESS
        LDAA    TMP3
        ADDA    SYSTIM+3
        STAA    7,X
        LDAA    TMP2
        ADCA    SYSTIM+2
        STAA    6,X
        LDAA    TMP1
        ADCA    SYSTIM+1
        STAA    5,X
        LDAA    TMPX
        ADCA    SYSTIM
        STAA    4,X
* NOW INSERT PB INTO TIME QUEUE IN WAKEUP TIME ORDER
        LDX     #TIMER  X:=ADDRESS OF QUEUE ROOT
SLPNXT  STX     TMPX    TMPX,1 POINTS TO QUEUE ROOT
        LDAA    0,X     IF QUEUE IS EMPTY
        INCA
        BEQ     SLPGO   THEN DO INSERTION HERE
        DECA
        STAA    TMP3
        LDAA    PBAREA
        STAA    TMP2    TMP2,3 POINTS TO TOP QUEUE ELEMENT
        LDX     PBAREA
        LDAA    4,X
        LDAB    5,X
        LDX     TMP2
        CMPA    4,X
        BCS     SLPGO
        BHI     SLP1
        CMPB    5,X
        BCS     SLPGO
```

```
        BHI     SLP1
        LDX     PBAREA
        LDAA    6,X
        LDAB    7,X
        LDX     TMP2
        CMPA    6,X
        BCS     SLPGO
        BHI     SLP1
        CMPB    7,X
        BCS     SLPGO
* WAKEUP TIME OF INPUT PB >= THAT OF TOP QUEUE
* ELEMENT, ADVANCE TO NEXT QUEUE ELEMENT
SLP1    LDX     TMP2    TMP2,3 STILL POINTS TO TOP
        INX     QUEUE BLOCK
        BRA     SLPNXT
* INSERT PB IN QUEUE
SLPGO   LDX     TMPX    X:=QUEUE ROOT ADDR
        LDAA    PBCRNT  INSERT CURRENT PROCESS
        LDAB    0,X     POINT B TO 1ST QUEUE BLOCK
        STAA    0,X     MAKE NEW BLOCK 1ST IN QUEUE
        STAA    PBFREE
        LDX     PBAREA
        STAB    LINK,X  POINT NEW PB'S LINK FIELD TO REST OF QUEUE
* LASTDT:=MIN(WAKEUP TIME OF FIRST BLOCK-SYSTEM TIME,$FFFF)
        LDAA    TIMER
        STAA    PBFREE
        LDX     PBAREA
        LDAA    6,X
        LDAB    7,X
        SUBB    SYSTIM+3
        SBCA    SYSTIM+2
        STAA    LASTDT
        STAB    LASTDT+1
        LDAA    4,X
        LDAB    5,X
        SBCB    SYSTIM+1
        SBCA    SYSTIM
*
        TSTA    IF REMAINING TIME >
        BNE     SLP5    MAX INTERVAL
        TSTB    THEN SET LASTDT
        BEQ     SLP6    FOR MAX INTERVAL
SLP5    LDX     #$FFFF
        STX     LASTDT
* TIMER LATCHES := LASTDT
SLP6    LDX     LASTDT
        STX     TIMERA+2 START TIMER
*
        JMP     DISPAT
*
*
*
*-------------------------------
```

```
* WHO
* SYSTEM CALL.
* INPUTS: A = 4
* OUTPUTS: B = CURRENT PROCESS'S (IE. CALLER'S) ID#
* ALL REGISTERS PRESERVED (EXCEPT B)
*
XWHO    LDAB    PBCRNT
        LSRB
        LSRB
        LSRB
        LSRB
        TSX
        STAB    1,X     STORE ID# IN CALLER'S B
        RTI
*
*
*
*-------------------------------------
* KILL
* SYSTEM CALL.
* INPUTS: A = 5
*       B = ID# OF PROCESS TO KILL
* OUTPUTS: NONE
* ALL REGISTERS PRESERVED (AN IRRELEVENT PROPERTY, IF THIS
* HAPPENS TO BE SUICIDE)
*
XKILL   LDAA    PBCRNT
        STAA    PBFREE
        LDX     PBAREA
        STS     SP,X    SAVE CALLER'S STACK
        LDX     #READY  ENQUEUE CALLER
        JSR     INSERT
* ZERO PRIORITY OF PROCESS TO BE DELETED
        ASLB    CONVERT ID# TO POINTER
        ASLB
        ASLB
        STAB    PBFREE
        LDX     PBAREA
        CLR     PRIO,X
* REMOVE PRIORITY 0 PROCESS FROM READY QUEUE
        LDX     #READY
        BSR     KILLS
        BCC     KILL5
* REMOVE PRIORITY 0 PROCESS FROM TIMER QUEUE
        LDX     #TIMER
        BSR     KILLS
        BCC     KILL5
* DELETE PRIORITY 0 PROCESS FROM SEMAPHORE QUEUES
        LDAB    #128    LOOP COUNT
        LDX     #SBS    SEM AREA START
KILL3   LDAA    1,X     GET SEM.QUEUE
        INX     ADVANCE PONTER TO SEM.QUEUE
        INCA    IS QUEUE NULL?
        BEQ     KILL4   IF YES THEN DON'T BOTHER WITH KILLS
```

```
        STX     TMP4    SAVE REGISTERS
        PSHB
        BSR     KILLS
        PULB
        BCC     KILL5
        LDX     TMP4
KILL4   INX
        DECB
        BNE     KILL3
KILL5   JMP     DISPAT
*
*KILLS - ROUTINE LOCAL TO KILL
* INPUTS: X = POINTER TO QUEUE ROOT
* DELETES A 0-PRIORITY PB FROM THE GIVEN QUEUE
KILLS   LDAA    0,X     GET ROOT
        INCA    IF IT IS NULL
        BEQ     KILL1   THEN WE ARE DONE
        DECA
        STAA    PBFREE  ELSE
        STX     TMP2    SAVE ADDRESS OF ROOT
        LDX     PBAREA
        LDAB    PRIO,X  GET BLOCK'S PRIORITY
        BNE     KILL2
        LDX     TMP2    RECOVER ADDRESS OF ROOT
        JSR     DELETE  DELETE BLOCK
        CLC
        RTS
KILL2   INX     ADVANCE TO NEXT BLOCK IN QUEUE
        BRA     KILLS   AND RECURSE
KILL1   SEC
        RTS
*
*
*-------------------------------------
* IRQ HANDLER
* THIS IS THE CODE TO WHICH IRQ'S SHOULD BE VECTORED.
* ITS SOLE FUNCTION IS TO WAKEUP THE INTERRUPT POLLER
* PROCESS. IF THERE ARE VERY URGENT DEVICES ON THE
* INTERRUPT BUS FOR WHICH THE SYSTEM RESPONSE WOULD BE
* TOO SLOW, THEN THE SYSTEM CAN BE SHORT-CIRCUITED AT
* THIS POINT BY EXECUTING SOME DEVICE HANDLER CODE RIGHT
* HERE -- BEFORE TRANSFERRING CONTROL BACK TO THE SYSTEM
* SOFTWARE. THIS SHOULD, HOWEVER, BE DONE WITH GREAT CARE
* AND FORETHOUGHT IF IT IS TO MESH SMOOTHLY WITHOUT CAUSING
* ERRORS, OR A CRASH.
IRQINT  LDAB    #0      INTERRUPT POLLER'S SEMAPHORE#
        JMP     XV      THE REST IS IDENTICAL TO A NORMAL
*        V OPERATION
*
*
*---------------
```

```
* FILE: PROCESSES
****************************************
*-----SYSTEM PROCESSES
****************************************
*
*
        ORG     $C00
*
*--------------------------------------
* IDLE PROCESS CODE
*
IDLE    EQU     *
        ORG     XIDLE
        FDB     IDLE
        ORG     IDLE
*
        WAI
        BRA     IDLE
*
*
*
*--------------------------------------
* INITIALIZATION PROCESS CODE
*
INITGO  EQU     *
        ORG     XINITG
        FDB     INITGO
        ORG     INITGO
*
* START UP IDLE PROCESS AT LOWEST PRIORITY
        LDX     XIDLE
        STX     IDLESP+6
        LDX     #IDLESP
        LDAB    XIDLE+2  PRIORITY
        LDAA    #2
        SWI     SYSTEM(START)
*
* START INTERRUPT POLLER PROCESS
        LDX     XINTPO
        STX     INTPOS+6
        LDX     #INTPOS INTERRUPT POLLER'S STACK
        LDAB    XINTPO+2        PRIORITY
        LDAA    #2
        SWI     SYSTEM(START)
* START CLOCK PROCESS
        LDX     XCLOCK
        STX     CLOCKS+6
        LDX     #CLOCKS CLOCK PROCESS'S STACK
        LDAB    XCLOCK+2        PRIORITY
        LDAA    #2
        SWI     SYSTEM(START)
*
* START UP ACIA RECEIVER HANDLER PROCESS
```

```
        LDX     #SMRCVE*2+SBS  INITIALIZE SEM TO 1
        INC     0,X
        LDX     XRCVR
        STX     RCVRSP+6
        LDX     #RCVRSP
        LDAB    XRCVR+2  PRIORITY
        LDAA    #2
        SWI     SYSTEM(START)
*
* START UP ACIA XMITTER HANDLER PROCESS
        LDX     #SMXMTE*2+SBS  INIT SEM TO 1
        INC     0,X
        LDX     #SMXMTR*2+SBS
        INC     0,X
        LDX     XXMTR
        STX     XMTRSP+6
        LDX     #XMTRSP STACK
        LDAB    XXMTR+2  PRIORITY
        LDAA    #2
        SWI     SYSTEM(START)
*
* START UP 'EXAMIN' PROCESS
        LDX     XEXAMI
        STX     EXAMSP+6
        LDX     #EXAMSP
        LDAB    XEXAMI+2         PRIORITY
        LDAA    #2       START FUNCTION CODE
        SWI     SYSTEM(START)
*
* START UP LEVER INTERRUPT PROCESS
        LDX     XLVR1
        STX     LVR1S+6
        LDX     #LVR1S
        LDAB    XLVR1+2  PRIORITY
        LDAA    #2       START
        SWI
*
* START UP COMMAND PROCESS
        LDX     XCMND
        STX     CMNDSP+6
        LDX     #CMNDSP
        LDAB    XCMND+2  PRIORITY
        LDAA    #2       START
        SWI
*
* INITIALIZE PHYSICAL DEVICES/INTERFACES
*
* CLEAR ALL 6840 CONTROL REGISTER SAVE LOCATIONS
        LDAA    #3
        LDX     #CR1A
CLRTIM  CLR     0,X
        INX
        DECA
```

```
        BNE     CLRTIM
* INITIALIZE CLOCK TIMER 1A
        LDAA    #$E0    16-BIT, SINGLE SHOT,
*        OUT AND INT ENABLED, EXT CLOCK
        LDAB    #$01
        STAB    CR2A    SET FOR CR1 ACCESS
        STAB    TIMERA+1
        STAA    CR1A
        STAA    TIMERA
        LDX     #$FFFF START WITH MAX INTERVAL
        STX     LASTDT
        STX     TIMERA+2
*
* INITIALIZE ACIA
        LDAA    #3      MASTER RESET CODE
        STAA    ACIAS
        LDAA    #$91    INT ON RCV ENABLED, RTS=0
        STAA    ACIAS   8 BITS, NO PARITY, 2 STOPS,
        STAA    ACIASV  300 BAUD
*
* INITIALIZE PIA0
*CA2, CB2 ARE OUTPUTS
*CA1, CB1 ARE INPUTS CAUSING INTERRUPTS
* ON UP TRANSITIONS.
*ALL OUTPUTS WILL BE CONSIDERED ACTIVE
*WHEN HIGH (EXCEPT THE CAGE LIGHT).
        LDAB    #$FF
        LDAA    #%00111011
        STAA    PIA0+1
        STAA    PIA0RA
        STAB    PIA0    SET FOR OUTPUTS
        LDAA    PIA0RA
        ORAA    #%100
        STAA    PIA0+1
        STAA    PIA0RA
        LDAA    #%00110011
        STAA    PIA0+3
        STAA    PIA0RB
        STAB    PIA0+2
        LDAA    PIA0RB
        ORAA    #%100
        STAA    PIA0+3
        STAA    PIA0RB
        LDAB    #0
        STAB    PIA0    SET ALL OUTPUTS
        STAB    PIA0+2  LOW
*
* INITIALIZE DAC REGISTERS
        LDX     #0
        STX     DAC1
        STX     DAC1R
        STX     DAC2
        STX     DAC2R
```

```
*
*
*
* DIE -- INITIALIZATION PROCESS IS NO LONGER NEEDED
        LDAA    #4
        SWI     SYSTEM(WHO) WHO AM I?
        LDAA    #5
        SWI     SYSTEM(KILL) COMMIT SUICIDE
*
        JMP     INITGO
*
*
*
*-------------------------------------
* INTERRUPT POLLER PROCESS CODE
*
INTPOL  EQU     *
        ORG     XINTPO
        FDB     INTPOL
        ORG     INTPOL
*
*
* P(IRQ)
        CLRB    SEM #0
        CLRA
        SWI     SYSTEM(P)
*
* TEST 6840 #A
        LDAA    TIMERA+1
        BPL     INTPL1  BRANCH IF NO INTERRUPT
        ANDA    #1      MASK FOR TIMER 1 FLAG
        BEQ     INTPL1  BRANCH IF NO INTERRUPT
        LDX     TIMERA+2 READ TIMER COUNT TO CLEAR FLAG
*
* V(CLOCK)      SEMAPHORE #1
        LDAA    #1
        TAB      A=1, B=1
        SWI      SYSTEM(V)
*
* ACIA INTERRUPT CHECK LOGIC
* PSEUDO-CODE
* IF ACIA RCVR INTERRUPT
* THEN READ CHAR INTO VIRTUAL DEVICE REGISTER
* SIGNAL RECEIVER PROCESS
* ELSE IF ACIA XMITTER FINISHED
* THEN DISABLE INTERRUPT FROM XMITTER
*       SIGNAL TRANSMITTER PROCESS
* FI
INTPL1  LDAA    ACIAS
        BPL     INTPL3  BRANCH IF NO INTERRUPT
        LSRA    CHECK RCVR STATUS
        BCC     INTPL2  CLEAR - TRY XMITTER
        LDAA    ACIAD   READ DATA CHAR
```

```
          STAA   ACIAVD  STORE IN VIRT. REGISTER
          LDAB   #SMRCVR
          LDAA   #1
          SWI    SIGNAL RECEIVER PROCESS
          BRA    INTPL3
INTPL2    LSRA   CHECK XMITTER STATUS
          BCC    INTPL3  CLEAR - DONE
          LDAA   ACIASV  DISABLE INT FROM XMITTER
          ANDA   #$9F
          STAA   ACIASV
          STAA   ACIAS
          LDAB   #SMXMTR
          LDAA   #1
          SWI    SIGNAL XMTR PROCESS
*
INTPL3    LDAA   PIA0+1  GET INTERRUPT FLAGS FOR PIA0
          ASLA
          BCC    INTPL5  NO INT. FROM CA1
          LDAA   PIA0RA  SET FOR PERIPHERAL REG A
          ORAA   #4
          STAA   PIA0RA
          STAA   PIA0+1
          LDAA   PIA0    CLEAR INTERRUPT FLAG
          LDAB   #SMPIA1 SIGNAL FOR CA1 INTERRUPT
          LDAA   #1
          SWI
*
* CHECK FOR PIA0, CB1 INTERRUPT
INTPL5    LDAA   PIA0+3
          ASLA
          BCC    INTPL6
*
* SET FOR REG B ACCESS
          LDAA   PIA0RB
          ORAA   #4
          STAA   PIA0RB
          STAA   PIA0+3
* READ REG B TO CLEAR INT.
          LDAA   PIA0+2
* SIGNAL CB1 INT.
          LDAB   #SMPIA2
          LDAA   #1
          SWI    V(SMPIA2)
*
INTPL6    JMP    INTPOL
*
*
*
*-------------------------------------
* CLOCK PROCESS CODE
*
CLOCK     EQU    *
          ORG    XCLOCK
```

```
        FDB     CLOCK
        ORG     CLOCK
*
        LDAB    #1      CLOCK PROCESS'S SEMAPHORE
        CLRA
        SWI     SYSTEM(P) WAIT FOR CLOCK INT. SIGNAL
        SEI     BEGIN CRITICAL SECTION
* UPDATE SYSTEM TIME
        LDAA    LASTDT+1
        ADDA    SYSTIM+3
        STAA    SYSTIM+3
        LDAA    LASTDT
        ADCA    SYSTIM+2
        STAA    SYSTIM+2
        BCC     CLCK1
        INC     SYSTIM+1
        BNE     CLCK1
        INC     SYSTIM
CLCK1   LDAA    TIMER   GET QUEUE ROOT
        INCA
        BEQ     CLCK4   IF QUEUE EMPTY THEN WE ARE DONE
        DECA
        STAA    PBFREE
        LDX     PBAREA
        LDAA    SYSTIM  COMPARE SYSTEM TIME TO BLOCK'S
        CMPA    4,X     WAKEUP TIME
        BHI     CLCK3
        BCS     CLCK4
        LDAA    SYSTIM+1
        CMPA    5,X
        BHI     CLCK3
        BCS     CLCK4
        LDAA    SYSTIM+2
        CMPA    6,X
        BHI     CLCK3
        BCS     CLCK4
        LDAA    SYSTIM+3
        CMPA    7,X
        BCC     CLCK3
* EITHER WAKEUP TIME HAS NOT ARRIVED YET,
* OR TIMER QUEUE IS EMPTY
CLCK4   LDAA    TIMER   IS QUEUE EMPTY?
        INCA
        BNE     CLCK5   BRANCH IF NOT
CLCK7   LDX     #$FFFF  YES-SET FOR MAX INTERVAL
        STX     LASTDT
        BRA     CLCK6
* QUEUE NOT EMPTY - COMPUTE TIME REMAINING
* TO NEXT WAKEUP TIME
* (= MIN(WAKEUP TIME-SYSTEM TIME,$FFFF))
CLCK5   LDAA    TIMER
        STAA    PBFREE
        LDX     PBAREA
```

```
        LDAA    6,X
        LDAB    7,X
        SUBB    SYSTIM+3
        SBCA    SYSTIM+2
        STAA    LASTDT
        STAB    LASTDT+1
        LDAA    4,X
        LDAB    5,X
        SBCB    SYSTIM+1
        SBCA    SYSTIM
*
        TSTA
        BNE     CLCK7
        TSTB
        BNE     CLCK7
* START TIMER
CLCK6   LDX     LASTDT
        STX     TIMERA+2
*
        CLI
        BRA     CLOCK
* SYSTEM TIME >= WAKEUP TIME, SO PROCESS MUST
* BE SCHEDULED TO RUN AGAIN
CLCK3   LDX     #TIMER  DELETE BLOCK FROM TIMER QUEUE
        JSR     DELETE
        LDX     #READY
        JSR     INSERT  AND INSERT IT INTO READY QUEUE
        BRA     CLCK1
*
*
*
**********************************
* ACIA TRANSMITTER HANDLER PROCESS
*
XMTR    EQU     *
        ORG     XXMTR
        FDB     XMTR
        ORG     XMTR
*
        CLRA
        LDAB    #SMXMTF
        SWI     WAIT OUTPUT BFR FULL
*
        LDAB    #SMXMTR
        CLRA
        SWI     WAIT INTERRUPT SIGNAL
*
        LDAA    XMTBF
        LDAB    ACIASV
        ANDB    #$9F
        ORAB    #$20
        SEI
        STAA    ACIAD
```

```
        STAB    ACIASV
        STAB    ACIAS
        CLI
*
        LDAA    #1
        LDAB    #SMXMTE
        SWI     SIGNAL OUTPUT BFR EMPTY
*
        BRA     XMTR
*
*
*
************************************
* ACIA RECEIVER HANLDER PROCESS
*
RCVR    EQU     *
        ORG     XRCVR
        FDB     RCVR
        ORG     RCVR
*
        LDAB    #SMRCVR
        CLRA
        SWI     P(SMRCVR) WAIT INTERRUPT
*
        LDAB    #SMRCVE
        CLRA
        SWI     P(SMRCVE) WAIT INPUT BFR EMPTY
*
        LDAB    ACIAVD  COPY CHAR INTO INPUT BFR
        STAB    RCVBF
*
        LDAB    #SMRCVF
        LDAA    #1
        SWI     V(SMRCVF) SIGNAL INPUT BFR FULL
*
        BRA     RCVR
*
*
*
************************************
* CHARACTER INPUT SUBROUTINE
*
XGETCH  EQU     *
        ORG     GETCH
        JMP     XGETCH
        ORG     XGETCH
*
        PSHB
        CLRA
        LDAB    #SMRCVF
        SWI     P(RCVBF FULL)
        LDAA    RCVBF   READ BUFFER
        PSHA
```

```
        LDAA    #1
        LDAB    #SMRCVE
        SWI     V(RCVBF EMPTY)
        PULA    GET BACK CHAR
        PULB    RESTORE B
        RTS
*
*-----------------------------------
*
* CHARACTER OUTPUT SUBROUTINE
*
XPUTCH  EQU     *
        ORG     PUTCH
        JMP     XPUTCH
        ORG     XPUTCH
*
        PSHB
        PSHA
        CLRA
        LDAB    #SMXMTE
        SWI     P(XMTBF EMPTY)
        PULA    GET CHAR
        PSHA
        STAA    XMTBF   COPY INTO OUTPUT BFR
        LDAA    #1
        LDAB    #SMXMTF
        SWI     V(XMTBF FULL)
        PULA    RESTORE REGS
        PULB
        RTS
*
*-----------------------------------
* TEST PROCESS.
* THIS IS SORT OF A MINI-MONITOR WHICH
* ALLOWS EXAMINING AND CHANGING MEMORY.
* COMMANDS -
* NNNN <COLON> --DISPLAYS 8 BYTES BEGINNING AT NNNN
* NNNN = NN, NN, NN, .... LOADS SUCCSSIVE BYTES
*       BEGINNING AT NNNN
* NNNN=PPS --START UP THE CODE BEGINNING AT NNNN AS
*       A PROCESS WITH PRIORITY PP. IF NOT
*       SUCCESSFUL, A BEEP AND A "?" WILL BE
*       USED TO INDICATE FAILURE.
* K --IF A PROCESS WAS SUCCESSFULLY STARTED WITH THE
* "S" COMMAND, THIS WILL KILL THAT PROCESS.
* NNV   - PERFORM A "V" (SIGNAL) OPERATION
*       ON SEMAPHORE NN (0 <= NN <= $7F)
* Q     - QUIT - THE MONITOR PROCESS IS NO LONGER NEEDED.
*
*
EXAMIN  EQU     *
        ORG     XEXAMI
        FDB     EXAMIN
```

```
        ORG     EXAMIN
*
        LDAA    #'#
        JSR     OUTCH
XAMR    JSR     INCH
        CMPA    #'=
        BNE     XAM1
        LDX     N
        STX     ADDR
        JMP     XAMR
XAM1    CMPA    #',
        BNE     XAM2
        LDX     ADDR
        LDAA    N+1
        STAA    0,X
        INX
        STX     ADDR
        JMP     XAMR
XAM2    CMPA    #$0D CR
        BNE     XAM3
XAM5    LDAA    #$0A
        JSR     OUTCH
        JMP     EXAMIN
XAM3    CMPA    #$3A    COLON
        BNE     XAM6
        LDAB    #8
        LDX     N
XAM4    LDAA    0,X
        JSR     PUTBYT
        INX
        DECB
        BNE     XAM4
        STX     N
XAM41   LDAA    #$0D
        JSR     OUTCH
        JMP     XAM5
XAM6    CMPA    #'V
        BNE     XAM9
        LDAB    N+1
        LDAA    #1
        SWI
        JMP     XAM41
XAM9    CMPA    #'S     START?
        BNE     XAM10   NO
        LDAA    MYSON   SON ALREADY EXISTS?
        INCA
        BNE     XAM91   YES->REFUSE TO CREATE ANOTHER
        LDX     ADDR    INITIAL PC VALUE
        STX     EXMSON+6        COPY IT INTO STACK
        LDX     #EXMSON STACK POINTER
        LDAB    N+1     PRIORITY
        LDAA    #2      FUNCTION CODE FOR 'START'
        SWI
```

```
        BCS     XAM91   GO HANDLE FAILURE
        STAA    MYSON   SUCCESS -- SAVE ID#
        JMP     XAM41   AND CONTINUE
XAM91   LDAA    #7      BEEP
        JSR     OUTCH
        LDAA    #'?
        JSR     OUTCH
        JMP     XAM41
XAM10   CMPA    #'K     KILL?
        BNE     XAM61   NO
        LDAA    #5      FUNCTION CODE FOR 'KILL'
        LDAB    MYSON   GET ID# TO KILL
        INCB
        BEQ     XAM41 IF NULL THEN DO NOTHING
        DECB
        SWI     ELSE KILL IT
        LDAA    #$FF    MYSON:=NULL
        STAA    MYSON
        JMP     XAM41   AND CONTINUE
XAM61   CMPA    #'Q
        BNE     XAM11
        LDAA    #4
        SWI      WHO AM I?
        LDAA    #5
        SWI      DIE
        JMP     EXAMIN
XAM11   CMPA    #'A
        BCS     XAM7
        ADDA    #-'A+10
XAM7    ANDA    #$0F
        LDAB    #4
XAM8    ASL     N+1
        ROL     N
        DECB
        BNE     XAM8
        ORAA    N+1
        STAA    N+1
        JMP     XAMR
*
INCH    JSR     GETCH
        JSR     PUTCH
        ANDA    #$7F
        RTS
*
OUTCH   JMP     PUTCH
*
PUTBYT  PSHA
        LDAA    #$20
        JSR     OUTCH
        PULA
        PSHA
        LSRA
        LSRA
```

```
          LSRA
          LSRA
          BSR     PUTDIG
          PULA
          PSHA
          ANDA    #$0F
          BSR     PUTDIG
          PULA
          RTS
*
PUTDIG    CMPA    #10
          BCC     PUTDG1
          ADDA    #'0
          JMP     OUTCH
PUTDG1    ADDA    #-10+'A
          JMP     OUTCH
*
* VARIABLES
ADDR      RMB     2
N         RMB     2
MYSON     FCB     $FF
*
* END OF EXAMINE
*
*
```

```
* FILE: STIMULATION
****************************
*
*
        ORG     $1000
*
****************************************
*
*
* DISPLACEMENTS FOR FIELD DEFINITIONS
* OF SPB FIELDS
*
PRWDDT EQU      0
PRCTDT EQU      PRWDDT+2
PRCCDT EQU      PRCTDT+3
ACRNT   EQU     PRCCDT+3
BCRNT   EQU     ACRNT+3
STEER   EQU     BCRNT+3
ASELCT EQU      STEER+1
BSELCT EQU      ASELCT+1
GSELCT EQU      BSELCT+1
PRNUM   EQU     GSELCT+1
TRNUM   EQU     PRNUM+2
TRDTTR EQU      TRNUM+2
FIXDT   EQU     TRDTTR+2
LIGHT   EQU     FIXDT+2
DURADT EQU      LIGHT+1
*
*SPECIAL CHARACTERS FOR
*PACKET PROTOCOL
*
* READY-TO-SEND CHAR
RTSCH   EQU     $02
* READY-TO-RECEIVE CHAR
RTRCH   EQU     $05
* NEGATIVE ACKNOWLEDGE
NACK    EQU     $15
* POSITIVE ACKNOWLEDGE CHAR
ACK     EQU     $06
*
*
*
* VARIABLES USED IN GENERATING
* STIMULATION
*
LVR1CN  RMB     2       LEVER 1 COUNT
*
* STIMULATION COUNT
STMCNT RMB      2
*
* MASK SPECIFYING IF LEVER IS TO BE
* COUNTED AND/OR TO TRIGGER
* STIMULATION
```

```
LVR1M   RMB     1
*
* POINTER TO CURRENT SPB
CURSPB RMB      2
*
*
*
* INPUT BUFFERS FOR PARAMETER BLOCKS
*
* PRIMER PARAMETER BLOCK
INPRM   RMB     32
* STIMULATION PARAMETER BLOCK
INSTM   RMB     32
*
*
*
**********************************
* IHLGET
* SUBROUTINE TO INPUT A CHARACTER
* RECEIVED PACKET
* NO REGISTERS PRESERVED.
* CHAR RETURNED IN A.
*
IHLGET  TST     NOB
* BUFFER MUST BE REFILLED
        BEQ     IHLG1
*
* GET NEXT CHAR FROM BUFFER
        LDX     NXTCH
        LDAA    0,X
        INX     ADVANCE POINTER
        STX     NXTCH
* DECREMENT BYTE COUNT
        DEC     NOB
        RTS
*
* WAIT FOR READY-TO-SEND
IHLG1   JSR     GETCH
        CMPA    #RTSCH  CHARACTER
        BNE     IHLG1
* XMIT READY-TO-RECEIVE CHAR
        LDAA    #RTRCH
        JSR     PUTCH
*
        JSR     GETCH   WAIT FOR COLON
        CMPA    #$3A
        BNE     *-5
*
        CLR     INCHK   CLEAR CHECKSUM
* GET NO. OF DATA BYTES
        BSR     IHLINS
        CMPA    #$11
        BCS     *+4
```

```
        LDAA    #0
        TAB
        STAA    NOB
* X <- POINTER TO BUFFER
        LDX     #INBFR
* INIT POINTER TO 1ST CHAR
        STX     NXTCH
        TSTB    DONE?
        BEQ     *+10    BRANCH IF SO
* IHLINS GET NEXT DATA BYTE
        BSR     IHLINS
        STAA    0,X
        INX
        DECB    DONE?
        BNE     *-6     BRANCH IF NOT
        BSR     IHLINS  GET CHECKSUM
        TST     INCHK   DID IT WORK OUT?
        BEQ     IHLG2   BRANCH IF YES
* ERROR- CLEAR BYTE COUNT
        CLR     NOB
* XMIT A NACK
        LDAA    #NACK
        BRA     *+4
* NO ERROR- XMIT ACK
IHLG2   LDAA    #ACK
        JSR     PUTCH
        JMP     IHLGET
*
*
*
* LOCAL ROUTINE- GET CHAR ADD TO INCHK
IHLINS  JSR     GETCH
        PSHA
        ADDA    INCHK
        STAA    INCHK
        PULA
        RTS
*
*
*
* VARIABLES FOR IHLGET
NOB     FCB     0 BYTE COUNT
INBFR   RMB     16 DATA
INCHK   RMB     1 CHECKSUM
NXTCH   RMB     2 PNTR TO NEXT CHAR
*
*
*
*************************************
* IHLPUT
* INTEL HEX CHARACTER OUTPUT ROUTINE.
* NO REGISTERS PRESERVED.
* CHAR TO OUTPUT PASSED IN A.
```

```
*
IHLPUT  PSHA
        LDAA    PNOB      BUFFER FULL?
        CMPA    #16
        BNE     IHLP1
        BSR     FLUSH
IHLP1   LDX     PFILL
        PULA
* ADD CHAR TO BUFFER
        STAA    0,X
        INX     ADVANCE POINTER
        STX     PFILL
        INC     PNOB      INC BYTE COUNT
        RTS
*
*
*-------------------------------
* FLUSH
* ROUTINE TO FLUSH PACKET OUTPUT
* BUFFER. NO PARAMETERS, REGISTERS
* NOT PRESERVED.
*
* XMIT REQUEST-TO-SEND CHAR
FLUSH   LDAA    #RTSCH
        JSR     PUTCH
*
* WAIT FOR READY-TO-RECEIVE CHAR
        JSR     GETCH
        CMPA    #RTRCH
        BNE     *-5
*
        LDAA    #':
        JSR     PUTCH
        LDX     #PNOB
        LDAA    PNOB      NO OF BYTES
        INCA
        STAA    PFILL
        CLR     OUTCHK    CHECKSUM
*
FLSH1   LDAA    0,X
        INX
        TAB
        ADDB    OUTCHK
        STAB    OUTCHK
        JSR     PUTCH
        DEC     PFILL
        BNE     FLSH1
*
        CLRA
        SUBA    OUTCHK
        JSR     PUTCH
*
FLSH2   JSR     GETCH
```

```
        CMPA    #NACK
        BEQ     FLUSH
        CMPA    #ACK
        BNE     FLSH2
* ENTRY POINT TO INITIALIZE POINTERS
FLSHIN CLR      PNOB
        LDX     #OUTBFR
        STX     PFILL
*
        RTS
*
*
*-------------------------
* VARIABLES FOR IHLPUT AND FLUSH
PNOB    FCB     0
OUTBFR  RMB     16
OUTCHK  RMB     1
PFILL   RMB     2
*
*
************************************
*
* COMMAND INTERPRETTER MAINLINE
*
* THIS IS THE EXECUTIVE PROCESS FOR
* THE RAT STIMULATION EXPERIMENTS
*
*
CMND    EQU     *
        ORG     XCMND
        FDB     CMND
        ORG     CMND
*
        JSR     FLSHINT
* INITIALIZE SMPCKO SEMAPHORE
        LDAB    #SMPCKO
        LDAA    #1
        SWI
*
CMND1   JSR     IHLGET  GET NEXT CHAR
*
        LDX     #NULL
        CMPA    #0      NULL COMMAND?
        BEQ     CMNDGO  YES
*
        LDX     #ACCEPT
* ACCEPT NEW SPBS ?
        CMPA    #1
        BEQ     CMNDGO
*
        LDX     #RPTSPB
* ECHO PARAMETER BLOCKS?
        CMPA    #2
```

```
        BEQ     CMNDGO
*
        LDX     #RDATA
* REPORT TRIAL DATA ?
        CMPA    #3
        BEQ     CMNDGO
*
        LDX     #BEGIN
* BEGIN TRIAL ?
        CMPA    #4
        BEQ     CMNDGO
*
        LDX     #CANCEL
* CANCEL TRIAL?
        CMPA    #5
        BEQ     CMNDGO
*
        BRA     CMND1
*
CMNDGO  JSR     0,X
        BRA     CMND1
*
*
************************************
* RDATA
* SUBROUTINE OF CMND TO REPORT DATA
* FROM THE LAST TRIAL.
*
RDATA   EQU     *
* P(SMPCKO) WAIT FOR PACKET OUTPUT SYS
        CLRA
        LDAB    #SMPCKO
        SWI
* RESPOND WITH 'REPORT' COMMAND ACK.
        LDAA    #3
        JSR     IHLPUT
* SEND LEVER PRESS COUNT
        LDAA    LVR1CN
        JSR     IHLPUT
        LDAA    LVR1CN+1
        JSR     IHLPUT
* SEND REWARD COUNT
        LDAA    STMCNT
        JSR     IHLPUT
        LDAA    STMCNT+1
        JSR     IHLPUT
*
        JSR     FLUSH
* V(SMPCKO)
        LDAA    #1
        LDAB    #SMPCKO
        SWI
*
```

```
        RTS
*
*
*****************************************
* RPTSPB
* SUBROUTINE OF CMND TO REPEAT SPB'S
* BACK TO MASTER COMPUTER.
*
RPTSPB EQU     *
* P(SMPCKO) WAIT FOR PACKET OUTPUT SYS
        CLRA
        LDAB    #SMPCKO
        SWI
*
* RESPOND WITH 'REPEAT' CMND ACK.
        LDAA    #2
        JSR     IHLPUT
*
        LDX     #INPRM  INIT POINTER
        STX     ACPTV1
        LDAA    #64     INIT COUNTER
        STAA    ACPTV2
*
RPTS1   LDX     ACPTV1
        LDAA    0,X
        INX
        STX     ACPTV1
        JSR     IHLPUT
        DEC     ACPTV2
        BNE     RPTS1
*
        JSR     FLUSH
*
* V(SMPCKO)
        LDAA    #1
        LDAB    #SMPCKO
        SWI
*
        RTS
*
*
*****************************************
* ACCEPT- ROUTINE TO ACCEPT NEW
* SPB'S
*
ACCEPT EQU     *
*
* P(SMPCKO) WAIT FOR PACKET OUTPUT SYS
        CLRA
        LDAB    #SMPCKO
        SWI
*
        LDX     #INPRM  BUFFER POINTER
```

```
          STX     ACPTV1
*
          LDAB    #64     BYTE COUNT
          STAB    ACPTV2
*
ACCPT1    JSR     IHLGET  GET NEXT BYTE
          LDX     ACPTV1
          STAA    0,X     STORE BYTE
          INX     AND ADVANCE POINTER
          STX     ACPTV1
*
          DEC     ACPTV2  DEC COUNT
          BNE     ACCPT1
*
* ACKNOWLEDGE COMMAND
          LDAA    #1 COMMAND CODE
          JSR     IHLPUT
          JSR     FLUSH
*
* V(SMPCKO)
          LDAA    #1
          LDAB    #SMPCKO
          SWI
*
          RTS
*
* VARIABLES
ACPTV1    RMB     2
ACPTV2    RMB     1
*
*
*****************************************
* NULL - CODE FOR NULL COMMAND
*
NULL      EQU     *
*
* P(SMPCKO)
          CLRA
          LDAB    #SMPCKO
          SWI
*
* ACKNOWLEDGE NULL COMMAND
          JSR     IHLPUT
          JSR     FLUSH
*
* V(SMPCKO)
          LDAA    #1
          LDAB    #SMPCKO
          SWI
*
          RTS
*
*
```

```
*
************************************
* CANCEL- SUBROUTINE OF COMMAND
* INTERPRETTER TO DESTROY AN IN-
* PROGRESS TRIAL
*
CANCEL EQU     *
*
* KILL STIMULATION PROCESS
        SEI
        LDAB    STIMID
        LDAA    #$FF
        STAA    STIMID
        CMPB    #$FF
        BEQ     *+5
        LDAA    #5      SYS(KILL)
        SWI
*
* KILL TRIAL PROCESS
        SEI
        LDAB    TRIALI
        LDAA    #$FF
        STAA    TRIALI
        CMPB    #$FF
        BEQ     *+5
        LDAA    #5      SYS(KILL)
        SWI
*
* DISARM LEVER
        SEI
        CLR     LVR1M
* & TURN OFF LEVER LIGHT
        LDAA    PIA0+2
        ANDA    #$BF
        STAA    PIA0+2
        CLI
*
* ACKNOWLEDGE COMMAND
* P(SMPCKO)
        CLRA
        LDAB    #SMPCKO
        SWI
* SEND CMND CODE
        LDAA    #5
        JSR     IHLPUT
        JSR     FLUSH
* V(SMPCKO)
        LDAA    #1
        LDAB    #SMPCKO
        SWI
*
        RTS
*
```

```
*************************************
* BEGIN- SUBROUTINE OF COMMAND
* INTERPRETTER TO BEGIN AN EXPERIMENTAL
* TRIAL
*
BEGIN   EQU     *
*
* DO NOTHING IF ALREADY RUNNING
        LDAA    STIMID
        CMPA    #$FF
        BEQ     *+3
        RTS
*
* INITIALIZE SEMAPHORES
        LDX     #SMSGO*2+SBS
        CLR     0,X     SMSGO:=0
        LDX     #SMSLCK*2+SBS
        CLR     0,X     SMSLCK:=0
*
* START STIMULATION PROCESS
        LDX     XSTIM   ENTRY POINT
        STX     STIMSP+6
        LDX     #STIMSP STACK
        LDAB    XSTIM+2 PRIORITY
        LDAA    #2      SYS(START)
        SWI
* SAVE ID#
        STAA    STIMID
*
* START TRIAL PROCESS
        LDX     XTRIAL  ENTRY POINT
        STX     TRIALS+6
        LDX     #TRIALS STACK
        LDAB    XTRIAL+2 PRIORITY
        LDAA    #2      SYS(START)
        SWI
* SAVE ID#
        STAA    TRIALI
*
        RTS
*
* ID#'S FOR STIMULATION & TRIAL PROC.
STIMID FCB      $FF
TRIALI FCB      $FF
*
*************************************
* TRIAL
* SUB-PROCESS OF COMMAND PROCESS TO
* RUN EXPERIMENTAL TRIAL
*
TRIAL   EQU     *
        ORG     XTRIAL
        FDB     TRIAL
```

```
            ORG     TRIAL
*
* FLASH CAGE LIGHT FOR 1/2 SEC.
            SEI
            LDAA    PIAORA
            ORAA    #$30
            ANDA    #$F7
            STAA    PIAORA
            STAA    PIA0+1   CA2 DOWN
            CLI
*
            LDX     #50      1/2 SEC
            CLRB
            LDAA    #3
            SWI     SYS(SLEEP)
*
            SEI
            LDAA    PIAORA
            ORAA    #$38
            STAA    PIAORA
            STAA    PIA0+1   CA2 UP
            CLI
*
* MAKE PRIMING BLOCK THE CURRENT ONE
            LDX     #INPRM
            STX     CURSPB
*
* AND TRIGGER PRIMING
            LDAB    #SMSGO   V(SMSGO)
            LDAA    #1
            SWI
*
* UNLOCK STIMULATION LOCK
            LDAB    #SMSLCK  V(SMSLCK)
            LDAA    #1
            SWI
*
* WAIT FOR COMPLETION OF PRIMING
            CLRA
            SWI     P(SMSLCK)
*
* ZERO LEVER COUNTS.
            LDX     #0
            STX     LVR1CN
            STX     STMCNT
*
* CURRENT SPB:= REWARD
            LDX     #INSTM
            STX     CURSPB
*
* LEVER1 LIGHT ON (=>LEVER ARMED)
            SEI
            LDAA    PIA0+2   REG B
```

```
        ORAA    #$40
        STAA    PIA0+2
        CLI
*
* UNLOCK STIMULATION LOCK
        LDAA    #1      V(SMSLCK)
        LDAB    #SMSLCK
        SWI
*
* SLEEP FOR TRIAL DURATION
        LDX     #INPRM
        LDAB    DURADT,X
        LDX     DURADT+1,X
        LDAA    #3      SLEEP CODE
        SWI
*
* WAIT FOR COMPLETION OF STIMULATION
        LDAB    #SMSLCK
        CLRA    P(SMSLCK)
        SWI
*
* LEVER1 LIGHT OFF
        SEI
        LDAA    PIA0+2
        ANDA    #$BF
        STAA    PIA0+2
        CLI
*
* SET LEVER MASK TO 'IGNORE LEVER'
        CLR     LVR1M
*
* FINISHED TRIAL- ACKNOWLEDGE COMMAND
* TO MASTER
*
* P(SMPCKO)
        CLRA
        LDAB    #SMPCKO
        SWI
*
* SEND COMMAND #
        LDAA    #4
        JSR     IHLPUT
        JSR     FLUSH
*
* V(SMPCKO)
        LDAA    #1
        LDAB    #SMPCKO
        SWI
*
* KILL STIMULATION PROCESS
*
* READ STIM PROCESS ID#
        SEI
```

```
        LDAB    STIMID
        LDAA    #$FF
        STAA    STIMID
        CMPB    #$FF
        BEQ     *+5
* & KILL IT
        LDAA    #5
        SWI
*
* KILL SELF- TRIAL PROCESS NOT NEEDED
        SEI
        LDAB    TRIALI
        LDAA    #$FF
        STAA    TRIALI
        LDAA    #5
        SWI
*
*****************************************
* STIMULATION PROCESS
*
*
STIM    EQU     *
        ORG     XSTIM
        FDB     STIM
        ORG     STIM
*
* WAIT FOR GO SIGNAL
        LDAB    #SMSGO
        CLRA
        SWI     SYS(P)
*
* WAIT FOR STIMULATION LOCK
        LDAB    #SMSLCK
        SWI     SYS(P)
*
* TURN OFF LEVER1 LIGHT
        SEI
        LDAA    PIA0+2  REG B
        ANDA    #$BF
        STAA    PIA0+2
        CLI
*
* PUSH # TRAINS ONTO STACK
        LDX     CURSPB
        LDAA    TRNUM+1,X
        PSHA
        LDAA    TRNUM,X
        PSHA
*
* SET UP CORRECT PARAMETERS
STIM1   JSR     STMPRM
*
* PUSH # PULSE PAIRS ONTO STACK
```

```
        LDX   CURSPB
        LDAA  PRNUM+1,X
        PSHA
        LDAA  PRNUM,X
        PSHA
*
*
* PULSE PAIR LOOP
* TOTAL TIME AROUND PULSE PAIR LOOP
*       = 91 + 56*PRCCDT (CYCLES)
*
* FIRE A PULSE PAIR
        SEI
STIM2   LDAA  PIA0RB
        ANDA  #$C7
        ORAA  #$38
        STAA  PIA0+3  BRING CB2 UP
        ANDA  #$F7
        STAA  PIA0+3  AND THEN DOWN
        STAA  PIA0RB
*
* ANY MORE PULSE PAIRS?
        TSX
        LDAA  0,X
        LDAB  1,X
        SUBB  #1
        SBCA  #0
        STAA  0,X
        STAB  1,X
        ORAB  0,X
        BEQ   STIM6   BRA IF NO MORE
*
* DELAY LOOP FOR INTER-PAIR DELAY
        LDX   CURSPB
        LDAA  PRCCDT,X DLY CONST
        INCA
        STAA  CCHI
        LDX   PRCCDT+1,X
STIM3   LDAA  PIA0+1  READ CRA
        ROLA   LEVER PRESS?
        BCC   STIM4   BRANCH IF NOT
*
        LDAA  PIA0    CLEAR FLAG
        INC   LVR1CN+1 INCREMENT
        BNE   *+7     LEVER COUNT
        INC   LVR1CN
        BRA   *+7
        BRA   *+2
        BRA   *+2
        NOP
        BRA   STIM5
*
* DELAY IN THIS BRANCH IS TO BALANCE
```

```
* THAT IN THE OTHER
STIM4   LDAA    #3
        DECA
        BNE     *-1
        BRA     *+2     4 CYCLES
        BRA     *+2     4 CYCLES
*
STIM5   DEX
        BEQ     *+7
        NOP
        BRA     *+2
        BRA     STIM3
        DEC     CCHI
        BNE     STIM3
*
        BRA     STIM2
*
* POP # PAIRS OFF STACK
STIM6   PULA
        PULA
*
* SET LEVER MASK FOR COUNTING ONLY
        LDAA    #$01
        STAA    LVR1M
*
* ENABLE INTERRUPTS AGAIN
        CLI
*
* MORE PULSE TRAINS TO DELIVER?
        TSX
        LDAA    0,X
        LDAB    1,X
        SUBB    #1
        SBCA    #0
        STAA    0,X
        STAB    1,X
        ORAB    0,X
        BEQ     STIM7
*
* SLEEP FOR INTER-TRAIN DELAY
        LDX     CURSPB
        LDX     TRDTTR,X
        CLRB
        LDAA    #3
        SWI
*
* AND LOOP FOR NEXT TRAIN
        JMP     STIM1
*
* SLEEP FOR FIXED INTERVAL DELAY
STIM7   LDX     CURSPB
        LDX     FIXDT,X
        CLRB
```

```
        LDAA    #3      SYS(SLEEP)
        SWI
*
* SET LEVER MASK TO ALLOW TRIGGERING AND COUNTING
        LDAA    #3
        STAA    LVR1M
*
* SIGNAL STIMULATION LOCKOUT SEMAPHORE
        LDAB    #SMSLCK
        LDAA    #1
        SWI
*
* END OF STIMULATION
*
* TURN ON LEVER1 LIGHT
        SEI
        LDAA    PIA0+2  REG B
        ORAA    #$40
        STAA    PIA0+2
        CLI
*
* POP # TRAINS OFF STACK
        PULA
        PULA
*
* & LOOP
        JMP     STIM
*
CCHI    RMB     1
*
*--------------------
* STMPRM - SUBROUTINE OF STIM TO
*SET UP PARAMETERS FOR STIMULATION.
*
* CONFIGURE TIMERS 2A, 3A FOR PULSE
* PAIR GENERATION
STMPRM  LDAA    #$B2
        SEI
        STAA    CR2A
        STAA    TIMERA+1
        STAA    CR3A
        STAA    TIMERA
        CLI
*
* SET PULSE PAIR PRAMETERS INTO
* TIMER LATCHES
        LDX     CURSPB
        LDAA    PRWDDT,X PULSE WIDTH
        LDAB    PRWDDT+1,X
        STAA    TIMERA+4
        STAB    TIMERA+5
* PULSE SEPARATION
        LDAA    PRCTDT+1,X
```

```
        LDAB    PRCTDT+2,X
        STAA    TIMERA+6
        STAB    TIMERA+7
* SET TIMER 3 PRESCALER
        LDAA    PRCTDT,X
        ANDA    #1
        SEI
        ORAA    CR3A
        STAA    CR3A
        STAA    TIMERA
        CLI
*
* SET PIA FOR PERIPHERAL REG. ACCESS
        SEI
        LDAA    PIA0RA
        ORAA    #$04
        STAA    PIA0RA
        STAA    PIA0+1
        CLI
*
* SET CURRENT MAGNITUDES
        LDX     CURSPB SET 'A' CURRENT
        LDX     ACRNT+1,X
        SEI
        STX     DAC1R
        STX     DAC1
        CLI
*
        LDX     CURSPB SET 'B' CURRENT
        LDX     BCRNT+1,X
        SEI
        STX     DAC2R
        STX     DAC2
        CLI
*
* CONSTRUCT REGISTER A CONTENTS
        LDX     CURSPB
        LDAA    ASELCT,X
        ANDA    #7
        ASLA
        ASLA
*
        LDAB    BSELCT,X
        RORB
        RORB
        RORB
        RORB
        ANDB    #$E0
        ABA
*
        LDAB    BCRNT,X
        ASLB
        ANDB    #2
```

```
        ABA
*
        LDAB    ACRNT,X
        ANDB    #1
        ABA
*
* COPY VALUE INTO PIA REGISTER
        STAA    PIA0
*
* SET GROUND ELECTRODE & PULSE STEERING
* BITS IN REG. B
        LDAA    GSELCT,X
        LDAB    STEER,X
        ASLB
        ASLB
        ASLB
        ABA
        ANDA    #$1F
        SEI
        ORAB    PIA0+2
        STAA    PIA0+2
        CLI
*
        RTS
*
*
**************************************
* LEVER INTERRUPT PROCESS
*
*
LEVER1  EQU     *
        ORG     XLVR1
        FDB     LEVER1
        ORG     LEVER1
*
        LDAB    #SMPIA1 WAIT FOR CA1
        CLRA    INTERRUPT (FROM LEVER)
        SWI
*
        LDAB    LVR1M
        RORB
        BCC     LVR11
* COUNT LEVER PRESS
        LDX     LVR1CN
        INX
        STX     LVR1CN
LVR11   RORB
        BCC     LEVER1
* COUNT STIMULATION
        LDX     STMCNT
        INX
        STX     STMCNT
* SIGNAL STIMULATION TO START
```

```
        LDAB    #SMSG0
        LDAA    #1
        SWI
*
        BRA     LEVER1
*
*****************************************
        END
```