united nations educational, scientific and cultural organization

international atomic energy agency

the
abdus salam
international centre for theoretical physics

The Sixth College on Microprocessor-Based Real-Time Systems in Physics

9 October - 3 November 2000

LECTURE NOTES

Volume I

MIRAMARE-TRIESTE

October 2000

Editors:
Abhaya S. Induruwa
Catharinus Verkerk

Foreword

The present volumes contain notes of the lectures delivered at the Sixth College on Microprocessor-based Real-time Systems in Physics, held at the Abdus Salam International Centre for Theoretical Physics, Trieste, Italy from October 9 till November 3, 2000. It is hoped that these notes provide a readable record of the College.

The "Realtime Colleges" are an outgrowth of the "Microprocessor Colleges" which were organized since 1981 under the impulse of late Professor Abdus Salam and Professor Luciano Bertocchi, and of which several were held in developing countries. All these Colleges were sponsored by UNESCO, IAEA and UNU.

From the beginning, laboratory exercises and projects formed an essential ingredient of the course. They made use of equipment developed in-house until 1994, when a shift was made to the use of the Linux operating system and PCs.

Over the past few years, a number of changes and improvements were made to the programme of lectures, exercises and projects. Particular attention was given to configuring the Linux operating system to present a nice-looking and user-friendly interface. More emphasis was put on the development of embedded systems. To this end new boards were designed and produced in Turkey and Malaysia and the necessary resident software and tools for cross-development implemented. Another change was the distribution of lecture notes in book form, instead of copies of transparencies.

The preparation of all these courses required a large effort from a number of people. We gratefully acknowledge the essential contributions of Chu Suan Ang, Paul Bartholdi, Manuel Gonçalves, Ravindra Karnad, Carlos Kavka, Anton Lavrentev, Ulrich Raich, Pablo Santamarina, Olexiy Tykhomyrov and Jim Wetherilt. Without their great efforts, it would not have been possible to constantly develop and implement new ideas and to keep the College up to date. We also wish to mention that in the past and present some 70-80 people, lecturers and instructors, have contributed to shaping this College into its present form. We gratefully acknowledge also their contributions and assistance.

We hope that the participants will enjoy the College and that they will benefit from it for their future activities.

Abhaya S. Induruwa,
Catharinus Verkerk,
Directors of the College,
Trieste, October 2000.

Contents

Principles of Real-Time Operating Systems, or Toward Real-Time

LINUX RELEASE 2.2.20

Sixth College on Microprocessor-based Real-time Systems in Physics

Abdus Salam ICTP, Trieste, Italy. October 9–November 3, 2000

Catharinus Verkerk[*]
email: Rinus.Verkerk@cern.ch

Abstract

We examine the operating system support needed for a real-time application. We'll see to what extent Linux satisfies the requirements and what has been done to adapt it.

[*]At present visitor at the Abdus Salam International Centre for Theoretical Physics, Trieste, Italy

1 Introduction and a few definitions

A *real-time system* is defined as a system that *responds to an external stimulus within a specified, short time.* This definition covers a very large range of systems. For instance, a data-base management system can justly claim to operate in real-time, if the operator receives replies to his queries within a few seconds. As soon as the operator would have to wait for a reply for more than, say, 5 seconds, she would get annoyed by the slow response and maybe she would object to the adjective "real-time" being used for the system. Apart from having unhappy users, such a slow data-base query system would still be considered a real-time system.

The real-time systems we want to deal with are much more strict in requiring short response times than a human operator is, generally speaking. Response times well below a second are usually asked for, and often a delay of a few milliseconds is already unacceptable. In very critical applications the response may even have to arrive in a few tens of microseconds.

In order to claim rightly that we are having a real-time system, we must specify the response time of the system. If this response time can be occasionally exceeded, without any real harm being done, we are dealing with a *soft real-time system.* On the contrary, if it is considered to be a *failure* when the system does not respond within the specified time, we are having a *hard real-time system.* In a hard real-time system, exceeding the specified response time may well result in serious damage of one sort or another, or in extreme cases even in the loss of human life. A data-base query system will generally fall in the first category: it will make little difference if a human operator will have to wait occasionally 6 seconds, instead of the specified 5 seconds response time, and nobody will dare to speak of a failure, as long as the replies to the queries are correct. This does not mean that all data-base systems are soft real-time systems: a data-base may well be used inside a hard real-time system, and its response may become part of the overall reaction time of the system.

Data-base systems are not at the centre of our attention in this course; we rather are interested in systems which control the behaviour of some apparatus, machinery, or even an entire factory. We call these *real-time control systems.* We are litterally surrounded by such real-time control systems: video recorders, video cameras, CD players, microwave ovens, and washing machines are a few domestic examples. In the more technical sphere we will find the control of machine tools, of various functions of a car, of a chemical plant, etc., but also automatic pilots, robots, driver-less metro-trains, control of traffic-lights, and many, many more. Several of those systems are hard real-time systems: the automatic pilot is a good example.

We implicitly assumed that the systems we are dealing with are computer controlled. We are in fact interested in investigating the role the computer plays, what constraints are imposed by the part of the system external to the computer, or the environment in general, and what these constraints imply for the program that steers the entire process. We will pay particular attention to the role the underlying operating system plays and to what extent it may help in the development and or running of a real-time control system.

At this point we should define two classes of real-time systems. On the one hand we have *embedded systems*, where the controlling microprocessor is an integral part of the entire product, invisible to the user and where the complete behaviour of the system is factory defined. The user can only issue a very limited and predefined set of instructions, usually with the help of switches, push-buttons and dials. There is no alpha-numeric keyboard available to give orders to the device, nor is there a general output device which can give information on the state of the system. On a washing machine we can select four or five different programs, which define if we will wash first with cold and then with warm water, or if we skip the first, or which define how often we will rinse, if we will use the centrifugal drying or not, etc. If we add the control the user has over the temperature of the water, we have practically exhausted the possibilities of user intervention. The microprocessor included in the system has been programmed in the factory and cannot be reprogrammed by the user. Cost has been the overriding design consideration, user convenience played a secondary or tertiary role. These embedded systems run a monolithic, factory defined program and there is no trace of an interface to an operating system which would allow a user to intervene. This does not mean that such an embedded system does not take account of a number of principles, which should not be neglected in a system that claims to operate in real-time. All real-time aspects are folded into the monolithic program, indistinguishable of the other functions of the program.

The other class of real-time control systems comprises those systems that make use of a normal computer, which has not been severed of its keyboard and of its display device and where a human being can follow in some detail how the controlled process is behaving and where he can intervene by setting or modifying parameters, or by requesting more detailed information, etc. The essential difference with an embedded system is that a system in this second class can be entirely reprogrammed, if desired. Also, in contrast to an embedded system, the computer is not necessarily dedicated to the controlled process, and its spare capacity may be used for other purposes. So, a secretary may type and print a letter, while the computer continues to control the assembly line.

It is obvious that the latter class of real-time control systems needs to run an operating system on the control computer. This operating system must be aware that it is controlling external equipment and that several operations initiated by it may be time-critical. The operating system must therefore be a **real-time operating system**. We will see in these lectures what this implies for the design and the capabilities of the operating system. We should keep in mind that we speak of generic real-time systems and generic real-time operating systems. The real-time control system does not necessarily use all features of the operating system, but the unused ones remain present, ready to be used at a possible later upgrade of the control system. This again is in contrast with the embedded system, where the parts of the operating system needed are cast in concrete inside the controlling program and where all other parts of it have been discarded.

2 The ingredients of a real-time computer controlled system

In order to investigate to some extent what the ingredients of a real-time control system are and what the implications are for a supporting operating system, we will take a simple example, which does not require any a-priori knowledge: a railway signalling system.

Safety in a railway system, and in particular collision-avoidance is based on a very simple principle. A railway track, for instance connecting two cities, is divided into sections of a few kilometers length each (the exact length depends on the amount of traffic and the average speed of the trains). Access to a section — called a block in railway jargon — is protected by a signal or a semaphore: when the signal exhibits a red light, access to the block is prohibited and a train should stop. A green light indicates that the road is free and that a train may proceed. The colour of the light is pre-announced some distance ahead, so that a train may slow down and stop in time. Access to a block is allowed if and only if there is no train already present in the block and prohibited as long as the block is "occupied". Normally all signals exhibit a red light; a signal is put to green only a short time before the expected passage of a train and if the condition mentioned above is satisfied. Immediately after the passage of the train, the signal is put back to red. The previous block is considered to be free only when the entire train has left it.

We will try to outline briefly -and rather superficiously -what would be required if we decided to make a centralized, computer controlled system for the signalling of the entire railway system in a small or medium-sized country, comprising a few thousand kilometers of track, with hundred or so trains running simultaneously. This would be a large-scale system, but it would be conceptually rather simple. The basic rule is: if there is a train moving forward in block $i - 1$, and block i is free, the signal protecting the entrance to block i shall be put to green and back to red again as soon as the first part of the train has entered block i. For the time being we consider only double track inter-city connections, where trains are always running in the same direction on a given track.

From the rule we see that we need to know at any instant in time which blocks are free and which are occupied. So we need a sort of a *data-base* to contain this information. This data base must be regularly updated, to reflect faithfully the real situation. In fact, whenever a train is leaving a block and entering another, the data-base must be updated.

How do we know that a train moves from one block to the next? Trains are supposed to run according to a time table and at predefined speeds, so a simple algorithm should be able to provide the positions of all trains in the system at any moment. Unfortunately this assumption is not valid under all circumstances and we need a reliable signalling system, exactly to be able to cope with more or less unexpected situations where trains run too late, or not at all, or where an extra train has been added, or another ran into trouble somewhere. We conclude that it is better to actually *measure* the event that a train crosses the boundary between two blocks. We could put a switch on the rails, which would be closed by the train when it is on top of the switch. We could scan all the switches in our system at regular intervals. How long — or rather how short — should this interval be? A TGV of 200 meter length and running at close to 300 km/h, would be on top of a switch for $2\frac{1}{2}$ seconds. A lonely locomotive, running at 100 km/h would remain on top of the contact for much less than a second. So we must scan some thousand or more contacts in, say $\frac{1}{2}$ second. This can be done, but it would impose a heavy load on the system and we would find the vast majority of the switches open in any case. We could refine our method and scan only those contacts where we expect a train to arrive soon. This would reduce the load on the system, as only hundred or so contacts have to be scanned, but it still is not very satisfactory, as we will continue to find many open contacts. Note that instead of contacts, we could have used other detection methods: strain gauges on the rails, or photo-cells.

A better way of detecting the passage of a train, is by using hardware interrupts [1]. We could generate an interrupt when the contact closes and

[1]For those who may have forgotten: a hardware interrupt is caused by an external

another when it opens again, indicating the entrance of a train into block i and the exit of the same train from block $i-1$, respectively. We don't lose time then anymore for looking at open contacts. We also simplify the procedure, for we do not have to look anymore at the data-base before the start of a scan, to find out which contacts are likely to be closed by a train soon.

We have discovered here a very important ingredient of any real-time control system: the *instrumentation with sensors and actuators.* In our case we must sense the presence of a train at given positions along the tracks, and we must actuate the signals, putting them to green and to red again. Generally speaking, the instrumentation of a real-time control system is a very important aspect, which must be carefully considered. Usually, apart from *sensors which provide single-bit information*, such as switches, push-buttons, photocells, *which can also be used to generate hardware interrupts*, we will need *measuring devices, giving an analog voltage output*, which then has to be *converted into a digital value* with an analog-to-digital converter. Conversely, *output devices may be single bit*, such as relays, lamps and the like, or *digital values, to be converted into analog voltages.* **Accuracy, reproducibility, voltage range, frequency response** etc. have to be considered carefully. The operation of a system may critically depend on how it has been instrumented. The *interface to the computer* is another aspect to take into account for its possible consequences. **Speed, reliability and cost** are some of the concurring aspects. We will not dwell any further on these topics in these lectures, as they are too closely related to the particular application, making a general treatment impossible.

For our railway signalling system we mentioned the timetable, claiming that we could not rely on it. We can however use it to check the true situation against it in order to detect any anomaly. These anomalies could then be reported immediately. For instance, we could tell the station master of the destination, that the train is likely to have a delay of x minutes. Another useful thing is to keep a *log* of the situation. This can be used for daily reports to the direction (where they would probably be filed away immediately), but they could prove valuable for extracting statistics and for global improvement of the system. *Operator intervention* is also needed. For instance, when a train, running from station A to station B, leaves station A, it does not yet exist in the data-base of running trains. Likewise, when it arrives at B, it has to be removed from this data-base. This could be

electrical signal. The normal flow of the program is interrupted and a jump to a fixed address occurs, where some work is done to *handle* the interrupt. A "return from interrupt" instruction brings us back to the point where the program was interrupted.

done automatically, in principle, but what do we do if it has been decided to run two extra trains, because there is an important football match? We conclude that *data-logging, operator intervention* and some *calculations* (to check actual situation against predicted one) are also essential ingredients of a real-time control system, in addition to the *interrupt handling, interfacing to the sensors and actuators* and *updating of the data-base* reflecting the state of the system.

This idyllic picture of our railway signalling system might stimulate us to start coding immediately. A program which uses the principles outlined above does not seem too difficult to produce. We simply let the program execute a large loop, where all different tasks are done one after the other. The interrupts have made it possible to get rid of a serious constraint, so all seems to be nice and straightforward. Once we would have a first version of the program ready, we would like to test it. Hopefully we will use some sort of a test rig at this stage, and abstain from experimenting with real trains. During the testing stage, we will then quickly wake up and find that we have to face reality.

In our model, we assumed double track connections between cities, where on a given track, trains always run in the same direction. But, even in the case that the entire railway network is double track between cities, we must nevertheless consider also single track operation, because a double track connection may have to be operated for a limited period of time and for a limited distance as a single track, repair or maintenance work making the other track unusable.

Assume that, on a single track, we have two trains, one in block $k+1$, the other in block $k-1$, running in opposite directions, both toward block k. If we would apply our simple rule, they would both be allowed to enter block k (supposing it was free) and a head-on collision would result. The problem can be solved by slightly modifying our rule: If a train is moving forward in block $i-1$ toward block i, then access to block i will be allowed if blocks i and $i+1$ are free. So both trains will be denied access to block k in our example. We have eliminated the possibility of a head-on collision, but we now have another problem. Assume that our two trains are in block $k-2$ and $k+1$ respectively and running toward each other. Applying our new rule, they would be allowed to enter block $k-1$ and k respectively and both trains would stop, nose to nose at the boundary between these two blocks. We have created a sort of a *deadlock situation.*

The true solution is of course not to allow a south-bound train into an entire section of single track, as long as there is still a north-bound train somewhere in this entire section, and vice-versa. A section consists of several blocks and inside a section there are no switches enabling a train to move from one track to another, nor to put it on a side-track. South-bound and north-bound trains compete for the same "**resource**", the piece of single track railway. They are **mutually exclusive** and only one type (north-bound or south-bound) of train should be allowed to use the resource. If the stretch of single track is long enough, and comprises several blocks, more than one north-bound train can be running on that stretch of track. Now assume that several north-bound trains are occupying the stretch of single track and that a south-bound train presents itself at the nothern end of the stretch. It obviously has to wait, but while it is waiting, do we continue to allow more north-bound trains into the stretch? This is a matter of *priority*, which should be defined for each train. A *scheduler* should take the priorities into account and deny the entrance into the stretch for a north-bound train if the waiting south-bound one has higher priority. As soon as the stretch has then been emptied of all north-going trains, the south-bound one can proceed, possibly followed by others.

A similar situation, where two trains may be competing for the same resource, arises when two tracks, coming from cities A and B, merge into a single track entering city C. Obviously, if two trains approach the junction simultaneously, only one can be allowed to proceed, which should be the one with the highest priority. It should be noted that the priority assigned to a train is not necessarily static. It may change dynamically. For instance, a train running behind schedule, may have its priority increased at the approach of the junction and allowed to enter city C, before another train which normally would have had precedence. This latter example illustrates a *synchronization problem*: some trains may carry passengers which have to change trains in city C; the two trains should reach the station of C in the right order.

We have thus discovered some more ingredients (or concepts) for a real-time control system: **priorities, mutual exclusion, synchronization**.

We started off by considering our railway signalling problem being controlled by a single program, which guides all trains through all tracks, junctions and crossings. We have gradually come to have a different look at the problem: *a set of trains, using resources* (pieces of railway track), *and sometimes competing for the same resource*. We can consider our trains as *independent objects*, more or less unaware of the existence of similar objects and of the competition this may imply. In order to get a resource, every train must put forward a request to some sort of a master mind (the real-time operating system), who will honour the request, or put the train in a waiting state.

At this stage, we realize that we better abandon our first version of the program, because it would have to be rewritten from scratch in any case. We have become aware that our particular real-time control system may have many things in common with other real-time systems and that it would be advantageous to take profit from the facilities a real-time operating system offers to solve the problems of mutual exclusion, priorities, etc. Once we have mastered the use of these facilities, we can build on our experience for the implementation of another real-time control system. In case we would obstinately continue to adapt our original program, we would probably find, after months of effort, that we have rewritten large parts of a real-time operating system, but which have been so intimately interwoven with the application program, that it will be difficult, if not impossible, to re-use it for the next application we may be called to tackle.

Other aspects we have not yet considered may also build very nicely on the foundations laid by a real-time operating system. For instance, we have the problem of dealing with *emergencies*. A train may have derailed and obstructed both tracks. Such an unusual and potentially dangerous situation must be immediately notified to the operating system which can then take the necessary measures. If they cannot be notified, a mechanism for detecting potentially dangerous situations must be devised: in our particular case, the system should be alerted if a train does not leave its block within a reasonable time. In other words, a *time-out* could be detected.

Now that we mentioned time, we are reminded of the fact that **time** may play an important role in any real-time system, either in the form of *elapsed time*, or of the *time of the day*. It is difficult to think of a system that could operate without the help of a clock. A **real-time clock** and the possibility to program it to generate a clock interrupt at certain points in time, or after a given time-interval has elapsed, are therefore indispensable ingredients of a real-time control system.

Reliability of the entire system is another item for serious consideration. You certainly do not want a parity error in a disk record to bring your system to a halt or to create a chain of very nasty incidents.

In many cases, we are not dealing with a closed system, so there must be a means of *communicating with other systems* (our national railway network is connected to other networks, and trains do regularly cross the border). *User-friendly interfaces to human operators*, which usually implies the use of graphics, are also very likely to be an essential ingredient of our real-time control system. A large synoptic panel, showing where all trains are in the network, would be the supervisor's dream, not to speak of makers of science fiction films.

In the following lectures, we will investigate in more depth the various features a real-time operating system should provide. Making use of these features will prevent us from re-inventing the wheel.

The question then arises: which real-time operating system should I use? There are several on the market: *OS-9000* for *Motorola 68000 machines*, and *QNX* or *LynxOS* for *Intel machines*, *Solaris* for *Sparc processors*, to mention only a few of the older ones. These systems are sold together with the tools necessary to build a real-time application: **compiler, assembler, shell, editor, simulator, etc.** A minimum configuration would cost US$ 2000-2500, a full configuration may push the price up into the 10 K$ range. This would cause no problem whatsoever for a railway company, but what about you?

Another solution is to use a **real-time kernel**, useful for embedded systems, which you *compile and link into your application. VxWorks, MCX11* and *μCOS* are examples. They are much cheaper —or practically free: *MCX11* and *μCOS* [2]—, but you will need a complete development system in addition. This development system could of course be Linux.

The ideal would be to be able to **use Linux for development of a real-time control system, as well as for running the application**. We will see shortly to what extent this is possible at present. Before proceeding, however, we will make sure that we understand the fundamental concept of a **process**.

3 Processes

In our example we have seen that a real-time system has a number of tasks to accomplish: besides ensuring that trains could proceed from block to block without making collisions, we had to log data, keep the data-base up-to-date, communicate with the operator, cater for emergency situations, etc. Not all of these tasks have the same priority, of course.

When we analyze a real-time system, we will almost invariably be able to identify *different tasks*, which are more or less *independent of each other*. "Independent of each other" really means that each task can be programmed without thinking too much of the other tasks the system is to perform. At most there is some *intertask communication*, but every task does its job on its own, without requiring assistance from other tasks. If assistance is required, the operating system should provide it. The system designer should identify and define the different tasks in such a way that they really are as independent of each other as possible. Some *synchronization* may be needed: certain tasks can only run after another task has completed. For instance, if some calculations have to be done on collected data, the data collection tasks could be totally separated from the calculation task. In order to make

[2] *RTEMS* is a more recent, very complete real-time executive, also available for free.

sense, the latter should only be executed when the data collection task has obtained all data necessary for the calculation. This implies that some inter-task communication is needed here. The true difficulty of dividing the overall system requirement up into different tasks consists of choosing the tasks for maximum independence, or —in other words— for **minimum need of inter-task communication and synchronization**.

These various tasks can now be implemented as different programs and then run as different **processes**.

What exactly is a process and what is the *difference between a program and a process?* A program is an orderly sequence of machine instructions, which could have been obtained by compilation of a sequence of high-level programming language statements. It is not much more than the listing of these statements, which can be stored on disk, or archived in a filing cabinet. It becomes useful only when it is run on a machine and executing its instructions in the desired sequence, thus obtaining some result. It is only useful when it has become a *running process.*

A process is therefore a running (or runnable) program, together with its data, stack, files, etc. It is only when the code of a program has been loaded into memory, and data and stack space allocated to it, that it becomes a runnable process. The operating system will then have set up an entry in the *process descriptor table*, which is also part of the process, in the sense that this information would disappear when the process itself ceases to exist. The operating system may decide at a certain moment to run this runnable process, on the basis of its priority and the priorities of other runnable processes. This would happen in general when the process that is using the CPU is unable to proceed — e.g. because it is waiting for input to become available —, or because the time allocated to it has run out.

We should emphasize that we are considering only the case of a single processor system, where only one process can run at a time. The other runnable processes will wait for the CPU to become free again. If the different processes are run in quick succession, a human observer would have the impression that these processes are executed simultaneously.

The consequence of this is that we can write a program to calculate Bessel functions, without having to think at all about the fact that when we will run our program, there may already be fifty or more other processes running, some of them even calculating Bessel functions. In as far as we have written our program to be autonomous, it will not be aware of the existence of other runnable processes in the computer system. Consequently, it cannot communicate with the other processes either: its fate is entirely in the hands of the operating system[3].

There may exist on the disk a general program to calculate Bessel functions and on a general purpose time-sharing computer system several users may be running this program. A reasonable operating system should then keep only one copy of the program code in memory, but each user process running this program should have its *own* process descriptor, its *own* data area in memory, its *own* stack and its *own* files. All users of the computer system will presumably run a **shell**. Command shells, such as *bash* or *tcsh* are very large programs and it would be an enormous waste if every single user of a time-sharing system would have his own copy of the shell in memory.

In general, we will have a number of runnable processes in our uniprocessor machine, and one process running at a given instant of time. When will the waiting processes get a chance to run? There are two reasons for suspending the execution of the running process: either the *time-slice* allocated to it *has been exhausted*, or *it cannot proceed any further before some event happens*. For instance, the process must wait for input data to become available, or for a *signal* from another process or the operating system, or it has to complete an output operation first, etc. The programmer does not have to bother about this. At a given point in the program, where it needs to have more input data, the programmer simply writes a statement such as: *read(file,buffer,n);*. The compiler will translate this into a call to a *library function*, which in turn will make a **system call**, (or **service request**), which will transfer control to the kernel. Our process becomes *suspended* for the time the kernel needs to process this system call. In the case of a read operation on a file, the kernel will set this into motion, by emitting the necessary orders to the disk controller. As the disk controller will need time to execute this order, the kernel will decide to **block** the execution of the process which was running and which made the system call. This blocked process will be put in the *queue of waiting processes*, and it will become runnable again later, when the disk controller will have notified the kernel —by sending a hardware interrupt— that the I/O operation has been completed. The kernel makes use of the **scheduler** to find, from the queue of runnable processes the one that should now be run. The kernel will then make a **context switch** and this will start our suspended process running.

A context switch is a relatively heavy affair: first all hardware registers of the old (running) process must be saved in the process descriptor of the old process. Then the new process must be selected by the scheduler. If the code and data and stack of the new process are not yet available in memory,

[3]And luckily so: the operating system will also provide protection, avoiding that other processes interfere with ours.

they must be loaded. In order to be loaded, it may be necessary first to make room in memory, by swapping out some memory pages which are no longer needed or which are rarely used. The page tables must be updated, and the process descriptors must be modified to reflect the new situation. Finally the hardware registers of our machine must be restored from the values saved at an earlier occasion for the process now ready to start running. The last register to be restored is the program counter. The next machine instruction executed will then be exactly the one where the new process left off when it was suspended the last time.

The execution of a program will thus proceed piecemeal, but without the programmer having to bother about it: the operating system takes care of everything. So the application programmer can continue to believe that his program is the only one in the world. The price to be paid for this convenience is the overhead in time and memory resources introduced by the intervention of the operating system.

For our Bessel function program we are entirely justified in thinking that we are alone in the world. There are however situations where this is not the case and where different processes interfere with each other, either willingly or unwillingly. Here is my favourite example of such a case of interference[4].

Assume that we have three separate bytes in memory which contain the hour, minutes and seconds of the time of the day. There is a hardware device which produces an interrupt every second and this will cause the process that will update these three bytes to be woken up. Any process which wants to know the time, can access these three memory bytes, one after the other (we assume that our machine can address only one byte at a time). Now suppose that it is 10.59.59 and that a process has just read the first byte "10", when a clock interrupt occurs. As the process that updates the clock has a higher priority than the running process, the latter is suspended. The clock process now updates the time, setting it to 11.00.00. Control then returns to the first process which continues reading the next two bytes. The result is: 10.00.00; which is one hour wrong. What happened here is that *two processes access the same resource* — the three memory bytes — and that one or both of them can alter the contents. No harm would be done if both processes had *read-only* access to the shared resource.

The reader should note that the concept of a process has allowed us to speak about them as if they were really running simultaneously. We do not have to include in our reasoning the fact that there is a context switch and that complicated things are going on behind the scenes. We only have to be

[4]The reader should be aware that the example describes a primitive situation; no modern operating system would allow this situation to occur.

aware that access to shared resources must be protected, in order to avoid that another process accesses the same resource "simultaneously". On a multi-processor system "simultaneous" can really mean "at the same instant in time", on a uni-processor machine it really means "concurrently". The processor concept is equally valid for a uni- and a multi-processor machine.

The places in the program where a shared resource is accessed are so-called **critical regions**. We must avoid that two processes access simultaneously the resource and this can be done by ensuring that a process cannot enter a critical region when another process is already in a critical region where it accesses the same shared resource. The entrance to a critical region must be protected with a sort of a lock.

Two operations are defined on such a lock: *lock* and *unlock*. The lock operation tests the state of the lock and if it is unlocked, locks it. The test and the locking are done in a single **atomic** operation. If the lock is already locked, the lock operation will stop the process from entering the critical region. The unlock operation will simply clear a lock which was locked, and allow the other process access to the critical region again. That these operations must be *atomic* means that it must be impossible to interrupt them in the middle. Otherwise we would get into awkward situations again. If the lock operation would not be atomic, we could have a situation where process 1 inspects the lock and finds it open. If immediately after this, process 1 gets interrupted, before it had a chance to close the lock, process 2 could then also inspect the lock. It finds that it is open, sets it to closed, enters the critical region where it grabs the resource (a printer for instance) and starts using it. Some time later process 1 will run again, it will also close the lock and it will also grab the same printer and start using it. Remember that process 1 previously had found the lock to be open and it is unaware that process 2 has been running in the meantime!

The lock and unlock operations must therefore be completed before an interruption is allowed. This can be done —primitively— by disabling interrupts and then enabling them after the operation. No reasonable operating system would allow a normal user to tinker with the interrupts, so most machines have a *test-and-set* instruction. The *test-and-set* instruction tests a bit and sets it to "one" if it was "zero". If it was already "one" it is left unchanged. The result of the test (i.e. the state of the bit before the *test-and-set* instruction was executed is available in the processor status word and can be tested by a subsequent *branch* instruction. The *test-and-set* instruction is a single instruction; a hardware interrupt arriving during the execution of the instruction will be recognized only after the execution is complete. This guarantees the atomicity of a *test-and-set* operation.

What do we do after the *test-and-set* instruction? If the lock was open, you can safely enter the critical region. If, on the contrary, process A finds the lock closed, it should go to **sleep**. The operating system will then suspend the execution of process A and schedule another process to run, say C, or E. The process B, which had closed the lock in the first place, will also be running again at some instant and eventually will unlock the lock and **wakeup**[5] the sleeping process A. The system will then make process A runnable again.

Now suppose that process A gets interrupted immediately after doing its — unsuccessful — lock operation and before it could execute the *sleep()* call. Process B will at some stage open the lock and wakeup A. As A is not sleeping, this wakeup is simply lost. When A will run again, it will truely go to sleep, this time forever.

The solution to the problem was given in 1965 by Dijkstra, when he defined the **semaphore**. A semaphore can count the number of such "*lost wakeups*", without trying to wake up a process that is not sleeping. It can therefore only have a positive value, or "0". Two **atomic operations** are defined on a semaphore, which we will call **up** and **down**[6]. Once an operation on a semaphore is started, no other process can access the same semaphore. Thus **atomicity** of a semaphore operation is guaranteed. The work done for a *down* (and similarly for an *up*) operation must therefore be part of the operating system and not of a user process. The *down* operation checks the value of the semaphore. If it is greater than zero, it decrements the value and the calling process just continues execution. On the contrary, If the *down* operation finds that the semaphore value is zero, the calling process is put to sleep. The *up* operation on a semaphore increments its value. If one or more processes were sleeping, one of them is selected by the operating system. The selected process will then be allowed to run can now complete its *down*, which had failed earlier. Thus, if the semaphore was positive, it will simply be incremented, but if it was "0" — meaning that there are processes sleeping on it — its value will remain "0", but there will be one process less sleeping.

We have described the general form of a semaphore: the **counting semaphore**, which is used to solve **synchronization** problems, ensuring that certain events happen in the correct order. A **binary semaphore** can only take the values "0" or "1" and is particularly suited for solving problems of *mutual exclusion*, which explains its other name: **mutex**.

To illustrate the use of mutexes and counting semaphores we show an example of the **Producer-Consumer** problem. Suppose we have two col-

[5]Process B itself does not directly wakeup A, of course. The operating system takes care of doing it.

[6]Various other names are also used: post and signal, P and V (the original names given by Dijkstra), and possibly others. For mutexes, lock and unlock are often used.

laborating processes: a *producer* which produces items and puts them in a *buffer* of finite size, and a *consumer* which takes items out of the buffer and consumes them. A data acquisition system which writes the collected data to tape is a good example of a producer-consumer problem. It is clear that the producer should stop producing when the buffer is full; likewise, the consumer should go to sleep when the buffer is empty. The consumer should wake up when there are again items in the buffer and the producer can start working again when some room in the buffer has been freed by the consumer.

```
#define N 100                /* number of slots in buffer */
typedef int semaphore;       /* this is NOT POSIX !!  */
semaphore mutex=1;           /* controls access to critical region */
semaphore empty=N;           /* counts empty buffer slots */
semaphore full=0;            /* counts full buffer slots*/

void producer(void)
{
  int item;
  while(TRUE) {              /* do forever (TRUE=1) */
      produce_item(&item);   /* make something to put in buffer */
      down(&empty);          /* decrement empty count */
      down(&mutex);          /* enter critical region */
      enter_item(&item);     /* put new item in buffer */
      up(&mutex);            /* leave critical region */
      up(&full);             /* increment count of full slots */
  }
}

void consumer(void)
{
  int item;
  while(TRUE) {              /* do forever */
      down(&full);           /* decrement full count */
      down(&mutex);          /* enter critical region */
      remove_item(&item);    /* take item from buffer */
      up(&mutex);            /* leave critical region */
      up(&empty);            /* increment count of empty slots */
      consume_item(&item)    /* use the item */
  }
}
```

In order to obtain this synchronization between the two processes, two *counting semaphores* are used: *full* which is initialized to "0" and counts the buffer slots which are filled, and *empty*, initialized to the size of the buffer and which counts the empty slots. Access to the buffer, which is shared between the two processes, is protected by a *mutex*, initially "1" and thus allowing access. The example is taken from Andrew Tanenbaum's excellent book[7]. The reader should study carefully the listing of the *Producer-Consumer* problem on the previous page. He should be aware that the example is simplified: instead of two *processes* and a buffer structure in *shared memory*, the listing shows two functions, using global variables. Also the semaphores are not exactly what the standards prescribe. Using semaphores and mutexes remains a difficult thing: changing the order of two *down* operations in the listing below may result in chaos again.

4 What is wrong with Linux?

UNIX has the bad reputation of not being a real-time operating system. This needs some explanation. Time is an essential ingredient of a real-time system: the definition says that a real-time system must respond within a given time to an external stimulus. Theoretically, it is not possible to guarantee on a general UNIX time-sharing system that the response will occur within a specified time. Although in general the response will be available within a reasonable time, the load on the system cannot be predicted and unexpected delays may occur. It would be a bad idea to try and run a time-critical real-time application on an overloaded campus computer. Nevertheless, before discarding altogether the idea of using UNIX or Linux as the underlying operating system for a real-time application, we should have a critical look at what the requirements really are, to what extent they are satisfied by off-the-shelf Linux, and what can be done (or has been done already) to improve the situation.

The UNIX and Linux schedulers have been designed for **time-sharing** the CPU between a large number of users (or processes). It has been designed to give a *fair share of the resources*, in particular of CPU time, to all of these processes. The priorities of the various processes are therefore adjusted regularly in order to achieve this. For instance, the numerical analyst who runs CPU-intensive programs and does practically no I/O, will be penalized,

[7]Andrew S. Tanenbaum, Modern Operating Systems, Prentice Hall International Editions, 1992, ISBN 0-13-595752-4. The reader is encouraged to read the chapter on Interprocess Communication, which provides a much more detailed treatment of synchronization problems than is possible here.

to avoid that he absorbs all the CPU time.

Such a scheduling algorithm is not suitable for running a real-time application. If the operating system would decide that this particularly demanding application had consumed a sufficiently large portion of the available CPU time, it would lower its priority and the application might not be able anymore to meet its deadlines.

A real-time application must have **high priority** and — in order to be able to meet its deadlines — *must run whenever there is no runnable program with a higher priority.* In practice, the real-time process should have the highest priority, and it should keep this highest priority throughout its entire life[8]. Another scheduling algorithm is therefore required: a certain class of processes should be allocated permanently the highest priorities defined in the system. The normal scheduler of Linux did not have this feature, but *another scheduler*, designed for mixed time-sharing and real-time use is available and *is usually compiled into the kernel.*

Time being a precious resource for a real-time system, overheads imposed by the operating system should be avoided as much as possible. Some of the overheads can be avoided by careful design of the real-time program. For instance, knowing that forking a new process is a time-consuming business, all processes which the real-time application may need to run, should be forked and exec'ed (the *fork* and *exec* system calls will be illustrated in section 5) *during the initialization phase* of the application. Other overheads cannot be avoided so simply and need some adaptation or modification of the operating system.

Context switches may be very expensive in time, in particular when the code of the new process to be run is not yet available in memory and/or when room must be made in memory. All code and data of a real-time application should be **locked into memory**, so that this part of a context switch would not cause a loss of time. Locking everything into memory will also prevent *page faults* to happen, avoiding this way other *memory swapping* operations. Originally Linux did not have the possibility of locking processes into memory, but again, *memory locking* is now compiled into all recent kernels.

A further help in reducing the overheads due to context switches is to use so-called *light-weight processes* or **multi-threaded user processes**. Linux as such does not provide these, but *library implementations do exist* to implement the standard POSIX pthreads.

[8]It would be wise to run a shell with an even higher priority, in order to be able to intervene when the real-time process runs out of hand. This shell would be sleeping, until it gets woken up by a keystroke.

Other places where to watch for lurking losses of time are Input/Output operations. Normally, when a file is opened for writing, an initial block of disc sectors is allocated — usually 4096 bytes — and *inodes* and *directory entries* are updated. When the file grows beyond its allocated size, the relatively lengthy process of finding another free block of 4096 bytes and updating inodes and directory entries is repeated. A real-time system should be able to grab all the disc space it needs during initialization, so that these time losses may be avoided. Linux does not allow this at present.

All input and output in Linux is **synchronous**. This means that a process requesting an I/O operation will be *blocked until the operation is complete* (or an error is returned). Upon completion of the operation, the process becomes *runnable* again and it will effectively run when the scheduler decides so. However, "completion" of an output operation means only that the data have arrived in an output buffer, and there is no guarantee that the data have really been written out to tape or disc. When the process is only notified of completion of the I/O operation when the data are really in their final destination, we have **synchronized I/O**, which may be a necessity for certain real-time problems. Linux does not spontaneously do *synchronized I/O*, but it *can be easily imposed by using sync or fsync.*

Asynchronous I/O may be another real-time requirement. It means that *the process* requesting the I/O operation *should not block* and wait for completion, *but continue processing* immediately after making the I/O system call. The standard device drivers of Linux do not work asynchronously, but the primitive system calls allow the option of continuing processing. A special purpose device driver could make use of this and thus do asynchronous I/O. The process will then be notified with an interrupt when the I/O operation has been completed.

The designer of a real-time system should of course also be aware that no standard device drivers exist for exotic[9] devices. They have to be written by the application programmer. In a standard UNIX system, such a new device driver must be compiled and linked into the kernel. Linux has a very nice feature: it allows to *dynamically load and link to the kernel so-called modules*, which can be — and very often are — device drivers.

We have shown before that it would be wise to divide a real-time system up into a set of processes, which can each care for their own business, without excessively interfering with each other. Nevertheless, some communication between processes may be needed. Old UNIX systems had only two interprocess communication mechanisms: **pipes** and **signals**. Signals

[9]With exotic I really mean *very weird non-standard devices.* The list of devices supported by Linux is indeed incredibly long!

have a very low information content, and only two user definable signals exist. System V UNIX added other IPC mechanisms: sets of **counting semaphores, message queues**, and **shared memory**. Most Linux kernels have the System V IPC features compiled in.

Probably no real-time system could live without a **real-time clock** and **interval timers**. They do exist in off-the-shelf systems, but the *resolution*, usualy 1/50 th or 1/100 th of a second, may not be enough. The user-threads package can work with higher resolutions, if the hardware is adequate.

The IEEE has made a large effort to standardize the user interface to operating systems. The result of this effort has been the POSIX 1003.1a standard, which defines a set of system calls, and POSIX 1003.1b, which defines a standard set of Shell commands. Both were approved by IEEE and by ISO and thus gained international acceptance. Also **real-time extensions to operating systems** have been defined in the **POSIX 1003.1c-1994** standard, which has also been accepted by ISO. All the points discussed above are part of this POSIX 1003.1c standard, except for the **multiple threads and mutexes, which are defined in a later extension**. To the best of my knowledge, the so-called "**pthreads**" are now also an part of the international standard. Linux is POSIX 1003.1a and .1b compliant, although it may not have been officially certified.

In summary, Linux used to be weak on the following, and may still be on a few items:

- **Mutexes**. A simple mutex did not exist in the original Linux kernel. The System V IPC semaphores can be used, although they are overkill, introducing a large overhead. Atomic bit operations are defined in *asm/bitops.h* and can be used more easily, but care should be exercised (danger of *priority inversion*). Mutexes are defined in the **pthreads** package. They will work between user threads inside a single process, and for some implementations also between threads and another process.

- **Interprocess Communication**. System V IPC is usually part of the Linux kernel and adds counting semaphores, message queues and shared memory to the usual mechanisms of pipes and of signals.

- **Scheduling**. A POSIX 1003.1c compliant scheduler for Linux exists and is part of the kernel in most *Linux distributions*.

- **Memory Locking**. Memory locking is part and parcel of the more recent Linux kernels (at least above 2.0.x and maybe earlier).

- **Multiple User Threads**. A few library implementations exist. The more recent Linux distributions have Leroy's Pthread library, which makes use of a particular feature of Linux: the "*clone*" system call. It is entirely compliant with the POSIX standard. You will soon get into close contact with it.
- **Synchronized I/O**. Can be obtained easily with *sync* and *fsync*.
- **Asynchronous I/O**. Not available in standard device drivers. Could be implemented for special purpose device drivers.
- **Pre-allocation of file space**. Not available to my knowledge.
- **Fine-grained real-time clocks and interval timers**. They are part of the available pthreads packages and could be used if the hardware is capable.

5 Creating Processes

Creating a new process from within another process is done with the *fork() system call*. *fork()* creates a new process, called the **child process**, *which is an exact copy of the original (parent) process*, including open files, file pointers etc. Before the *fork()* call there is only *one* process; when the *fork()* has finished its job, there are *two*. In order to deal with this situation, *fork()* **returns twice**. To the parent process it returns the **process identification (PID)** of the child process, which will allow the parent to communicate later with the child. To the child process it returns a 0. As the two processes are exact copies of each other, an *if* statement can determine if we are executing the child or the parent process.

There is not much use of a child process which is an exact copy of its parent, so the first thing the child has to do is to load into memory the program code that it should execute and then start execution at *main()*. A child is obviously too inexperienced to do this on its own, so there is a system call that does it for him: *execl()*. The entire operation of creating a new process therefore goes as follows:

```
/* here we have been doing things */
child=fork();           /* PID of new process --> child */
if(child){              /* here for parent process */
                        /*continue parent's business*/
    }
else {                  /* here for child process */
```

```
    execl("/home/boss/rtapp/toggle_rail_signal",\
     "toggle_rail_signal", N_sigs, NULL);
    perror("execl");   /* here in case of error */
    exit(1);
    }
/* here continues what the parent was doing */
```

execl() will do what was described above, so in our example it will load the executable file */home/boss/rtapp/toggle_rail_signal* and then start execution of the new process at *main(argc,argv)*. The other arguments of *execl()* are passed on to *main()*. *execl* is one of six variants of the exec system call: *execl, execv, execle, execve, execlp, execvp*. They differ in the way the arguments are passed to *main()*: **l** means that a list of arguments is passed, **v** indicates that a pointer to a vector of arguments is passed. **e** tells that environment pointer of the parent is passed and the letter **p** means that the *environment variable* **PATH** should be used to find the executable file.

This completes the creation of a new process. On a single CPU machine, one of the two processes may continue execution, the other will wait till the scheduler decides to run it. There is no guarantee that the parent will run before the child or vice versa.

The new process can *exit()* normally when it has done its job, or when it hits an error condition. The parent can *wait* for the child to finish and then find out the reason of the child's death by executing one of the following system calls:

```
pid_t wait(int *status); /* wait for any child to die*/
or: pid_t waitpid(pid_t which, int *status, int options)
              /* wait for child "which" to die */
```

These wait calls can be useful for doing some cleaning-up and to avoid leaving *zombies* behind. When the parent process exits, the system will do all the necessary clean-up, childs included.

We can now understand what the shell does when we type a command, such as *cp file1 dir*. The shell will parse the command line, and assume that the first word is the name of an executable file. It will then do a *fork()*, creating a copy of the shell, followed by an *execl()* or *execv()* which will load the new program, in our example the copy utility *cp*. The rest of the command line is passed on to *cp* as a list or as a pointer to a vector. The shell then does a *wait()*. When an & had been appended to the command line, then the shell will *not* do a wait, but will continue execution after return from the *exec* call.

The following gives a more complete and rather realistic example of a *terminal server and a client*[10]. The reader is invited to study this example in detail.

The code for the **server** looks like:

```
#define      POSIX_C_SOURCE 199309

#include     <unistd.h>
#include     <stdio.h>
#include     <sys/types.h>
#include     <sys/wait.h>
#include     <signal.h>
#include     <errno.h>
#include     "app.h" /* local definitions */

main(int argc, char **argv)
{
    request_t r;
    pid_t terminated;
    int status;

    init_server();   /* set things up * /

    do {
        check_for_exited_children();
        r = await_request();   /*get some input*/
        service_request(r);   /*do what wanted*/
        send_reply(r);        /*tell we did it*/
    } while (r != NULL);

    shutdown_server();   /*tear things down*/
    exit(0);
}

void
service_request(request_t r)
{
    pid_t child;
    switch (r->r_op) {
```

[10]the example is taken from Bill O. Gallmeister, POSIX.4, Programming for the Real World, O'Reilly, 1995.

```
        case OP_NEW:
            /* Create a new client process */
            child = fork();
            if (child) {
                /* parent process */
                break;
            } else {
                /* child process */
                execlp("terminal","terminal \
                 application","/dev/com1",NULL);
                perror("execlp");
                exit(1);
            }
            break;
        default:
            printf("Bad op %d\n", r->r_op),
            break;
    }
    return;
}
```

The **terminal** end of the application looks like:

```
#include     <unistd.h>
#include     <stdio.h>
#include     <sys/types.h>
#include     <sys/wait.h>
#include     <signal.h>
#include     "app.h"  /* local definitions */

char *myname;

main(int argc, char **argv)
{
    myname = argv[0];
    printf("Terminal \"%s\" here!", myname);
    while (1) {
        /* deal with the screen */
        /* await user input */
    }
    exit(0);
}
```

Presumably *request_t* is defined in *app.h* as a pointer to a structure. *await_request()* is a function which sleeps until a service request arrives from a terminal. The operations performed by the other functions: *init_server, service_request(), check_for_exited_children(), send_reply()* and *shutdown_server()* are implied by their names.

6 Interprocess Communication

In the case where we have a real-time application with a number of processes running concurrently, it would be a normal situation when some of these processes need to communicate between them. We said already that the classical UNIX system only knows pipes and signals as communication mechanisms. Interprocess communication, suitable for real-time applications is an essential part of the POSIX standard, which adds a number of mechanisms to the minimal UNIX set. In the following we will briefly describe the various IPC mechanisms and how they can be invoked. We will follow as much as possible the POSIX standard, except where the facilities are not implemented in Linux. In that case we will describe the mechanism Linux makes available.

6.1 UNIX and POSIX 1003.1a Signals

The old signal facility of UNIX is rather limited, but it is available on every implementation of UNIX or one of its clones. Originally, signals were used to *kill* another process. Therefore, for historical reasons, the system call by which a process can send a signal to another process is called *kill()*. There is a set of signals, each identified by a number (they are defined in <*signal.h*>), and the complete system call for sending a signal to a process is:
`kill(pid_t pid, int signal);`
The integer *signal* is usually specified symbolically: *SIGINT, SIGALRM* or *SIGKILL*, etc., as defined in <*signal.h*>. *pid* is the process identification of the process to which the signal shall be sent. If this receiving process has not been set up to **intercept signals**, its *execution will simply be terminated by any signal sent to it.* The receiving process can however be set up to *intercept certain signals* and to perform certain actions upon reception of such an intercepted signal. Certain signals cannot be intercepted, they are just killers: SIGINT, SIGKILL are examples. In order *to intercept a signal,*

the receiving process must have *set up a signal handler* and notified this to the operating system with the *sigaction() system call.* The following is an example of how this can be done:

A *structure sigaction* (not to be confounded with the system call of the same name!) is defined as follows:

```
struct sigaction {
    void (*sa_handler)();
    sigset_t sa_mask;
    int sa_flags;
void(*sa_sigaction)(int,siginfo_t *,void *); };
```

This structure encapsulates the action to be taken on receipt of a signal.

The following is a program that shall exit gracefully when it receives the signal *SIGUSR1*. The function *terminate_normally()* is the **signal handler**. The administrative things are accomplished by defining the elements of the structure and then calling *sigaction()* to get the signal handler registered by the operating system.

```
void
terminate_normally(int signo)
{
    /* Exit gracefully */
    exit(0);
}

main(int argc, char **argv)
{
    struct sigaction sa;
    sa.sa_handler = terminate_normally;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = 0;
    if (sigaction(SIGUSR1, &sa, NULL)) {
        perror("sigaction");
        exit(1);
    }
  ...
}
```

The operating system itself may generate signals, for instance as the result of machine exceptions: *floating point exception*, *page fault*, etc. Signals may also be generated by something which happens *asynchronously* with the process itself. The signals then *aim at interrupting the process*: I/O completion, timer expiration, receipt of a message on an empty message queue, or typing CTRL-C or CTRL-Z on the keyboard. Signals can also be sent from one user process to another.

The structure *sigaction* does not only contain the information needed to register the signal handler with the operating system (in the process descriptor), but it also contains information on what the receiving process should do when it receives the registered signal. It can do one of three things with the signal:
– it can block the signal for some time and later unblock it.
– it can ignore the signal, pretending that nothing has happened.
– it can handle the signal, by executing the signal handler.

The POSIX.1 signals, described so far, have some serious limitations:
– there is a lack of signals for use by a user application (there are only two: SIGUSR1 and SIGUSR2).
– signals are not queued. If a second signal is sent to a process before the first one could be handled, the first one is simply and irrevocably lost.
– signals do not carry any information, except for the number of the signal.
– and, last but not least, signals are sent and received asynchronously. This means in fact that a process may receive a signal at any time, for instance also when it is updating some sensitive data-structures. If the signal handler will also do something with these same data-structures, you may be in deep trouble. In other words, when you write your program, you must always keep in mind that you may receive a signal exactly at the point where your pencil is.

Linux is compliant with this POSIX 1003.1a definition of signals.

6.2 POSIX 1003.1c signals

From the description above, we have seen that the POSIX 1003.1a signals are a rather complicated business (in UNIX jargon this is called flexibility). The POSIX 1003.1c extensions to the signal mechanism introduces even more flexibility. POSIX 1003.1c really defines an entirely *new set of signals*, which can peacefully co-exist with the old signals of POSIX 003.1a1. The historical name *kill()* is replaced by the more expressive *sigqueue()*.

The main improvements are:
– a far larger number of user-definable signals.
– signals can be queued; old untreated signals are therefore not lost.

- signals are delivered in a fixed order.
- the signal carries an additional integer, which can be used to transmit more information than just the signal number.

POSIX 1003.1c signals can be sent *automatically* as a result of *timer expiration, arrival of a message on an empty queue*, or *by the completion of an asynchronous I/O operation*. Unfortunately, the POSIX 1003.1c signals may not be part of Linux, so we will not dwell on them any further.

6.3 pipes and FIFOs

Probably one of the oldest interprocess communication mechanisms is the **pipe**. Through a pipe, the *standard output* of a program is pumped into the *standard input* of another program. A pipe is usually set up by a shell, when the pipe symbol (|) is typed between the names of two commands. The data flowing through the pipe is lost when the two processes cease to exist. For a **named pipe**, or **FIFO** *(First In, First Out)*, the data remains stored in a file. The *named pipe* has a name in the filesystem and its data can therefore be accessed by any other process in the system, provided it has the necessary permissions.

A running process can set up a pipe to communicate with another process. The communication is uni-directional. If duplex communication is needed, two pipes must be set up: one for each direction of communication. The two "ends of a pipe" are nothing else than file descriptors: one process writes into one of these files, the other reads from the other.

Setting up a pipe between two processes is not a terribly straightforward operation. It starts off by making the *pipe()* system call. This creates *two file descriptors*, if the calling process still has file descriptors available. One of these descriptors (in fact the second one) concerns the end of the pipe where we will write, the other descriptor (the first one) is attached to the opposite end, where we will read from the pipe. If we now create another process, this newly created process will inherit these two file descriptors. We now must make sure that both parent and child processes can find the file descriptors for the pipe ends. The *dup2* system call will in fact do this, by duplicating the "abstract" file descriptors *pipe_ends[0]* and *pipe_ends[1]* into well-known ones. *dup2* copies a file descriptor into the first available one, so we should close first the files where we want the pipe to connect (usually *standard out* for the process connected to the writing end and *standard in* for the process which will read from the pipe). Here is a *skeleton* program for doing this in the case of a *terminal server*, which *forks off a terminal process to display messages from the server*:

```
/* First create a new client */
if (pipe(pipe_ends) < 0) {
    perror("pipe");
    exit(1);
}

global_child=child=fork();
if (child) {
    /*here for parent process*/
    do_something();
}
else {
    /*here for the child*/
    /* pipe ends will be 0 and 1 (stdin and stdout) */
    (void)close(0);
    (void)close(1);
    if (dup2(pipe_ends[0], 0) < 0)
        perror("dup2");
    if (dup2(pipe_ends[1], 1) < 0)
        perror("dup2");
    (void)close(pipe_ends[0]);
    (void)close(pipe_ends[1]);
    execlp(CHILD_PROCESS, CHILD_PROCESS, "/dev/com1", NULL);
    perror("execlp");
    exit(1);
}
```

The terminal process, created as the child could look:

```
#include <fcntl.h>

char buf[MAXBYTES]

/* pipe should not block, to avoid waiting for input */
if(fcntl(channel_from_server, F_SETFL, O_NONBLOCK) < 0){
    perror("fcntl");
    exit(2);
}
while (1) {
    /* Put messages on the screen */
```

```
        /* check for input from the server */
        nbytes = read(channel_from_server, buf, MAXBYTES);
        if (nbytes < 0) && (errno != EAGAIN))
            perror("read");
        else if (nbytes > 0) {
            printf("Message from the Server:  \"%s\"\n", buf);
    }
    ...
```

In this example[11], the server process simply writes to the write end of the pipe (which has become stdout) and the child reads from the other end, which has been transformed by *dup2* into stdin. To set up a communication channel in the other direction as well, the whole process must be repeated, inverting the roles of the server and the terminal client (the first becomes the reader, and the second the writer) and using two other file descriptors (for instance 3 and 4 if they are still free). Note that the *dup* calls must be made before the child does its *exec* call, otherwise, the file descriptors for the two pipe ends would be lost.

The use of named pipes is simpler: the *FIFO* exists in the file system and any process wanting to access the file can just open it. One process should open the *FIFO* for reading, the other for writing. A FIFO is created with the POSIX 1003.1a *mkfifo()* system call.

6.4 Message Queues

When we have compiled the *System V IPC facilities* into the Linux kernel, we have **message queues** available, which however do not conform to the POSIX 1003.1c standard. We will nevertheless describe them briefly, as they are the only ones we have at present.

In system V the **message resource** is described by a *struct msqid_ds*, which is allocated and initialized when the resource is created. It contains the permissions, a pointer to the last and the first message in the queue, the number of messages in the queue, who last sent and who last received a message, etc. The messages itself are contained in:

```
struct msgbuf {
        long mtype;
        char mtext[1]; }
```

[11]Which was also taken from Gallmeister's book.

To set up a message queue, the creator process executes a *msgget* system call:

```
msqid = msgget(key_t key, int msgflg);
```

The *msqid* is a unique identification of the particular message queue which ensures that messages are delivered to the correct destination. The exact role of the key is complicated; in most cases the key can be chosen to be *IPC_PRIVATE*. The use of *IPC_PRIVATE* will create a new message queue if none exist already. If you want to do unusual things or make full use of the built-in *flexibility*, you may fabricate your own key with the *ftok(char* pathname, char proj)* library call and play with *msgflg*.

A process wanting to receive messages on this queue must also perform a *msgget* call, in order to obtain the *msqid*.

A message is sent by executing:

```
int msgsnd(int msqid,struct msgbuf *msgp,int msgsz, \
int msgflg);
```

and similarly a message is received by:

```
int msgrcv(int msqid, struct msgbuf *msgp, int msgsz, \
long msgtyp, int msgflg);
```

msgtyp is used as follows:

$$\text{if msgtyp} \begin{array}{ll} = 0 : & \text{get first message on the queue,} \\ > 0 : & \text{get first message of matching type,} \\ < 0 : & \text{get message with smallest type which is } \leq \text{abs(msgtyp).} \end{array}$$

Finally, the *msgctl* calls allow you to get the status of a queue, modify its size, or destroy the queue entirely.

The message queue can be empty. If a message is sent to an empty queue, the process reading messages from the queue is woken up. Similarly, when the queue is full, a writer trying to send a message will be blocked. As soon as a message is read from the queue, creating space, the writer process is woken up.

6.5 Counting Semaphores

System V **semaphore arrays** are an **oddity**. The *semget* call allocates an **array of counting semaphores**. Presumably, and hopefully, the array may be of length 1. You also specify operations to be performed on a series of members of the array. The operations are only performed if they will **all succeed!**

Counting semaphores can be useful in **producer-consumer problems**, where the producer puts items in a buffer and the consumer takes items away.

Two counting semaphores keep track of the number of items in the buffer and allow to "gracefully" handle the *buffer empty* and *buffer full* situations.

Producer-consumer situations can easily arise in a real-time application: the *producer collects data from measuring devices*, the *consumer writes the data to a storage device* (disk or tape).

Another example is a large paying car park: There is one *counting semaphore* which is initialized to the total number of places in the car park. A *separate process is associated* with *each entrance or exit gate*. The process at an entrance gate will **do a wait on the semaphore, e.g. decrement it**. If the result is greater than zero, the process will continue, issue a ticket with the time of entrance, and open the gate. It closes the gate as soon as it has detected the passage of the car. If the *value of the counting semaphore* is zero when the decrement operation is tried, the process is **blocked** and added to the pile of blocked processes. This is just what is needed: the car park is full and the car will have to wait, so no ticket is issued, etc.

The processes at the exit gates do the contrary: after having checked the ticket, they open the gate and then do a *post or increment* operation on the semaphore, effectively indicating that one more place has become free. This operation will always succeed.

The *System V counting semaphore mechanism* is rather similar to the message queue business: You create a semaphore (array) as follows:
`int semid = semget(key_t key, int nsems, int semflg);`
The key IPC_PRIVATE behaves as before. All processes wanting to use the semaphore must execute this *semget* call. You can then operate on the semaphore:
`int semop(int semid, struct sembuf *sops, unsigned nsops);`
(here is the oddity, you do nsops operations on nsops members of the array; the operations are specified in an array of *struct sembuf*). This structure is defined as:

```
struct sembuf
       ushort sem_num;  /*index in array*/
       short sem_op;    /*operation*/
       short sem_flg    /*operation flags*/
```

Two kinds of operations can result in the process getting blocked:
i) If *sem_op* is 0 and *semval* is non-zero, the process *sleeps* on a queue, waiting for *semval* to become zero, or returns with error *EAGAIN* if either of *(IPC_NOWAIT | sem_flg)* are true.
ii) If *(sem_op* $<$ *0)* and *(semval* $+$ *sem_op* $<$ *0)*, the process either sleeps on a queue waiting for *semval* to increase, or returns with error *EAGAIN* if *(sem_flg & IPC_NOWAIT)* is true.

Atomicity of the semaphore operations is guaranteed, because the mechanism is embedded in the kernel. The kernel will not allow two processes to simultaneously use the kernel services. In other words, a system call will be entirely finished before a context switch takes place.

Note: If you want to use a semaphore which takes only the values 0 or 1 (for instance for mutual exclusion), you are better off by using the atomic bit operations, defined in <asm-i386/bitops.h>: *test_bit, set_bit and clear_bit.*

6.6 Shared Memory

Shared Memory is exactly what its name says: two or more processes **access the same area of physical memory**. This segment of physical memory is **mapped into** two or more **virtual memory spaces**.

Shared Memory is considered a low-level facility, because the shared segment **does not benefit from the protection** the operating system normally provides. To compensate for this disadvantage, **shared memory is the fastest IPC mechanism.** The processes can read and write shared memory, without *any system call being necessary.* The *user himself must provide the necessary protection*, to avoid that two processes "simultaneously" access the shared memory. This can be obtained with a **binary semaphore** or **mutex**.

A *mutex* can be simulated by performing the *set_bit(int nr, void * addr)* call, which sets the desired bit *nr* and returns the old value of the bit. The short integer on which this operation is performed must also reside in shared memory, in order to be accessible by both processes.

The shared memory facility available in Linux comes from System V, and is may therefore be **not** conforming to POSIX.1c. The related system calls are similar to the System V calls we have already seen:

There is, of course, a *shared memory descriptor*, `struct shmid_ds`.
Shared memory is allocated with the system call:
`shmid = shmget(key_t key, int size, int shmflg);`
The *size* is in bytes and should preferably correspond to a multiple of the page size (4096 bytes). All processes wanting to make use of the shared memory segment must make a *shmget* call, with the same key.
Once the memory has been allocated, you map it into the virtual memory space of your process with:

```
char *virt_addr;
virt_addr = shmat(int shmid, char *shmaddr, int shmflg);
```

shmaddr is the requested attach address:
if it is 0, the system finds an unmapped region;

if it is non-zero, then the value must be *page-aligned.*
By setting *shmflg = SHM_RDONLY* you can request to attach the segment *read-only.*

You can get rid of a shared memory segment by:
`int shmdt(char *virt_addr);`
Finally, there is again the *shmctl* call, which you may use to get the status, or also to destroy the segment (a shared segment will only be destroyed after all users have *detached* themselves).

If you are using shared memory, and you need *malloc* as well, you should `malloc` a large chunk of memory first, before you attach the shared memory segment. Otherwise *malloc* may interfere with the shared memory.

A word about the Linux implementation of the System V IPC mechanisms is in order. All System V system calls described above make use of a single Linux system call: ipc(). A library of the system V IPC calls is available, which maps each call and its parameters into the Linux ipc() call. An example is:

```
int semget (key_t key, int nsems, int semflg)
{
    return ipc (SEMGET, key, nsems, semflg, NULL);
}
```

The constants are defined in <linux/ipc.h>

7 Scheduling

The original scheduling algorithm of Linux aimed at giving a **fair share of the resources** to each user. It therefore was a typical **time-sharing scheduler**. A time-sharing scheduler is based on priorities, like any other type of scheduler, but the system keeps changing the priorities to attain its aim of being fair to everyone.

For *time-critical real-time applications* you want another sort of scheduler. You need a **high priority for the most critical real-time processes**, and a *scheduler* which will run such a high priority process whenever **no process with higher priority is runnable**[12].

Less critical processes of the real-time application can run at *lower priorities* and other user jobs could also be fitted in at priorities below.

[12]Remember that you need a sleeping shell at a still higher priority.

SVR6 (System V, Release 6) has a scheduler that does both time-sharing and real-time scheduling, depending on the priority assigned to a process. Critical processes run at priorities between, say, 0 and 50, and benefit from the priority scheduling. Other jobs run at lower priorities and have to accept the time-sharing scheduler. This aspect of System V has not been ported to Linux.

A POSIX.1c compliant scheduler **has** been ported to Linux. In order to make use of it, you must make *patches to the kernel code* and recompile the kernel together with this POSIX.1c scheduler. At the time these notes were prepared, we had not yet had a chance to try it.

The advantage of a POSIX.1c scheduler is, of course, that your application program will be portable between different platforms.

What does a POSIX.1c scheduler do? Here is what it provides[13]:

```
#include         <unistd.h>
#ifdef _POSIX_PRIORITY_SCHEDULING
#include         <sched.h>
int i, policy;
struct sched_param *scheduling_parameters;
pid_t pid;

sched_setscheduler(pid_t pid, int policy, \
   struct sched_param *scheduling_parameters);
int sched_getscheduler(pid_t pid);
int sched_getparam(pid_t pid, \
   struct sched_param *scheduling_parameters);
int sched_setparam(pid_t pid, \
   struct sched_param *scheduling_parameters);
int sched_yield(void);
int sched_get_priority_min(int);
int sched_get_priority_max(int);
#endif _POSIX_PRIORITY_SCHEDULING
```

You see that you define a *scheduling "policy"*. You have a choice:

SCHED_FIFO:	pre-emptive, priority-based scheduling,
SCHED_RR:	pre-emptive, priority-based with *time quanta*,
SCHED_OTHER:	implementation dependent scheduler.

With the first choice, the process *will run* until it gets blocked for one reason or another, or *until a higher priority process becomes runnable.* The

[13]Again from Gallmeister's book.

second policy adds a **time quantum**: a process running under this scheduling policy will only run for a certain duration of time. Then it goes back to the end of the queue for its priority (each priority level has its own queue). Thus, at a given priority level, all processes in that level are scheduled **round-robin**. In future, **deadline scheduling** will probably have to be added as another choice.

There is a range of priorities for the *FIFO* scheduler and another range for the *RR* scheduler.

After a *fork()*, the child process *inherits the scheduling policy and the priority* of the parent process. If the priority of the child then gets increased above the priority of the running process, the latter is *immediately pre-empted*, even before the return from the *sched_setparam* call! So be careful, you may seriously harm yourself.

On the other hand, you may "*yield*" the processor to another process. You cannot really be sure which process this is going to be. As a matter of fact, the only thing *yield* does, is to put your process at the end of the queue at your particular priority level.

All this is nice, but we are still *stuck with the fact that the kernel itself cannot be pre-empted.* This is usually not too much of a problem. Most of the system calls will take only a short time to execute.

Usually, the system calls that may take a considerable time (such as certain I/O related calls), should be relegated — as far as possible — to those tasks that run at a lower priority level. Also some common sense will help: it is much faster to write once 512 bytes to disk than to write 512 times a single byte!

Other system calls do take a long time. *fork* and *exec* for example. You should therefore **create** all necessary processes during the **initialization phase** of your application. Let the processes that you only need sporadically just *sleep* for most of the day.

8 Timers

You may want to arrange for certain things to happen at **certain times**, or a given **time interval after** something else happened. So you will nearly always have the need for a **timer** and/or an **interval timer**.

Standard UNIX (and Linux) has a **real-time clock**. It counts the number of seconds since 00:00 a.m. January 1, 1970. (called the *Epoch*). You get its value with the *time()* function:

```
#include <time.h>
```

```
time_t time(time_t *the_time_now);
```

You can also call *time* with a NULL pointer.

Linux also has the *gettimeofday* call, which stores the time in a structure:

```
struct timeval {
    time_t tv_sec    /* seconds */
    time_t tv_usec }     /* microseconds */
```

`gettimeofday` returns a 0 or -1 (success, failure respectively).

You can make things happen **after a certain time interval** with *sleep*:
`unsigned int sleep(unsigned int n_seconds);`

The process which executes this call will be stopped and resumed *after n_seconds* have passed. The resolution is very crude! As a matter of fact, many real-time systems would need a resolution of milliseconds and, in extreme cases, even microseconds.

To overcome this drawback, Linux has also **interval timers**. each process has three of them:

```
#include       <sys/time.h>

int setitimer(int which_timer,
    const struct itimerval *new_itimer_value, \
    struct itimerval *old_itimer_value);
int getitimer(int which_timer, \
    struct itimerval *current_itimer_value);
```

The first argument, *which_timer*, has one of three values: *ITIMER_REAL*, *ITIMER_VIRTUAL* and *ITIMER_PROF*. *setitimer()* sets a new value of the interval timer and returns the old value in *old_timer_value*. When a timer **expires, it delivers a signal**: *SIGALRM*, *SIGVTALRM* and *SIGPROF* respectively. The calls make use of a structure:

```
struct itimerval {
    struct timeval it_val      /* initial value */
    struct timeval it_interval }     /* interval */
```

The *ITIMER_REAL* measures the time on the "wall clock" and therefore includes the time used by other processes. *ITIMER_VIRTUAL* measures the time spent in the user process which set up the timer, whereas *ITIMER_PROF* counts the time spent in the user process **and** in the kernel on behalf of the user process. It is thus very useful for *profiling*.

The resolution of these interval timers is given by the constant *HZ*, defined in *<sys/param.h>*. On Linux machines, *HZ=100*, so the resolution of the interval timers is 10000 microseconds.

POSIX.1c extends the timer facilities to a number of implementation defined clocks, which may have different characteristics. Timers and intervals can be specified in nanoseconds.

9 Memory Locking

As we already pointed out before, the real-time processes – at least the critical ones – should be **locked into memory**. Otherwise you could have the very unfortunate situation that your essential task has been swapped out, just before it becomes runnable again. **Faulting** a number of pages of code back into memory may add an intolerable overhead.

Remember also that *infrequently used pages may be swapped out* by the system, without any warning. Faulting them back in again may make you miss a **deadline**. Thus, not only the *program code*, but also the *data and stack pages* should be locked into memory.

A POSIX.1c conformant memory locking mechanism is available for Linux. Unfortunately, we have not yet been able to test it. It does the following:

```
#include        <unistd.h>
#ifdef _POSIX_MEMLOCK
#include        <sys/mman.h>
int mlockall(int flags);
int munlockall(void);
#endif /* _POSIX_MEMLOCK */
```

mlockall will lock **all** your memory, e.g. *program, data, heap, stack and also shared libraries.* You may choose, by specifying the flags, to lock the space you occupy at present, but also what you will occupy in future.

Instead of locking everything, you may also lock parts:

```
#include        <unistd.h>
#ifdef _POSIX_MEMLOCK_RANGE
#include        <sys/mman.h>

int mlock(void *address, size_t length);
int munlock(void *address, size_t length);
#endif /* _POSIX_MEMLOCK_RANGE */
```

Finally, you may want to lock just a few essential functions: a signal handler or an interrupt handler, for instance. You should not do this from within the interrupt handler, but from a separate function:

```
void intr_handler()
{
    ...  /* do your work here */
}
void right_after_intr_handler()
{
        /* this function serves to get an address */
        /* associated with the end of intr_handler() */
}

void intr_handler_init()
{
    ...
    i = mlock(ROUND_DOWNTO_PAGE(intr_handler),\
      ROUND_UPTO_PAGE(right_after_intr_handler - \
      intr_handler));
}
```

The function *right_after_intr_handler()* does nothing. It serves only to get an address associated with the *end* of the interrupt handler. This is needed to calculate the argument *length* for the *mlock()* call.

10 Multiple User Threads

All we have seen so far happened at the *process* level and *kernel intervention* was needed for every coordinating action between processes. The overall picture has become quite complicated and a programmer must master many details or else he runs into trouble.

Is there not another solution, where the user has more direct control over what is going on? Fortunately, there is: **multiple user threads**. POSIX.4a (or POSIX.1c if you prefer) standardizes the **API (Application Programmer's Interface)** for multiple threads.

Threads are independent flows of control *inside a single process.* Each *thread* has its *own thread structure* — comparable to a *process descriptor* —, its *own stack* and its *own program counter.* All the rest, i.e. *program code, heap storage* and *global data*, is **shared** between the threads. Two or more threads may well execute the same function simultaneously. The services needed to *create threads, schedule their execution, communicate and synchronize between threads* are provided by the **threads library** and *run in user space.* For the kernel exists only the *process*; what happens inside this process is invisible to the kernel.

Lightweight Processes, as in Solaris or SunOS 4.x, are somewhere midway: a small part of the process structure has been split off and can be replicated for several LWPs, all continuing to be part of the same process, using the same memory map, file descriptors, etc. The split-off part is still a kernel structure, but the kernel can now make rapid context switches between LWPs, because only a small part of the complete process structure is affected. Inside a LWP, multiple threads may be present.

Multiple threads offer a solution to programming which has a number of advantages. The model is particularly well suited to *Shared-memory Multiple Processors*, where the code, common to all threads, is executed on different processors, one or more threads per processor. Also for real-time applications on uniprocessors, threads have advantages. In the first place, the *fastest, easiest intertask communication mechanism, — shared memory —* is there for *free*!

There are other advantages as well. The **responsiveness** of the process may increase, because when one thread is *blocked*, waiting for an event, the other can continue execution. The fact that threads offer a sort of "do-it-yourself" solution makes the user have a better grasp of what he is doing and thus he can produce better structured programs. *Communication* and *synchronization* between threads is easier, more transparent and faster than between processes. Each thread conserves its ability to communicate with another process, but it is wise to concentrate all *inter-process communication* within a single thread.

Multiple threads will in general lead to performance improvements on shared memory multiprocessors, but on a uniprocessor one should not expect miracles. Nevertheless, the fact that there is **less overhead** and that some threads may **block while others continue**, will be felt in the **performance**.

It sounds as if we just discovered a gold mine. Well ..., there are a few things which obscure the picture somewhat. For threads to be usable with no danger, the **library functions** our program uses must be **threads-safe**. That is, they must be *re-entrant*. Unfortunately, many libraries contain functions which modify global variables and therefore are **not** re-entrant. For the same reason, your threaded program must be re-entrant, so it has to be compiled with *_REENTRANT* defined. In addition, for a real-time application, you still need at least a few facilities from the operating system:

memory locking and real-time priority scheduling[14].

Threads can be implemented as a *library of user functions.* The standard set of functions is defined in POSIX.1c, but other implementations also exist. The package we are using[15] implements the **POSIX.1c pthreads**. There are some 50 service requests defined. They are briefly described in Annex III and in more detail in the man pages. We will illustrate only a few of them, the most important ones.

pthreads defines functions for *Thread Management, Mutexes, Condition Variables, Signal Management, Thread Specific Data* and *Thread Cancellation.* Threads, mutexes and condition variables have *attributes*, which can be modified and which will change their behaviour. Not all options defined by the various attributes need to be implemented. *<pthread.h>* defines eight data types:

Type	**Description**
pthread_attr_t	`Thread attribute`
pthread_mutexattr_t	`Mutex attribute`
pthread_condattr_t	`Condition variable attribute`
pthread_mutex_t	`Mutual exclusion lock (mutex)`
pthread_cond_t	`Condition variable`
pthread_t	`Thread ID`
pthread_once_t	`Once-only execution`
pthread_key_t	`Thread specific data key`

Attributes can be *set or retrieved* with calls of the following type:

```
int pthread_attr_setschedpolicy( pthread_attr_t *attr, \
int newvalue);
```
or:
```
int pthread_mutexattr_getprotocol( pthread_mutexattr_t *attr, \
          *protocol);
```

See Annex III for the complete list. The *scheduling policy* can be one of: `SCHED_FIFO`, `SCHED_RR` and `SCHED_OTHER`, as for the POSIX.1c standard. The scheduling parameters can also be set and retrieved.

When the process is forked, *main(argc, argv)* is entered. In the main program you may then create threads. Each thread is a function, or a sequence of functions. At thread creation, the *entry point* must be specified:

[14]Alternatively, you run on a dedicated machine, where you have killed all daemons, so that your application is the only active process in the system.

[15]Xavier Leroy's implementation, called **LinuxThreads**, which is part of many recent Linux distributions (Xavier.Leroy@inria.fr).

```
int pthread_create( pthread_t *thread, \
          const pthread_attr_t *attr, void *(*entry)(void *), \
          void *arg );
```

```
   void pthread_exit( void *status );
```

does what is expected from it. It should be noted that *NULL* may often be used to substitute an argument in the function call. This is notably the case for `pthread_attr_t *attr` and `void *status` above.

An important function is:

```
int pthread_join( pthread_t thread, void **status );
```

When this primitive is called by the running thread, its execution will be suspended until the target `thread` terminates. If it has already terminated, execution of the calling thread continues. *pthread_join()* is therefore an important mechanism for synchronizing between threads. So-called *detached threads* cannot be joined. You specify at creation time or at run time if the thread has to be detached or not.

Mutexes can have as the *pshared* attribute `PTHREAD_PROCESS_SHARED` or `PTHREAD_PROCESS_PRIVATE`, meaning that the mutex can be accessed also by other processes or that it is private to our process. Private mutexes are defined in all implementations, shared mutexes are an option. The two usual operations on a mutex are:

```
int pthread_mutex_lock( pthread_mutex_t *mutex );
```

and

```
int pthread_mutex_unlock( pthread_mutex_t *mutex );
```

but you can also try if a mutex is locked and continue execution, whatever the result:

```
int pthread_mutex_trylock( pthread_mutex_t *mutex );
```

All memory occupied by the process is shared among the various threads, which we said was an important advantage of threads. Nevertheless, sometimes a thread needs to protect its data against attacks from other threads. For this reason a few primitives which allow to create and manipulate *thread specific data* are defined. For details see the man pages.

We have not yet met **condition variables**, which are another feature of pthreads. Condition variables are always associated with a *mutex*. A condition variable is used to *signal* a thread that a particular condition has been satisfied in another thread. The first thread — the one receiving the signal — will then be allowed to proceed if it had blocked on the condition variable (CV). It works as follows:

Thread 1	**Thread 2**
lock the mutex	
test the condition	
FALSE! *unlock mutex*	
sleep on CV	
	lock the mutex
	change the condition
	signal thread 1
	unlock mutex
lock mutex	
test condition again	
TRUE! *do the job*	
unlock mutex	

Translated into code, this becomes:

Thread 1 **Thread 2**

```
pthread_mutex_lock(&m);
while (!my_condition) {
while (pthread_cond_wait(&c, &m) != 0) { ;
                                          pthread_mutex_lock(&m);
                                          my_condition = TRUE ;
                                          pthread_cond_signal(&c);
                                          pthread_mutex_unlock(&m);
do_thing();
}
}
pthread_mutex_unlock(&m);
```

Note that *pthread_cond_wait()* will automatically free the mutex for you and your thread will go to sleep on the condition variable.

pthreads is really a subject in itself and our quick review has been very superficial. Threads are well suited for implementing **Server-Client problems**. Due to the shared memory, the communication between the *server* and the — possibly many — *clients* is easy.

We close this section with a complete code example[16]. The example concerns an *Automatic Teller Machine*, e.g. one of those machines that distribute

[16]This and the following example are from *B. Nichols et.al.*, *Pthreads Programming* See Bibliography, item ii)

banknotes. The main program, which is the **server**, receives requests over a communication line from ATMs scattered all over town. For each request received, the server spawns a *worker* or *client thread* which undertakes the actions necessary to satisfy the request. This example mainly illustrates the creation of several threads.

```
typedef struct workorder {
        int conn;
char req_buf[COMM_BUF_SIZE];
} workorder_t;

main(int argc, char **argv)
{
  workorder_t *workorderp;
  pthread_t   *worker_threadp;
  int  conn, trans_id;

  atm_server_init(argc, argv);

  for(;;) {
    /*** Wait for a request ***/
    workorderp = (workorder_t *)malloc(sizeof(workorder_t));
    server_comm_get_request(&workorderp->conn,
            &workorderp->req_buf);

    sscanf(workorderp->req_buf, "%d", &trans_id);
    if (trans_id == SHUTDOWN) {
          . . .
      break;
    }

    /*** Spawn a thread to process this request ***/
    worker_threadp = (pthread_t *)malloc(sizeof(pthread_t));
    pthread_create(worker_threadp, NULL, process_request,
          (void *)workorderp);

    pthread_detach(*worker_threadp);
    free(worker_threadp);
  }
  server_comm_shutdown();
}
```

The *worker thread* (the **client**) looks as follows:

```
void process_request(workorder_t *workorderp)
{
  char resp_buf[COMM_BUF_SIZE];
  int trans_id;
  sscanf(workorderp->req_buf, "%d", &trans_id);

  switch(trans_id) {
    case WITHDRAW_TRANS:
         withdraw(workorderp->req_buf, resp_buf);
 break;

    case BALANCE_TRANS:
         balance(workorderp->req_buf, resp_buf);
 break;
  .
  .
    default:
         handle_bad_trans_id(workorderp->req_buf, resp_buf);
  }

  server_comm_send_response(workorderp->conn, resp_buf);
  free(workorderp);
}
```

There are two points to note in this example. The first concerns the passing of arguments to a child thread. The standard allows a single argument only. Encapsulating several arguments in a single structure and passing a pointer to this structure to the child is a way to program around the restriction. The second point is a subtle one and concerns the use of *malloc*. Using static storage for the workorder does not work: for every newly created thread the workorder would be overwritten and most threads would work with a corrupted workorder.

The following example, taken from the same source, illustrates the use of a *mutex* and a *condition variable*. Two of the threads created in this example simply increment a counter and check if it has reached a limit value. In that case they *signal the condition variable*. The third thread waits on the condition variable and prints its value. The main thread will exit when all three threads it created have "*joined*".

```
#include <pthread.h>
#define TCOUNT 10
#define WATCH_COUNT 12

int count = 0;
pthread_mutex_t count_mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t count_threshold_cv = PTHREAD_COND_INITIALIZER;
int thread_ids[3] = {0, 1, 2};
main()
{
  pthread_t threads[3];

  pthread_create((&threads[0], NULL, inc_count, &thread_ids[0]);
  pthread_create((&threads[1], NULL, inc_count, &thread_ids[1]);
  pthread_create((&threads[2], NULL, watch_count, &thread_ids[2]);
  for(i = 0; i < 3; i++)
    pthread_join((&threads[i], NULL);
}

void watch_count(int *idp)
{
  pthread_mutex_lock(&count_mutex);
  while(count <= WATCH_COUNT) {
    pthread_cond_wait(&count_threshold_cv, &count_mutex);
    printf("watch: Thread %d, Count is %d\n", *idp, count);
  }
  pthread_mutex_unlock(&count_mutex);
}

void inc_count(int *idp)
{
  int i;
  for(i = 0; i < TCOUNT;, i++)
    pthread_mutex_lock(&count_mutex);
    count++;
    printf("inc: Thread %d, count is %d\n", *idp, count);
    if(count == WATCH_COUNT)
      pthread_cond_signal(&count_threshold_cv);
    pthread_mutex_unlock(&count_mutex);
  }
}
```

In this example the reader should note that two threads share identical code and that only one copy of this code is present in memory. The program counter and the stack are of course private property of each individual thread.

A brief resume of the POSIX 1003.1c definitions is given in Annex III. For more details, the reader is referred to the "man pages".

11 Real Time Linux

In October 1996, we learned that a **Real-time Linux** had been developed at the *New Mexico Institute of Technology* and that a *beta-version* was available for testing. Since then Barabanov et. al. have released updated versions of **RTLinux**, and during this College participants can again experiment with it.

It is based on a *different principle* from what we have described so far: it uses the concept of a **virtual machine**. RTLinux embodies a small, **real-time executive**, and standard Linux runs underneath it, as a *low priority task.* The *time critical* parts of the application run directly under `RTLinux` and are scheduled by RTLinux itself. "Classical" Linux is run only when there is *no real-time task ready to run.*

The real-time executive *intercepts the interrupts* and therefore it *can react fast.* Interrupts which have nothing to do with the real-time tasks are passed down to Linux. When Linux *disables interrupts* (with the cli() call), `RTLinux` will stop passing interrupts to Linux. But those interrupts *remain available* to RTLinux for a later time. Linux is used for the lower priority and slower tasks, such as file manipulation. But still all facilities of normal Linux are available.

Communication between a *real-time task* and an *ordinary Linux process* is done via a *special IO interface*, called a **real-time fifo**.

A real-time application should be split into *small and simple parts*, which have *real-time constraints* on the one and larger pieces for more *complex processing* on the other hand.

The real-time component is written as a *dynamically loadable Linux kernel module.* A complete example [17] follows. In this example the real-time part reads periodically data from an external device and puts it into a real-time fifo. The Linux process reads the data from the fifo and can process it. In the example, the data is simply written to *stdout.*

```
#define MODULE
```

[17] The example is taken from: M. Barabanov and V. Yodaiken, *Introducing Real-Time Linux*, Linux Journal, February 1997, page 19-23.

```
#include <linux/module.h>

/* always needed for real-time task */
#include <linux/rt_sched.h>
#include <linux/rt_fifo.h>
RT_TASK mytask;

/* This is the main program */
void mainloop(int fifodesc)
{
int data;

/* in this loop we obtain data from the */
/* device and put it into fifo number 1 */
while (1) {
    data = get_data();
    rt_fifo_put(fifodesc, (char*)&data, sizeof(data));
    /* give up the CPU until next period */
    rt_task_wait();
    }
}

/* This function is needed in any module */
/* It will be invoked when the module is loaded */
int init_module(void)
{
  #define RTfifoDESC 1
  RTIME now = rt_get_time();

  /* 'rt_task_init' associates a function */
  /* with the RT_TASK structure and sets parameters: */
  /* Priority=4, stack size=3000 bytes, pass 1 to */
  /*'mainloop' as an argument */

  rt_task_init(&mytask, mainloop, RTfifoDESC, 3000, 4);

  /* Mark 'mytask' as periodic */
  /* It could be interrupt driven as well */
  /* Period is 25000 time units. It starts */
  /* 1000 time units from now */
  rt_task_make_periodic(&mytask, now+1000, 25000);
  return 0;
}

/* Clean-up routine. It is called when the */
/* module is unloaded */
void clean_up(void)
{
  /* kill the real-time task */
  rt_task_delete(&mytask);
  return;
}
```

The ordinary Linux process executes the following program:

```
#include <rt_fifo.h>
#include <stdio.h>

#define RTfifoDESC 1
#define BUFSIZE 10
int buf[BUFSIZE];

int main()
{
  int i, n;
  /* create fifo number 1, size 1000 bytes */
  rt_fifo_create(1, 1000);
  for (n=0; n<1000; n++) {
    /* read data from fifo and print it */
    rt_fifo_read(1, (char*)buf, BUFSIZE * sizeof(int));
    for (i=0; i<BUFSIZE; i++) {
      printf("%d ", buf[i]);
    }
    printf("\n");
  }
  /* destroy fifo number 1 */
  rt_fifo_destroy(1);
  return 0;
}
```

The latest version of RTLinux can be obtained from http://luz.nmt.edu/r̃tlinux and besides the executive, it contains kernel patches, documentation, examples and installation tips. Recently, **shared memory** has been added as a means of communicating between RTLinux and standard Linux.

In order to build RTLinux, the Linux kernel must be recompiled. A user can then run either RTLinux or normal Linux at his choice, if `lilo.conf` is adjusted accordingly.

RTLinux is used at various Institutes around the world. At the Humboldt University in Berlin it is used for data acquisition with an ADC. On a 33 MHz 486 machine, a rate of 3000 samples/s is achieved, using a cheap ADC board connected to the PC via a serial line. At the Universidad Politecnica di Valencia, it is used to develop *earliest deadline first schedules*, including a comprehensive graphical display. Also NASA is using it, but its web sites are not accessible, and we were unable to obtain more information. The New Mexico Institute of Technology itself uses RTLinux in a teaching environment and in the Sunrayce Project (an embedded control system for a solar car?).

The authors of RTLinux say that they could run a repetitive task at a rate of **once every 150 μseconds** on a *133 MHz Pentium*. Tasks can be scheduled within a precision of 10 μseconds.

12 RTAI

Paolo Mantegazza and his collaborators at the *Dipartimento di Ingegneria Aerospaziale* of the *Politecnico di Milano* have done a lot of work to improve RTLinux and make it more versatile and user-friendly. The result is a completely new package, **RTAI**, the **Real Time Application Interface**. The package is available from www.aero.polimi.it/projects/rtai. It comes with a large amount of documentation, including explanations of the internals of RTAI, detailed installation procedures, a sort of tutorial and several example programs.

RTAI implements a **hard realtime** system that coexists with Linux. It can run periodic tasks with a frequency exceeding 10 kHz, with a jitter of ± 5 μsec. It can also run in *one-shot* mode.

RTAI has a number of interesting features. It can run on a single processor machine or on a *Symmetric Multi Processor* PC. In the case of SMP, RTAI can be confined to a subset of the processors, or even to a single one. One can also choose to have the realtime interrupts handled by one specific processor, without affecting the interrupts intended for Linux. Pentium or better processors are prefered, but RTAI can run quite reasonably on a 486 machine. To get the most out of it an APIC timer should be available in the processor, besides the usual 8254 timer.

The scheduler functions are also available for the normal Linux processes, which means that you have at your disposal an **uniform API** for all your applications, be they hard, firm or soft realtime. You may use *messages, semaphores*, *shared memory* and *time intervals* for communication from *Linux to Linux*, *RTAI to RTAI* and also between *Linux and RTAI.*

When you build RTAI and install it, the realtime application will run in **kernel mode**. You can also make your application run in **user space**, without the need of being the superuser, once the superuser has installed the necessary *kernel modules* for you.

RTAI in fact presents itself in the form of **kernel modules**. Three are required for a basic configuration (comparable to the one provided by RTLinux): the rtai module (rtai), the scheduler (rtl_sched) and the fifo module (rtl_fifo). The application program is added as a fourth module, for instance: rt_process. The latter must be written by the user. A simple example will be shown below. The **fifo** (*First In First Out* buffer) model is the same as for RTLinux: the realtime task or tasks write to a *fifo* and read from another *fifo*. Processes on the Linux side see these *fifos* as normal *character devices.*

Linux maintains all its features and can thus be used to post-process the acquired data, display the data and archive them, in case we are speaking of a data acquisition system. Likewise, Linux can do also the necessary calculations in the case of a control system. The authors assure that it is possible to run a remote data acquisition system at a rate of one sample every 100 μsec, complete with the network, X11 for displaying results, etc, without Linux falling flat. The authors also suggest that you may use the *interrupt trapping mechanism* on its own, without the scheduler and the rest of the RTAI machinery. This would give you much closer control over the bare hardware, which could be useful for writing a *driver module.*

The following example[18] shows a simple data acquisition application, running in periodic mode at a 10 kHz sampling rate:

```
#define MODULE
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/version.h>
#include <asm/io.h>
#include <rtai.h>
#include <rtl_sched.h>
#include <math.h>
#include "acquisition_lib.h"
```

[18]taken from the *"Beginners Guide"* in the RTAI package.

```
#define STACK_SIZE 2000
#define LOOPS 1000000000

static RT_TASK acquisition_task;

/* This is the function that performs data-acquisition by reading from the
   specific board at a frequency of 10000 Hz */

static void fun_acquis(int t)
{
  unsigned int loops = LOOPS;
  while(loops--){
    read_adc();
    rt_task_wait_period();
  }
}

int init_module(void)
{
  RTIME now, tick_period;
  int period = 100000;
  rt_set_periodic_mode();      /* The periodic mode is set because
                                  we have only one task with a fixed period */
  tick_period = start_rt_timer((int)nano2count(period));
                               /* Conversion of timer period from
                                  nanoseconds to internal count units */
  rt_task_init(&acquisition_task,fun_acquis,period,STACK_SIZE, 1, 1, 0);
  now = rt_get_time();
  rt_task_make_periodic(&acquisition_task,now + tick_period,tick_period);
  return 0;
}

void cleanup_module(void)
{
  stop_rt_timer();
  rt_task_delete(&acquisition_task);
}
```

The reader should note that the routine read_adc() has to be written to satisfy the requirements of the specific acquisition board used.

13 KURT, the Kansas University Real Time Linux

Very recently, Balaji Shrinavasan of the Information and Telecommunications Technolgy Center (ITTC) of the University of Kansas announced another version of real-time Linux.

It is called **KURT**, for *Kansas University Real Time* Linux.

The author calls it a *firm* real-time system, somewhere between a *hard* and a *soft* real-time system. It is based on a different principle from *RTLinux*.

KURT allows the *explicit scheduling* of real-time **events**, instead of just *processes*. The event scheduling is done by the system.

Once KURT has been installed, Linux has acquired a second mode of operation. The two modes are: *normal-mode* and *real-time mode*. In the first the system behaves as normal Linux, but when the kernel is running in *real-time mode*, it executes only *real-time processes*. All system resources are then dedicated to the real-time tasks. There is a *system call* that toggles between the two modes.

During the *setup phase* the schedule of events to be executed in *real-time mode* is established and the processes that must run in this mode are marked. The kernel is then switched to the *real-time mode*. When all tasks have finished, the kernel is switched back to *normal-mode*.

In order to obtain this behaviour, KURT consists of a **Real-Time Framework** which takes care of scheduling any *real-time event*. When such an event is to be executed, the *real-time framework* calls the **event handler** of the associated **RTMod** (Real-Time Module). The *RTMods* can be very simple; calling them according to a defined schedule is the responsibility of the *real-time framework*. This framework provides the system calls that switch the kernel between the two modes.

The *RTMods* are *kernel modules*, which are loaded at runtime. An *RT-Mod* registers itself with the *real-time framework*. It then provides pointers to functions for the *event handler*, *initialization* and *clean-up*.

When a *RTMod* must be invoked is defined in the **Real-Time Schedule**, which is just a **file**. This file can be built beforehand. It can be copied entirely into memory, or it can remain on disk. In the latter case, the timing of events may become distorted by disk access times.

In addition to the *Real-Time Schedule*, *processes* can be run *periodically* in a round-robin fashion.

Events can be scheduled with a high time resolution, when another package has been installed: **UTIME**, for *μsecond time*. This package was developed at the same Institute as KURT. If it is not installed then the time resolution is only the usual 10 ms.

To install *UTIME* and/or *KURT*, the Linux kernel must be recompiled.

A more detailed description and *sample programs* are available. The reader should look for them in the directory /usr/local/tarfiles.

The packages together with the necessary Kernel patches can be obtained by anonymous ftp from the WEB page:
`http://hegel.ittc.ukans.edu/projects/kurt` for *KURT* and
`http://hegel.ittc.ukans.edu/projects/utime` for *UTIME*.
`http://hegel.ittc.ukans.edu/projects/posix` has extensions to the Linux kernel for better POSIX 1003.1c compatibility.

14 Embedded Linux

With portable telephones, handheld and palmtop computers, wireless connections to Internet, automated home appliances and what not, there is a need for **embedded operating systems**. A year or so ago efforts have started to develop these. Several manufacturers have turned to Linux, having recognised that the availability of **open software** bears many advantages, not only to individual developers, hackers and other maniacs, but also for industry. Linux Journal has a regular review on this topic and the September 2000 issue concentrates on the topic. Just to show how active this field is, I cite a few examples from this issue.

- The first example concerns a network of *home infromation appliances*, with a central server and wireless connections to clients scattered through the home. The server obviously runs Linux, but also the clients have an embedded Linux system with a reduced (*small footprint*) X11. Hooking a keyboard, a monitor and a mouse to such a client box, you use it as a PC. Hooking high-fi speakers to it, you listen to music, etc. All the work is done on the server, and data is passed over the network to and from the clients. (http://www.adomo.com).

- *Yopi* is a handheld computer, $12.5 \times 7.5\ cm^2$ with a color display and a 206 MHz StrongArm CPU with 32 Mbyte DRAM. Another 32 Mbyte of flash ROM stores the Linux system and the core set of applications. The object has no keyboard and looks like a gameboy.

- There is a commercial realtime system based on Linux on the market: *Linux/RT* from *TimeSys Corporation* (email: info@timesys.com). Linux/RT is based on the Carnegie Mellon University *Linux Resource Kernel, Linux/RK.* There is support for *Robust Embedded (RED)* Linux system event logging. Linux/RT also contains RTAI. You run one or the other at any one time. The product may not yet be very mature.

- Aplio bets on voice transmission over Internet (*VoIP*). Their boxes plug into a telephone as simply as an answering machine and the other side of it plugs into an Ethernet port. The boxes run an embedded Linux kernel. It is really based on μClinux, a Linux version for microcontrollers without a memory management unit.
- The firm that produced **LynxOS** (see section 2) changed name. It now calls itself **LynuxWorks**. They released *BlueCat*, a version of Linux tailored for embedded applications. There LynxOS real time operating system will become binary compatible with Linux, so that any executable that runs on Linux can also run on LynxOS.
- Compaq produces *iPaq*, a handheld computer running Linux. The project to develop iPaq originated at Digital Equipment Corporation, which has been bought by Compaq.
- RedHat, in collaboration with Cygnus is working on the development of EL/IX, a Linux-based operating system for embedded applications.
- Lineo is another firm working on Linux for embedded applications.
- To my shame, I don't know where μClinux comes from and what it can do.

15 Conclusion

We have tried in this course to give a brief overview of the requirements of a real-time application and we have investigated to what extent Linux can do the job. We have also mentioned the improvements to Linux which have already been made. Among these improvements, `RTLinux` and `KURT` are of the greatest importance for the development of real-time applications. Recent developments in this direction include RTAI and the various projects for *embedded Linux*.

Also the **pthreads** package does contain a major part of the improvements a user would like to see. When a real-time application has been written using *pthreads*, the only essential features the operating system has still to provide are *memory locking* and *real-time scheduling*.

The important thing to remember is that you should analyze your problem very carefully, before deciding that you can (or cannot) use such or such an operating system. Hopefully, this course has shown you the points to consider, and where to search for existing and acceptable solutions.

All depends therefore on your application. If you expect **high data rates** or **high interrupt rates** or if you are otherwise pressed for time constraints, or if you must meet stringent deadlines, then you will need many of the mechanisms described and you should have resort to `RTLinux` or `KURT` or else you may have to accept acquiring a true real-time operating system.

This can be the case in physics experiments, in particular in Particle Physics and in Nuclear Physics.

There will however be situations where you don't need the heavy guns and where the standard Linux system will do the job. To give you an idea: Ulrich Raich runs a real-time application on a 66 MHz 486 machine, concurrently with X11. The machine sustains a rate of 200 external interrupts per second, in addition to the 100 Hz clock interrupts. It obviously all depends on what has to be done as the result of an interrupt.

With prices of PCs and PC-boards going down, there is now a tendency to use a PC-board also for an **embedded system**, where before you would have used a small, dedicated microprocessor. Using a PC-board has the obvious advantage of **portability**: you can develop your application on a large configuration, and then **download** it to the embedded system.

Many people may be just interested in hooking up existing instruments, for instance those which are equipped with an interface to the GPIB bus. This situation arises routinely in chemistry labs, or medical analysis labs, etc. There is good news for those people as well: Packages for controlling instruments with GPIB and Camac exist. The first parts of these were released already in 1995. It has graphical interfaces, uses X11 and is extensible[19]. And **they are free!**. Such packages will certainly deliver the ideal solution for laboratories using standard equipment.

Device drivers for *VME crates and modules* are part of some off-the-shelf Linux distributions (*SuSE 6.3 for instance*).

To end, I wish you a happy time programming your real-time applications! Now that many more tools are available than a few years ago, there is a good chance that this wish comes true.

Enjoy!

[19]You can ftp these packages from koala.chemie.fu-berlin.de.

16 Annex I – Annotated bibliography

The last four or five years have seen a flood of books on **Linux**. A number of them are nothing but collections of **HOW-TOs** from the **Linux Documentation Project**. Others are specific for certain **Linux Distributions**, e.g. *Slackware, RedHat, Caldera Desktop, Yggdrasil Plug and Play Linux.* These books generally contain one or more *CD-ROMs*, or the CD-ROM set is sold separately (the *Linux Developers Resource* from *InfoMagic* is an example).

Below is an annotated bibliography of the books I found most useful and which is limited to those publications which are **not** specific to a distribution, or just collections of HOW-TOs. Unfortunately the list has scarcely any item more recent than 1998.

1. Matt Welsh and Lar Kaufman, *Running Linux*, Sebastopol, CA95472, 1995, O'Reilly & Associates, Inc; ISBN 1-56592-100-3
   An excellent book, very complete and very readable. Contains extensive indications on how to obtain and install Linux, followed by chapters on UNIX commands, System Administration, Power Tools (including X11, emacs and LaTeX), Programming, Networking. The annexes contain a wealth of information on documentation, ftp-sites, etc. One of the most readable books on Linux.

2. Marc Ewing, *Running Linux Installation Guide and Companion CD-ROM*, O'Reilly & Associates, Inc.; no apparent ISBN.

3. Matt Welsh, *Linux Installation Guide*, 1995, Pacific Hi-Tech, 3855 South 500 West Suite M, Salt Lake City, Utah 84115,
   email: orders@pht.com; No ISBN found.
   The book is thin (221 pages) and cheap ($ 12.95). It contains a few extra chapters on XFree86, TCP/IP, UUCP, e-mail and usenet.

4. Olaf Kirch, *Linux Network Administrator's Guide*, Sebastopol CA95472, 1995, O'Reilly & Associates, Inc; ISBN 1-56592-087-2
   Another excellent book on Networking for Linux. Covers not only local networks and TCP/IP, but also the use of a serial line to connect to Internet, and other chapters on NFS, Network Information System, UUCP, e-mail and News Readers. Essential reading if you want to use your Linux box on the network.

5. Stefan Strobel and Thomas Uhl, *Linux, unleashing the workstation in your PC*, Berlin, 1994, Springer Verlag; ISBN 3-540-58077-8
   This book is good to whet the appetite of someone who has no idea of

what Linux is or what it can do. It has many illustrations, in particular of graphics applications and it mentions many software packages which are not part of the usual Linux distributions, together with indications on how to obtain and install the package.

6. *Linux Bible*, 1994, San Jose, Yggdrasil Computing. No apparent ISBN. I know about this book only from the advertisements.

7. Kamram Hussain, Timothy Parker et al., *Linux Unleashed*, 1996, SAMS Publishing, ISBN 0-672-30908-4
   Approx 1100 pages of text, covering Linux and many tools and applications: Editing and typesetting (groff and Tex), Graphical User Interfaces, Linux for programmers (C, C++, Perl, Tcl/Tk, Other languages, Motif, XView, Smalltalk, Mathematics, Database products), System Administration, Setting up an Internet site and Advanced Programming topics. The book contains a CD-ROM with the Slackware distribution.

8. Randolph Bentson, *Inside Linux, a look at Operating System Development*, 1996, Seattle, Specialized system Consultants, Inc; ISBN 0-916151-89-1.
   This book provides some more insight into the internal workings of operating systems, with the emphasis being placed on Linux. It is written in general terms and does not contain code examples.

9. John Purcell (ed.), *Linux MAN, the essential manpages for Linux*, 1995, Chesterfield MI 48047, Linux Systems Lab, ISBN 1-885329-07-5.
   Indispensable for those who cannot stare at a screen for more than 8 hours a day, or who like to sit down in a corner to write their programs with pencil and paper, but want to be sure they use system calls correctly. As the title says, 1200 pages of "man pages" for Linux, from *abort* to *zmore*, and including system calls, library functions, special files, file formats, games, system administration and a kernel reference guide.

10. M. Beck, H. Böhme, M. Dziadzka, U. Kunitz, R. Magnus, D. Verworner, *Linux Kernel Internals*, 1996, Addison Wesley, ISBN 0-201-87741-4.
    There is at least a second edition: ISBN 0-201-33143-8, 1998. For the real sports! A translation of a german book, revealing all the internals of the Linux kernel, including code examples, definitions of structures, tables, etc. The book contains a CD-ROM with Slackware and kernel sources. Indispensable if you want to make modifications to the kernel yourself.

11. Alessandro Rubini, *Linux Device Drivers*, 1998, O'Reilly & Associates, ISBN 1-56592-292-1. This book is a **real must** for anyone wanting to write or modify a device driver for Linux. Before publishing this book, the author had written many articles in the *kernel corner* of *Linux Journal*. The book leads the reader step by step through every corner of a Linux device driver. No secrets are left unveiled.

Having mentioned *Linux Journal*, I should add that you can subscribe via one of the following addresses: e-mail: subs@ssc.com, or on the web: www.linuxjournal.com, Fax: +1-206 297 7515 and by normal mail: SSC, Specialized System Consultants, Inc., PO Box 55549, Seattle, WA 98155-0549, USA. From some thirty pages back in 1994, Linux Journal has grown to around 200 pages monthly. A good fraction of the pages is nowadays occupied by advertisements, but there remain still some 120 or more pages of interesting reading.

Also note that the last few years a number of periodicals on Linux have seen the light in languages other than english.

The following books concern **real-time** and **POSIX.1c**:

i) Bill O. Gallmeister, *POSIX.4: Programming for the Real World*, 1995, O'Reilly & Associates, Inc.; ISBN 1-56592-074-0.
This book gives an in-depth treatment of programming real-time applications, based on the POSIX.4 standard. Several of the examples in the present course were taken from this book. In addition to approximately 250 pages of text, the book contains 200 pages of "man pages" and solutions to exercises.

ii) Bradford Nichols, Dick Buttlar and Jacqueline Proulx Farrell, *Pthreads Programming*, 1996, O'Reilly & Associates, Inc, ISBN 1-56592-115-1.
Probably the best book on Pthreads published so far, concentrating on the POSIX 1003.1c standard and written with a good didactical structure.

iii) Bil Lewis, Daniel J. Berg, *Threads Primer, A Guide to Multithreaded Programming*, 1996, Sunsoft Press (Prentice Hall);
ISBN 0-13-443698-9.
An introduction to threads programming, mainly based on the Solaris implementation of threads, but containing comparisons to POSIX threads and a full definition of the *Applications Programmer's Interface* to **POSIX.4a pthreads**.

iv) S. Kleiman, Devang Shah, B. Smaalders, *Programming with Threads*, 1996, Sunsoft Press (Prentice Hall; ISBN 0-13-172389-8. This book contains a more in-depth treatment of threads programming than the previous title. It is also more pthreads-oriented.

v) Andrew S. Tanenbaum, *Modern Operating Systems*, 1992, Prentice Hall; ISBN 0-13-595752-4.
This excellent book is not specifically tuned to real-time, but it provides a comprehensive introduction to the features of modern operating systems and their implementation. An older edition of the book contained a complete listing of the **minix** operating system. The reader may appreciate that Linux was born when Linus Torvalds set out to improve minix ...

vi *Last minute addition:* O'Reilly is expected to issue a *community written* book on Linux realtime. The editor is *Phil Daly* of *realtimelinux.org*. The announcement says: *"The volume will be the definite guide to the installation and use of real time Linux and will feature a bootable CD-ROM to help "get you going" with this exciting technological development. The text will be made available under an Open Content License agreement with content under constant review."*

Note that there are many more books available, in particular from O'Reilly, which may be of relevance to topics treated in the present course. Finally, there is a paper on Real Time Linux:
M. Barabanov and V. Yodaiken, *Introducing Real-Time Linux*, Linux Journal, February 1997, pages 19-23.

For RTAI, there is a large amount of documentation available from: http://www.aero.polimi.it/projects/rtai/.

To conclude, some mailing lists which may be useful. To subscribe to any of the lists, send an email to: **missdomo@realtimelinux.org** with **subscribe 'listname'** in the **body** of the email.

- **realtime** — realtime@realtimelinux.org — general discussion.
- **api** — api@realtimelinux.org — API discussion.
- **documentation** — documentation@realtimelinux.org
- **drivers** — drivers@realtimelinux.org — discussion of drivers for use with Realtime Linux.
- **kernel** — kernel@realtimelinux.org — Linux kernel modifications for use with Realtime Linux.
- **networking** — networking@realtimelinux.org — Realtime networks.
- **ports** — ports@realtimelinux.org — Porting of RTL/RTAI to other platforms.
- **testing** — testing@realtimelinux.org — discussion on testing

17 Annex II – CD-ROM sets

All the well-known Linux distributions (*Caldera, Corel, Debian, RedHat, Slackware, SuSE* and I will certainly miss out a few...) now come on two or more CD-ROMs. Beside the base system they contain in general a wealth of additional, optional packages and a lot of documentation. If you have acquired one of these distributions, there will be no real need for other CD-ROMs, possibly with one exception:

"Linux Developers Resource 6 CD set", approx.$ 50.00 (I paid approximately 70000 Lit a few years back). Contains Several Linux distributions and many many things from the GNU and other archive sites. Available from: *InfoMagic, P.O. Box 30370, Flagstaff, AZ 86003, fax: +1-602-526-9573, e-mail: info@infomagic.com.*
This is probably the most useful CD-ROM set. It is updates twice a year. You can also open a subscription and receive the bi-annual update automatically.

18 Annex III — Resume of POSIX 1003.1c definitions

POSIX.1c/D10 Summary

Disclaimer

Introduction

All source that uses POSIX.1c threads must include the header file.

```
#include <pthread.h>
```

In addition, Solaris requires the pre-processor symbol **_REENTRANT** to be defined in the source code before any C source (including header files).

```
#define_REENTRANT
```

The POSIX.1c thread library should be the last library specified on the cc(1) command line.

```
voyager% cc -D_REENTRANT ... -lpthread
```

Name Space

Each POSIX.1c type is of the form:

```
pthread[_object]_t
```

Each POSIX.1c function has the form

```
pthread[_object]_operation[_np|_NP]
```

where *object* is a type (not required if object is a thread), operation is a type-specific operation and np (or NP) is used to identify non-portable, implementation specific functions.

All POSIX.1c functions (except for pthread_exit, pthread_getspecific and pthread_self) return zero (0) for success or an errno value if the operation fails.

There are eight(8) POSIX.1c types:

Table 0-1 POSIX.1c types

Type	Description
pthread_attr_t	Thread attribute
pthread_mutexattr_t	Mutual Exclusion Lock attribute
pthread_condattr_t	Condition variable attribute
pthread_mutex_t	Mutual Exclusion Lock (mutex)
pthread_cond_t	Condition variable (cv)
pthread_t	Thread ID
pthread_once_t	Once-only execution
pthread_key_t	Thread Specific Data (TSD) key

Feature Test Macros

POSIX.1c consists of a base (or common) component and a number of implementation optional components The base is the set of required operations to be supplied by every implementation. The pre-processor symbol (**_POSIX_THREADS**) can be used to test for the presence of the POSIX.1c base. Additionally, the standards document describes a set of six (6) optional components. A pre-processor symbol can be used to test for the presence of each All of the symbols appear in the following table.

Table 0-2 POSIX.1c Feature Test Macros

Feature Test Macro	Description
_POSIX_THREADS	base threads
_POSIX_THREAD_ATTR_STACKADDR	stack address attribute
_POSIX_THREAD_ATTR_STACKSIZE	stack size attribute
_POSIX_THREAD_PRIORITY_SCHEDULING	thread priority scheduling
_POSIX_THREAD_PRIO_INHERIT	mutex priority inheritance
_POSIX_THREAD_PRIO_PROTECT	mutex priority ceiling
_POSIX_THREAD_PROCESS_SHARED	inter-process synchronization

Macro Dependency

If **_POSIX_THREAD_PRIO_INHERIT** is defined then **_POSIX_THREAD_PRIORITY_SCHEDULING** is defined.

If **_POSIX_THREAD_PRIO_PROTECT** is defined then **_POSIX_THREAD_PRIORITY_SCHEDULING** is defined.

If **_POSIX_THREAD_PRIORITY_SCHEDULING** is defined then **_POSIX_THREADS** is defined.

If **_POSIX_THREADS** is defined then **_POSIX_THREAD_SAFE_FUNCTIONS** is defined.

POSIX.1c API

In the following sections, function arguments that are of the form:

```
type name = NULL
```

indicate that a value of **NULL** may safely be used for name.

```
int     pthread_atfork( void (*prepare)(void) = NULL,
                        void (*parent)(void) = NULL,
                        void (*child)(void) = NULL );
```

Register functions to be called during fork execution.

errors **ENOMEM**

notes `prepare` functions are called in reverse order of registration. `parent` and `child` functions are called in order of registration.

Thread Attributes

All thread attributes are set in an attribute object by a function of the form:

```
int     pthread_attr_setname( pthread_attr_t *attr, Type t );
```

All thread attributes are retrieved from an attribute object by a function of the form:

```
int     pthread_attr_getname( const pthread_attr_t *attr, Type *t );
```

Where *name* and *Type* are from the table below.

Table 0-3 Thread Attributes

Name and Type	Feature Test Macro	Value(s)
`int inheritsched`	**_POSIX_THREAD_PRIORITY_SCHEDULING**	**PTHREAD_INHERIT_SCHED,**
		PTHREAD_EXPLICIT_SCHED
`int schedpolicy`	**_POSIX_THREAD_PRIORITY_SCHEDULING**	**SCHED_FIFO,**
		SCHED_RR,
		SCHED_OTHER
`struct sched_param schedparam`	**_POSIX_THREADS**	POSIX.1b, Section 13
`int contentionscope`	**_POSIX_THREAD_PRIORITY_SCHEDULING**	**PTHREAD_SCOPE_SYSTEM,**
		PTHREAD_SCOPE_PROCESS
`size_t stacksize`	**_POSIX_THREAD_ATTR_STACKSIZE**	**>= PTHREAD_STACK_MIN**

Table 0-3 Thread Attributes

Name and Type	Feature Test Macro	Value(s)
`void *stackaddr`	**_POSIX_THREAD_ATTR_STACKADDR**	`void *stack`
`int detachstate`	**_POSIX_THREADS**	**PTHREAD_CREATE_DETACHED,**
		PTHREAD_CREATE_JOINABLE

```
int     pthread_attr_init( pthread_attr_t *attr );
```

Initialize a thread attribute object.

errors **ENOMEM**

```
int     pthread_attr_destroy( pthread_attr_t *attr );
```

Destroy a thread attribute object.

errors none

Thread Management

```
int     pthread_create( pthread_t *thread,
                        const pthread_attr_t *attr = NULL,
                        void *(*entry)(void *), void *arg );
```

Create a new thread of execution.

errors **EAGAIN, EINVAL**

note Maximum number of **PTHREAD_THREADS_MAX** threads per process

```
int     pthread_detach( pthread_t thread );
```

Set the `detachstate` of the specified thread to **PTHREAD_CREATE_DETACHED**.

errors **EINVAL, ESRCH**

```
pthread_t    pthread_self( void );
```

Return the thread ID of the calling thread.

errors none

```
int     pthread_equal( pthread_t t1, pthread_t t2 );
```

Compare two thread IDs for equality.

errors none

```
void    pthread_exit( void *status = NULL );
```

Terminate the calling thread.

errors none

```
int     pthread_join( pthread_t thread, void **status = NULL );
```

Synchronize with the termination of a thread.

errors **EINVAL, ESRCH, EDEADLK**

note This function is a cancellation point.

```
#include <sched.h>
int     pthread_getschedparam( pthread_t thread, int *policy, struct sched_param *param );
```

Get the scheduling policy and parameters of the specified thread.

control **_POSIX_THREAD_PRIORITY_SCHEDULING**

errors **ENOSYS, ESRCH**

```
#include <sched.h>
int     pthread_setschedparam( pthread_t thread, int policy,
                               const struct sched_param *param );
```

Set the scheduling policy and parameters of the specified thread.
control **_POSIX_THREAD_PRIORITY_SCHEDULING**
errors **ENOSYS, EINVAL, ENOTSUP, EPERM, ESRCH**
`policy` **{ SCHED_RR, SCHED_FIFO, SCHED_OTHER }**

Mutex Attributes

All mutex attributes are set in a mutex attribute object by a function of the form:

`int pthread_mutexattr_setname( pthread_attr_t *attr, Type t );`

All mutex attributes are retrieved from a mutex attribute object by a function of the form:

`int pthread_mutexattr_getname( const pthread_attr_t *attr, Type *t );`

Where *name* and *Type* are from the table below

Table 0-4 Mutex Attributes

Name and Type	Feature Test Macro	Value(s)
`int protocol`	**_POSIX_THREAD_PRIO_INHERIT,**	**PTHREAD_PRIO_NONE,**
	_POSIX_THREAD_PRIO_PROTECT	**PTHREAD_PRIO_PROTECT,**
		PTHREAD_PRIO_INHERIT
`int pshared`	**_POSIX_THREAD_PROCESS_SHARED**	**PTHREAD_PROCESS_SHARED,**
		PTHREAD_PROCESS_PRIVATE
`int prioceiling`	**_POSIX_THREAD_PRIO_PROTECT**	POSIX.1b, Section 13

`int pthread_mutexattr_init( pthread_mutexattr_t *attr );`
Initialize a mutex attribute object.
errors **ENOMEM**

`int pthread_mutexattr_destroy( pthread_mutexattr_t *attr );`
Destroy a mutex attribute object.
errors **EINVAL**

Mutex Usage

`int pthread_mutex_init( pthread_mutex_t *mutex, const pthread_mutexattr_t *attr = NULL );`
`pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;`
Initialize a mutex.
errors **EAGAIN, ENOMEM, EPERM, EBUSY, EINVAL**

`int pthread_mutex_destroy( pthread_mutex_t *mutex );`
Destroy a mutex.
errors **EBUSY, EINVAL**

`int pthread_mutex_getprioceiling( const pthread_mutex_t *mutex, int *prioceiling );`
Get the `prioceiling` value of the specified mutex.
control **_POSIX_THREAD_PRIO_PROTECT**
errors **ENOSYS, EINVAL, EPERM**

`int pthread_mutex_setprioceiling( pthread_mutex_t *mutex, int prioceiling,`
`int *old_ceiling );`

Set the `prioceiling` value and return the old `prioceiling` value in the specified mutex.
control **_POSIX_THREAD_PRIO_PROTECT**
errors **ENOSYS, EINVAL, EPERM**

`int pthread_mutex_lock( pthread_mutex_t *mutex );`
Acquire the indicated mutex.
errors **EINVAL, EDEADLK**

`int pthread_mutex_trylock( pthread_mutex_t *mutex );`
Attempt to acquire the indicated mutex.
errors **EINVAL, EBUSY, EINVAL**

`int pthread_mutex_unlock( pthread_mutex_t *mutex );`
Release the (previously acquired) mutex.
errors **EINVAL, EPERM**

Once-only Execution

`pthread_once_t once = PTHREAD_ONCE_INIT;`
Initialize a once control variable.

`int pthread_once( pthread_once_t *once_control, void (*init_routine)(void) );`
Execute `init_routine` once.
errors none specified

Condition Variable Attributes

All condition variable attributes are set in a condition variable attribute object by a function of the form:

`int pthread_condattr_setname( pthread_condattr_t *attr, Type t );`

All condition variable attributes are retreived from a condition variable attribute object by a function of the form:

`int pthread_condattr_getname( const pthread_condattr_t *attr, Type *t );`

Where *name* and *Type* are from the table below

Table 0-5 Condition Variable Attributes

Name and Type	Feature Test Macro	Value(s)
`int pshared`	**_POSIX_THREAD_PROCESS_SHARED**	**PTHREAD_PROCESS_SHARED,**
		PTHREAD_PROCESS_PRIVATE

`int pthread_condattr_init( pthread_condattr_t *attr );`
Initialize a condition variable attribute object.
errors **ENOMEM**

`int pthread_condattr_destroy( pthread_condattr_t *attr );`
Destroy a condition variable attribute object.
errors **EINVAL**

Condition Variable Usage

`int pthread_cond_init( pthread_cond_t *cond,`

```
        const pthread_condattr_t *attr = NULL );
pthread_cond_t    cond        = PTHREAD_COND_INITIALIZER;
```
Initialize a condition variable.
errors **EAGAIN, ENOMEM, EBUSY, EINVAL**

`int pthread_cond_destroy( pthread_cond_t *cond );`
Destroy a condition variable.
errors **EBUSY, EINVAL**

`int pthread_cond_signal( pthread_cond_t *cond );`
Unblock at least one thread currently blocked in the specified condition variable.
errors **EINVAL**

`int pthread_cond_broadcast( pthread_cond_t *cond );`
Unblock all threads currently blocked on the specified condition variable.
errors **EINVAL**

`int pthread_cond_wait( pthread_cond_t *cond, pthread_mutex_t *mutex );`
Block on the specified condition variable.
errors **EINVAL**
note This function is a cancellation point.

```
int   pthread_cond_timedwait( pthread_cond_t *cond, pthread_mutex_t *mutex,
              const struct timespec *abstime );
```
Block on the specified condition variable not longer than the specified absolute time.
errors **ETIMEDOUT, EINVAL**
note This function is a cancellation point.

Thread Specific Data

`int pthread_key_create( pthread_key_t *key, void (*destructor)(void *) = NULL );`
Create a thread-specific data key.
errors **EAGAIN, ENOMEM**
note system limit of **PTHREAD_KEYS_MAX** per process.
system limit of **PTHREAD_DESTRUCTOR_ITERATIONS** calls to `destructor` per thread exit.

`int pthread_key_delete( pthread_key_t key );`
Destroy a thread-specific data key.
errors **EINVAL**

`void *pthread_getspecific( pthread_key_t key );`
Return the value bound to the given key for the calling thread.
errors none

`int pthread_setspecific( pthread_key_t key, const void *value );`
Set the value for the given key in the calling thread.
errors **ENOMEM, EINVAL**

Signal Management

`#include <signal.h>`
`int pthread_sigmask( int how, const sigset_t *newmask = NULL, sigset_t *oldmask = NULL );`
Examine or change calling threads signal mask.

errors **EINVAL**
how **{ SIG_BLOCK, SIG_UNBLOCK, SIG_SETMASK }**

`#include <signal.h>`
`int pthread_kill( pthread_t thread, int signo );`
Deliver signal to indicated thread.
errors **ESRCH, EINVAL**

`#include <signal.h>`
`int sigwait( const sigset_t *set, int *sig );`
Synchronously accept a signal.
errors **EINVAL, EINTR**
note This function is a cancellation point.

Cancellation

`int pthread_setcancelstate( int state, int *oldstate );`
Set the cancellation state for the calling thread.
errors **EINVAL**
`state` **{ PTHREAD_CANCEL_ENABLE, PTHREAD_CANCEL_DISABLE }**

`int pthread_setcanceltype( int type, int *oldtype );`
Set the cancellation type for the calling thread.
errors **EINVAL**
`type` **{ PTHREAD_CANCEL_DEFERRED, PTHREAD_CANCEL_ASYNCHRONOUS }**

`int pthread_cancel( pthread_t thread );`
Cancel the specified thread.
errors **ESRCH**
note threads that have been cancelled terminate with a status of **PTHREAD_CANCELED.**

`void pthread_testcancel( void );`
Introduce a cancellation point.
errors none
note This function is a cancellation point.

`void pthread_cleanup_pop( int execute );`
Pop the top item from the cancellation stack and optionally execute it.
errors none specified
note push and pop operations must appear at the same lexical level.
`execute` **{ 1, 0 }**

`void pthread_cleanup_push( void (*routine)(void *), void *arg );`
Push an item onto the cancellation stack.
errors none specified

C Refresh

Sixth College on Microprocessor-based Real-time Systems in Physics

Abdus Salam ICTP, Trieste. October 9–November 3, 2000

Carlos Kavka
Departamento de Informática
Universidad Nacional de San Luis
San Luis
Argentina.

email: ckavka@unsl.edu.ar

Abstract

This chapter is intended to refresh your C programming language knowledge. It is not a complete guide or reference on the language. The topics are introduced mainly through simple examples.

1 Introduction

The C programming language was developed by Dennis Ritchie in the Bell Laboratories and was designed to be run on a PDP-11 computer with a Unix operating system. It is a small, flexible and concise language, with a mix of low-level assembler-style commands and high-level commands. It is an excellent selection in those areas where you may want to use assembly language but would keep it a 'simple to write' and 'easy to maintain' program.

The first standard was the Kernighan and Ritchie's book: "The C programming language" (1988). The ANSI C standard was defined when it was evident that the C programming language was becoming a very popular language. The ANSI C standard defines not only the syntax and semantics of the programming language but also a standard library. We will follow this standard in all the examples.

There is also another standard known as POSIX.1, which defines the interface of the system calls and some library functions, used to obtain services from the operating system. We will encounter this standard in the examples.

2 Getting started

The first example (see figure 1) is a program that prints the mean of two integer values. Not so much, but enough to begin. Note that the line numbers in the left column do not belong to the program; they are intended only for reference.

All C programs need a `main` function, and this is the place where the execution begins. In this example, three local variables in `main` are created. The first two, named `a` and `b` are of type integer, and the other one, named `answer` is of type float.

The sentences in line *11* assign values to the two integer variables. Then the function `mean` is called to calculate the mean of the two integer arguments given to it. The types of the formal parameters of the function (in this case `x` and `y`) should be compatible with the actual parameters in the call. The initial values of `x` and `y` are copied from the variables mentioned in the call (`a` and `b`).

The function `mean` returns the mean of the two integer arguments (a float, hence the float before the function name). It also declares a local variable `f` of type float to be used to store the mean value, which is computed in line *5*. This value is returned to the `main` program through the return statement.

We have used 2.0 (a float constant) instead of 2 (an integer constant) in line *5*, because we want a float as the result of the division operation. If we

```
#include <stdio.h>
#include <stdlib.h>

float mean(int x,int y) {
  float f;

  f = (x + y) / 2.0;
  return f;
}

int main() {
  int a,b;
  float answer;

  a = 3; b = 2;
  answer = mean(a,b);
  printf("the mean of %d and %d is %f\n",a,b,answer);
  exit(0);
 }
```

Figure 1: a program to print the mean of two integers

divide an integer by another integer, we obtain an integer:

```
(3 + 2) / 2   = 2
(3 + 2) / 2.0 = 2.5
```

The values are printed in the `main` function by using the standard library function `printf`. The on-line manual page describes this function fully. For now, just note that the first argument is a string in which some format specifiers are embedded: %d for integers and %f for floats. The string is printed with these format specifiers replaced with the values of the variables that follow the string as arguments.

The `exit` function terminates a program normally. It expects a single integer as argument, which is called the *exit status*, and can be examined by the process which puts this process to run (possibly the shell). If simply `main` 'falls off the end' (implicit return), the exit status of the process is undefined. By convention, an argument of 0 means OK, and an argument between 1 and 255 means an error has occurred.

The two `#include` directives found in lines *1* and *2*, instruct the preprocessor to include the definitions and declarations from the include files `stdio.h` and `stdlib.h`. For standard include files we use the form <*filename*> to indicate that the standard include directory must be searched. Our own directory is searched first if we use the form "*filename*" instead. The on-line manual page for each function shows which files must be included.

3 Control structures

3.1 Repetition statements

The C programming language provides three structures for looping: the `while` loop, the `do while` loop and the `for` loop.

The `while` loop continues to loop while some condition is true. When the condition is false, the looping is discontinued. Let's see an example (figure 2).

```
1.  #include <stdio.h>

2.  int main() {
3.    int i = 1,sum = 0;

4.    while(i < 5) {
5.      sum += i;
6.      i++;
7.    }
8.    printf("summation is %d\n",sum);
9.    exit(0);
10. }
```

Figure 2: `while` statement

This small program just prints the summation of the integer numbers from 1 to 4. As long as the expression of the while statement in parenthesis is true, all statements within the braces are repeatedly executed.

If the variable `i` were initialized to any number greater than or equal to 5, the statements inside the braces of the while loop would not be executed at all. If the variable were not incremented in the loop, the loop would never terminate. If there were just one statement to be executed within the loop, no braces would be needed.

Note the short expressions used in lines *5* and *6*. The operator `++` is called the increment operator. The meaning of these expressions is the following:

```
sum += i      sum = sum + i
i++           i = i + 1
```

The `for` loop is nothing new; just a new way to describe the `while` loop. The same example from figure 2 is re-written using the `for` statement in figure 3.

```
#include <stdio.h>

int main() {
  int i,sum;

  for(sum = 0,i = 1;i < 5;i++)
    sum += i;

  printf("summation is %d\n",sum);
  exit(0);
}
```

Figure 3: `for` statement

The `for` statement has three expressions separated by semi-colons (`;`). The first one contains sentences that are executed prior to the first pass through the loop. In this case, two assignments. The comma (`,`) operator allows to put more than one expression, where only one is allowed. The second field is the test which is evaluated at the beginning of each pass through the loop. The third field is executed in every pass, but after all the statements in the body of the loop.

The `for` loop is convenient because all the control information of the loop is in one place. We will see later more examples on the use of the `for` statement.

The other construction, the `do while` loop is a variation of the `while` loop. The main difference is that the condition is evaluated at the end of the loop. This means that the body is executed at least once. The same example is re-written in figure 4. Note that the meaning of the program is the same, because in both cases, the body of the loop is executed at least once.

```
#include <stdio.h>

int main() {
  int i = 1,sum = 0;

  do {
    sum += i;
    i++;
  } while (i < 5);
  printf("summation is %d\n",sum);
  exit(0);
}
```

Figure 4: `do ... while` statement

3.2 break and continue

The `break` statement is used to jump out from a loop.

The `continue` statement does not cause a termination of the loop but causes a jump out of the present iteration. It always jumps to the end of the loop just prior the terminating brace. The loop is terminated or not based on the loop test. In the `for` statement, the last expression is evaluated as usual. A complete example is provided in figure 5.

3.3 if and switch

In the simplest form, the `if` statement has a condition and a statement. If the condition is true, the statement is executed, and if it is false, the statement is skipped. Note that the single statement can be replaced by a compound statement composed of several statements between braces.

The second form is similar, but with the addition of the word `else` and another statement. If the condition is false, this statement is executed.

The `switch` statement is like a multi-branch if. The key word `switch` is followed by a value between parenthesis, and a set of cases between braces, identified by the word `case` followed by a constant. The control is transferred to the first statement of the case whose constant is the same as the value between parenthesis. If no constant is found, the control is then transferred to the first sentence after the key word `default`, if there is one. If no case

```
#include <stdio.h>
#define N     50

int main() {
  int i,c;
  int n_spaces = 0,n_symbols = 0,n_chars = 0;

  for(i = 0;i < N;i++) {
    c = getchar();

    if (c == EOF) break;
    if (c == '\n') continue;

    switch(c) {
      case ' ': n_spaces++;
                break;
      case ',':
      case '.':
      case ';': n_symbols++;
      default:  n_chars++;
    }
  }
  printf("chars: %d spaces: %d symbols %d\n",
          n_chars,n_spaces,n_symbols);
  exit(0);
}
```

Figure 5: control statements

matches, and there is no default, no action is performed. Once an entry point is found, statements will be executed until a `break` is found, or until the control runs out of the switch braces.

An example that shows the use of most of the control structures discussed so far is presented in figure 5.

The objective of this program is to read at most 50 characters from the standard input, and print the number of spaces, the number of punctuation symbols (only '.', ';' and ','), and the number of characters read without considering spaces. Newlines ('\n') must be ignored, but considered in the

50 characters limit. EOF should be considered as the end of the input.

The line *2* contains a definition of a constant by using the preprocessor directive `#define`. Each occurrence of the identifier `N` in the program is replaced with the string 50. It can also be used to define macros.

The function `getchar()` reads a character from the standard input.

The `break` in line *8* will cause a jump out of the loop effectively terminating the loop, if the character read is an EOF. The `continue` statement on line *9* will cause a jump to the end of the loop if the character read is a newline. The third expression of the `for` statement (`i++`) will be executed.

If the character is a space, the corresponding counter (`n_spaces`) will be incremented, and the `break` in line *12* will cause a jump out of the `switch` statement.

If the character is a symbol, the corresponding counter (`n_symbols`) will be incremented, and the execution will continue also with the default sentences, allowing the counter of characters (`n_chars`) to be incremented.

If the character does not belong to this set, the default sentences are executed, incrementing the counter of characters (`n_chars`).

4 Expressions

- Most operations in C that are designed to operate with integers will work equally well with characters, because they are a form of integer values. The following code will convert upper case characters to lower case.

  ```
  int c;

  c = getchar();
  if (c >= 'A' && c <= 'Z')
    c = c - 'A' + 'a';
  ```

  The `&&` operator is the *logical and*. The *logical or* operator is `||` and the *negation* operator is `!`.

- The operators `++` and `--` are known as the increment and decrement operators respectively. `i++` is equivalent to `i = i + 1`, and `i--` is equivalent to `i = i - 1`. The operation can be done after the variable is used, or before, by using `i++` or `++i`, so

  ```
  ...
  ```

```
i = 10;
printf("i = %d\n",i++);
```

and

```
...
i = 10;
printf("i = %d\n",++i);
```

will both leave i as 11, but in the first example 10 will be printed, and in the second 11 will.

- There is an abbreviated form that can be used with binary operators. For example, the following expressions are equivalent:

```
i = i + 6        i += 6
i = i * 12       i *= 12
```

- The assignment expressions produce a value: the value that is effectively assigned, so

```
a = (b = 1 + 2) + 4;
```

will assign 3 to b and 7 to a.

- Comparisons will return 1 if the comparison is true, and 0 if it is false, so

```
i = (3 <= 8) + 2;
```

will assign 3 to i.

- There is no boolean type in C; the integer 0 stands for false, and any number different from 0 is considered as true. So

```
while(i != 0)
```

where != stands for *different*, is equivalent to `while(i)`.

- Some conditional expressions can be abbreviated by using the conditional operator (`?:`). For example,

  ```
  if (x < 2)
    a = 5;
  else
    a = 12;
  ```

  can be re-written as

  ```
  a = (x < 2) ? 5 : 12;
  ```

- There are also operations for bit manipulation, that can be applied to operands of types `int`, `short`, `long`, `unsigned` and `char`. They are the *bitwise and* `&`, the *bitwise or* `|`, the *bitwise exclusive or* `^`, the *left shift* `<<`, the *right shift* `>>` and the *one's complement* `~`.

  The example of the figure 6 shows a function used to count the number of bits in 1 in an `unsigned long`.

```
1.  int n_bits(unsigned long x) {
2.    int n = 0;

3.    while (x) {
4.      if (x & 0x01) n++;
5.      x >>= 1;
6     }
7.  return n;
8.  }
```

Figure 6: bit manipulation

As we previously said, `while(x)` is equivalent to `while(x != 0)`. This is a safe stop point, because we are shifting `x`, and as it is unsigned, it is filled with 0's from the left.

The test on line *4* checks if the least significant bit of `x` is 1. Note that the constant 0x01 is hexadecimal. If a constant begins with 0 (zero) it is an octal one (like 077).

The expression in line *5* is an abbreviated form of `x = x >> 1`.

- The *cast* operator can be used to prescribe a conversion to a target data type, independent of the context. For example,

  ```
  int x = 5,y = 2;
  float f,g;

  f = x / y;
  g = x / (float)y;
  ```

  will assign 2 to f, and 2.5 to g. The cast consists of the name of a type between parenthesis.

5 Arrays, Structures and Unions

An array is a set of contiguous variables of the same type, that can be accessed through an integer index. For example, the declaration:

```
int a[100];
```

reserves memory for 100 integer variables. They can be accessed by using subscripts from 0 to 99. For example, the program in figure 7 initializes all the components in an array and then print the summation of them.

A structure is a collection of variables grouped as a single object, where each one could be from a different type. For example, the following structure could be used to define a point giving its x and y coordinates:

```
structure point {
  float x;
  float y;
};
```

We can declare variables of this type:

```
struct point a,b;
```

and fill data by using the dot (.) operator:

```
a.x = 2.5;
a.y = 5.6;
```

We could have created an initialized point by using:

```
#include <stdio.h>
#define N       50

int main() {
  int a[N],i,sum = 0;

  for(i = 0;i < N;i++)
    a[i] = i * 2;

  for(i = 0;i < N;i++)
    sum += a[i];

  printf("summation is %d\n",sum);
  exit(0);
}
```

Figure 7: arrays

```
struct point b = { 5.0 , 1.25 };
```

Structures can be assigned, passed to functions and returned, but they cannot be compared, so:

```
c = a;
```

is possible (all the fields from a are copied into c), but you cannot do:

```
if (a == b) ...                 /* not possible */
```

The figure 8 shows a program that assigns into a point structure c the structure a if a is equal to b. If this is not the case, the greater coordinates between a and b are assigned to c.

A union is like a structure, but the fields occupy the same memory locations, with enough memory allocated to hold the largest one. For example, the following union has two fields that overlap.

```
union option {
  int number;
  float price;
};
```

```
struct point {
  float x;
  float y;
};

int main() {
  struct point a = { 2.3 , 3.1 },b,c;

  b.x = 1.5;
  b.y = 8.9;

  if (a.x == b.x && a.y == b.y)
    c = a;
  else {
    c.x = (a.x > b.x) ? a.x : b.x;
    c.y = (a.y > b.y) ? a.y : b.y;
  }
  exit(0);
}
```

Figure 8: structures

An assignment to one of its fields overlap what it has in the other, so

```
union option x;

x.number = 13;
x.price = 12.5;
```

the value 13 will be over-written with the value 12.5. The programmer has to remember what the union is used for.

Structures and arrays can be combined, for example,

```
struct point arr[10];
```

is an array of then structures point, and their components can be accessed for example as follows:

```
arr[4].x = 3;
```

6 Type declarations

The `typedef` declaration allows us to give an identifier to a type, so it can be used in the same way as the predefined ones. For example,

```
typedef int integer;
```

will define the type `integer` as the standard type `int`, so now we can declare an `int` variable `x` by doing:

```
integer x;
```

A more useful example is the following:

```
typedef int array[100];
typedef struct point Point;
```

Now, we can declare:

```
array a;
Point x;
```

and `a` is an array of 100 integers, and `x` a structure. An array of 10 structures point can be defined as:

```
Point b[10];
```

7 Pointers

All variables are stored in some position in the memory, for example, as a result of

```
int i = 10;
```

the situation in the memory (simplified) could be as is shown in figure 9, assuming the base address of the variable is 3000.

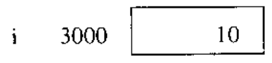


Figure 9: memory situation 1

A pointer to an integer can be defined as follows:

```
int *p;
```

and it can point to `i` by assigning to it the address of the variable. This value can be obtained by using the `&` operator:

```
p = &i;
```

and the situation in memory will be as is shown in figure 10.

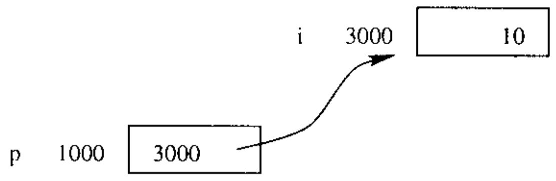


Figure 10: memory situation 2

The value pointed by a pointer can be accessed by using the operator `*`. We can print the value pointed by a pointer, and modify it by executing:

```
printf("value pointed by p = %d\n",*p);  /* prints 10 */
*p = 5;
printf("value of i = %d\n",i);           /* prints 5 */
```

8 Pointers as parameters

We have seen that C copies the values of the actual arguments into the formal parameters of the function when it is called. It is not possible for a function to modify the arguments, so a function that swaps the values of the arguments cannot be defined.

To be able to remove this restriction, the addresses of the arguments can be passed as parameters. In figure 11 the code for a function that swaps the values of the arguments is shown.

The memory situation when the function is called is depicted in figure 12.

Note that the function defines the parameters as pointers to integers. The addresses of the variables are passed by *value*, so they can not be modified. But this is not important. We want to use them to be able to interchange the values of the original variables.

The standard library function `scanf` can be used to read from the standard input. The first argument is a format string which gives information on the external representation of the data (similar to the one used in `printf`).

```
void swap(int *x,int *y) {
  int temp = *x;
  *x = *y;
  *y = temp;
}

int main() {
  int a = 2,b = 5;
  swap(&a,&b);
}
```

Figure 11: pointers as arguments

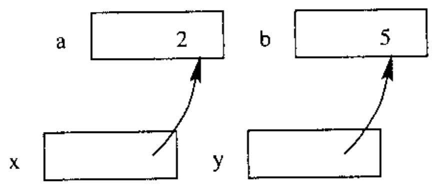


Figure 12: parameters when swap is called

The next arguments are the addresses of the variables where the input values must be stored. For example, to read two integers and one float value from standard input, we can do:

```
int i,j;
float f;

scanf("%d %d %f",&i,&j,&f);
```

We must pass the address of the variables. This is the only way in which the `scanf` function will be able to store the values.

9 Pointers to structures

It is also possible to assign the address of structures to pointers. For example, if we declare a structure of type Point (declared in section 6):

```
Point s = { 2.0 , 3.0 };
```

and a pointer to Point structures:

```
Point *p;
```

We can make the pointer p to point to s by executing:

```
p = &s;
```

so the situation in memory may now be as is shown in figure 13.

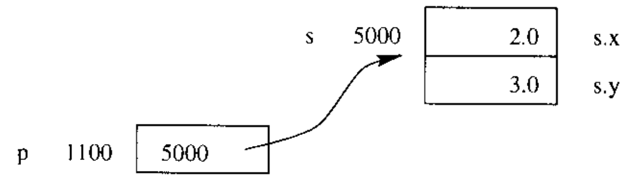


Figure 13: pointer to a structure

To modify a field of the structure by using the pointer, we can write:

```
(*p).y = 8.0;
```

The parenthesis can not be omitted, since the dot operator has a higher precedence than the asterisk operator. The same behavior of these two operators can be obtained with the operator ->. So, we can write:

```
p->y = 8.0;
```

10 Program structure

We have seen that a program consists of a set of functions. The variables we have used so far were all local variables to these functions. When the program is not executing statements in a function, these local variables do not even exist. Space is created for them when the function is called. This space is deallocated when the function is abandoned. These variables have an *automatic* storage class.

Local variables can be defined in such a way that the values they can have will still remain between calls, even if they can not be accessed when the statements of the function are not under execution. These variables have an *static* storage class. See the example in figure 14.

This program has a function f that receives no arguments and has no return value (this is the meaning of the void key word). It defines in lines *3*

```
1.  #include <stdio.h>
2.  void f() {
3.    int a = 0;
4.    static int b = 0;

5.    printf("a = %d b = %d\n",a++,b++);
6.  }

7.  int main() {
8.    f();
9.    f();
10.   exit(0);
11.  }
```

Figure 14: storage classes

and *4* two local variables named a and b initialized to 0. a has an automatic storage class, and b has a static storage class. This means that a is initialized every time the function is called. b is initialized just the first time the function is called, and the value is maintained through successive function calls. So, the values printed by the program are:

```
a = 0    b = 0
a = 0    b = 1
```

We have defined only variables that are local to functions. It is possible to define variables that can be accessed in more than one function, and also local to some compound statement.

Large C programs usually consist of several source files. They are compiled separately, and the object files are combined into one executable program. C provides the possibility that variables and functions defined in one module can be used in another one. They are called *external.*

This is illustrated in figure 15. The example is not meaningful, but it shows the different possibilities.

In module `one.c`, two variables are declared outside the scope of the functions. The variable b is `static`. This means it can be accessed in all functions, but in the same file in which it is defined (it is called *file scope*). The variable a is an extern variable, and can be accessed in the file in which it is defined, and also in all the files in which it is declared (*program scope*).

```
1.  /* module one.c */
2.  int a;
3.  static float b;

4.  int main() {
5.    int f(int,float);
6.    extern float g(int);
7.    b = 3.9 + g(2);
8.    a = f(2,b);
9.    exit(0);
10. }

11. int f(int x,float y) {
12.   return a + b + x;
13. }

14. /* module two.c */
15. extern int a;
16. extern int f(int,float);

17. float g(int x) {
18.   return (x + a + f(x,3.1)) / 2.0;
19. }
```

Figure 15: scope

The definition is in line *2* and a declaration is in line *15*. A declaration just specifies the attributes, and a definition does the same thing, but it also allocates memory space. An external variable has only one definition, but it can have several declarations.

The declaration in line *15* allows the variable a from module `one.c` to be accessed in module `two.c`.

In order to access the function g in module `one.c`, a declaration is provided in line *6*. As it is a local declaration, the function g can be called just from the `main` function.

The line *5* contains a declaration of the function `f`, which is defined later in the same file. This declaration is called a prototype and is required every time we want to call a function that is defined later in the file.

The declaration of the function `f` in line *16* allows this function to be called from functions in the file `two.c`.

11 Bibliography

Brian W. Kernighan and Dennis M. Ritchie. *The C programming language.* Prentice Hall, 1988.

Leendert Ammeraal. *C for programmers.* John Wiley and Sons Ltd., 1986.

W. Richard Stevens. *Advanced programming in the Unix Environment.* Addison Wesley Professional Computing Series, 1992.

Tim Love. *ANSI C for Programmers on Unix Systems.* Cambridge University Engineering Department.

Available on ftp::/svr-ftp.eng.cam.ac.uk:misc/love_C.ps.Z.

Advanced C

Sixth College on Microprocessor-based Real-time Systems in Physics

Abdus Salam ICTP, Trieste. October 9–November 3, 2000

Carlos Kavka
Departamento de Informática
Universidad Nacional de San Luis
San Luis
Argentina.

email: ckavka@unsl.edu.ar

Abstract

In this chapter, we will cover some more advanced characteristics of the C programming language.

1 Pointers and Arrays

The relation between pointers and arrays in C is quite strong. All operations that could be defined with arrays can also be implemented by using pointers.

Let us define an array of integers, and a pointer to integer:

```
int a[5] = { 7 , 4 , 9 , 11 , 8 };
int *p;
```

After the assignment

```
p = &a[0];
```

the pointer p points to the beginning of the array a. This could have been done also by:

```
p = a;
```

because the name of the array represents also a pointer to the first element a[0].

The situation in memory may now be as follows:

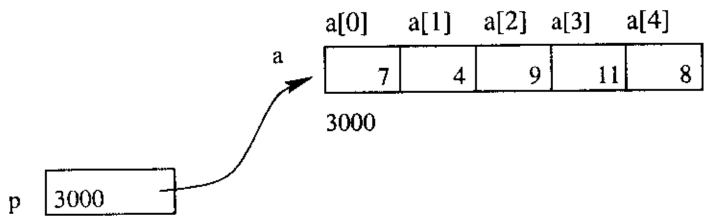


Figure 1: a pointer to an array

The value of the first component of the array can be assigned into an integer variable x by using an index i

```
x = a[0]
```
(where i = 0 in this case)

or through the pointer

```
x = *p;
```

It is allowed to add an integer constant to a pointer. By definition, if a pointer p points to a component of an array a, p+i points to i components after p. See figure 2.

The value of the fourth component of the array a can be assigned into the integer variable x by using the index

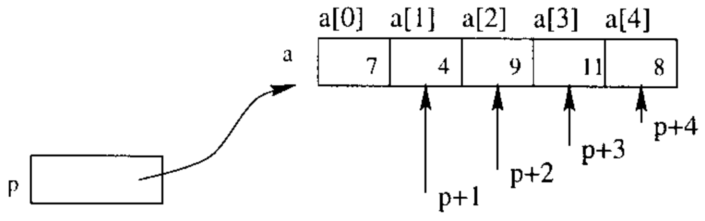


Figure 2: adding constants to p

```
x = a[3];
```

or by using the pointer

```
x = *(p+3);
```

If a pointer `p` points to the beginning of the array `a`, then `*(p+i)` is equivalent to `a[i]`. Note that this is also valid for the name of the array, so `*(a+i)` is equivalent to `a[i]`.

2 Pointer arithmetic

C allows several forms of arithmetic operations with pointers, and this is one of the distinctive features of the language.

We have seen in the previous section that it is possible to add a constant to a pointer that points to an array. Likewise, subtraction is permissible. When a constant `i` is added (subtracted) to a pointer `p`, `p` is moved ahead (back) in the array `i` positions, without considering the size of the components.

Let us, for example, assume that we have declared an array of structures:

```
typedef struct {
  int x;
  int y;
} Point;
Point a[4];
```

and two pointers to this kind of structure:

```
Point *p,*q;
```

These pointers can point to some components in the array:

```
p = &a[1];
q = &a[3];
```

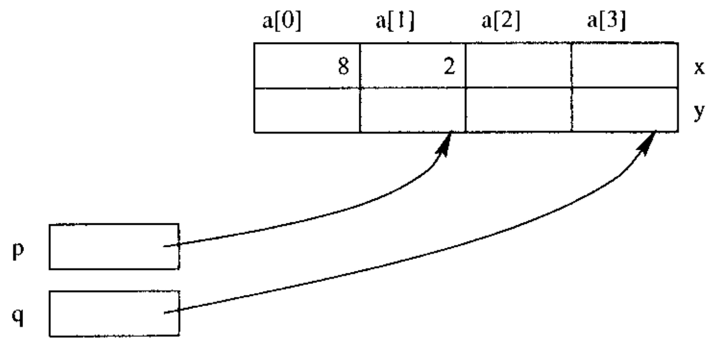


Figure 3: pointers to array of structures

as can be seen in figure 3.

It is possible to access the components of the structure pointed by a pointer using the pointer operator, for example,

```
p->x = 2;
```

which in this case, is equivalent to `a[1].x = 2`.

By subtracting the constant 3 from the pointer `q`, we can access the 0th component of the array:

```
*(q - 3).x = 8;
```

Pointers to components of an array can also be compared. In the following example, the first two conditions evaluate to true (1) and the last one to false (0):

```
p < q
p != q
p >= q
```

Pointers can also be modified. As an example, if we execute:

```
  p = p - 1;   (or p--)
```

`p` will now point to the previous component in the array, as is shown in figure 4.

The subtraction of pointers is also valid, and it produces an integer that represents the number of components between the two pointers. As an example,

```
   q - p    returns 3
```

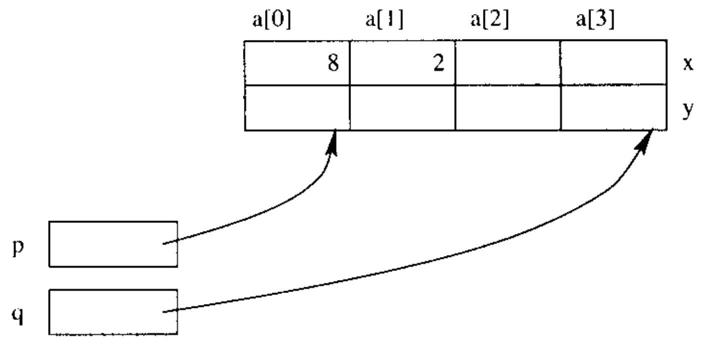


Figure 4: after `p--`

```
p - q    returns -3
```

To initialize all the fields of the structures of the array `a` to 0, we can execute:

```
for(p = &a;p <= &a[3];p++) {
  p->x = 0;
  p->y = 0;
}
```

3 Pointers to void

A pointer to `void` is called a *generic pointer*, and it can point to objects of any type. We have defined in the previous sections just pointers that point to an object of a specified type. Let us see an example:

```
int i;
float f[5];

void *p,*q;
p = (void*)&i;
q = (void*)&f[3];
```

In this example, two generic pointers `p` and `q` are defined. The pointer `p` is pointed to the integer variable `i`, and `q` is pointed to a component of the array `f` of type float. Note that we must use the cast operator to explicitly convert the types.

These pointers can point to objects of any type. However, there are some operations that cannot be done with these pointers. For example, it is not possible to add a constant to a generic pointer. The reason is that the compiler does not know the size of the object pointed by the pointer. So, for example,

`q++` cannot be done

although, with the appropriate cast, the following operation can be done:

`(float*)q++` is legal.

As another example, to print the integer value pointed to by `p`, we can do:

```
printf("%d\n",*((int*)p));
```

4 Strings

A string is represented in C as an array of characters. The end of the string is denoted by a *null* character, which is written as `'\0'`. So, one extra byte is needed to represent the string.

As an example,

```
char str1[] = "C is nice";
```

will define an array of 10 elements, as is shown in figure 5. Note that the size of the array is obtained from the length of the string plus one byte for the null character. If we had defined a longer array, the extra space will remain uninitialized.

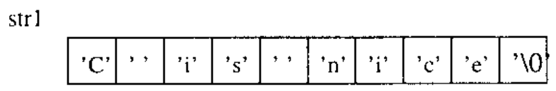


Figure 5: our first string

The name of the array can be considered as a pointer. However, there is a significant difference if we define a string like this

```
char *str2 = "C is nice";
```

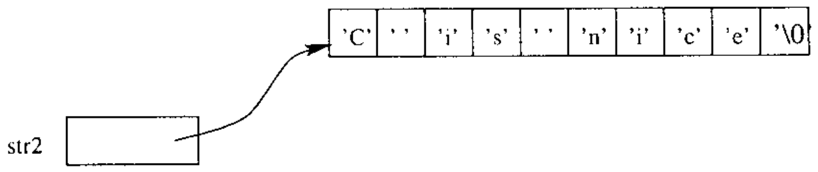


Figure 6: our second string

In this case, we obtain a real pointer and an array, as is shown in figure 6.

In both cases, references to individual characters can be done, by using both the notation of pointers or with indexes:

`str1[2]`	is equivalent to	`*(str1+2)`
`str2[2]`	is equivalent to	`*(str2+2)`

However, an important difference is that the name of the array is a constant pointer, so it cannot be modified:

`str1++`	is not allowed, and
`str2++`	advances the pointer by one position.

C does not provide operators that work with whole strings. As an example, if we have two strings `s1` and `s2`, we would like to execute `s1 = s2` to assign strings. This is not possible, because they are pointers, and we would have just copied the addresses. We must copy the characters one by one, by using a loop. The next function `strcpy` allows us to do this.

```
void strcpy(char *s1,char *s2) {
  while (*s1++ = *s2++);
}
```

The following is a situation in which the function `strcpy` can be used:

```
char a[12];
char b[] = "C is nice";

strcpy(a,b);
```

Note that as we are passing the name of the arrays as arguments, we are really passing the addresses of these arrays as arguments. The figure 7 shows the situation when the function `strcpy` is just called.

The code of the `strcpy` function is extremely compact and efficient, and it could be intimidating. The `while` loop has no body. This means that the condition will be evaluated, until it becomes false. Note that the condition is an assignment expression,

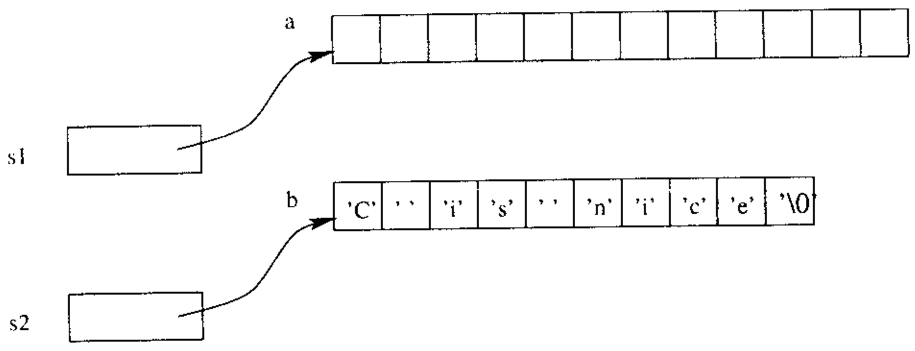


Figure 7: just to execute `strcpy`

```
*s1++ = *s2++
```

so the assigned value will be used to determine if the condition is true or not: if this value is 0, the condition will be considered false, and true if it is different from 0.

In the assignment expression, the right hand side is considered first:

```
*s2++
```

The ++ is executed first, so the pointer `s2` is advanced to the next position. However, it is a *post*-increment operation, this means, that it returns the pointer as it was before the operation. This value is de-referenced with the * operation. In the example, the first time this expression is evaluated, `s2` will point to the position b[1], and the character obtained will be `'C'`.

On the left side:

```
*s1++
```

the process is the same. The character is assigned to the position pointed to by `s1`, and the pointer is advanced to the next position. The process is repeated until the character `'\0'` is copied. In this last case, the value returned by the assignment will be 0, and the condition will be evaluated to false. This situation can be seen in figure 8.

Note that the target string must have enough space to contain the characters to be copied from the source string.

5 Library functions for strings

There exist many functions that work with strings in the standard library. We will go through some of them.

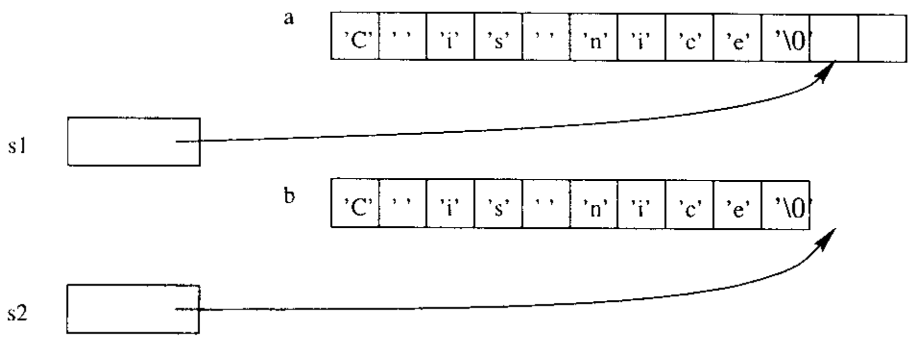


Figure 8: just to return from `strcpy`

- `void strcpy(char *s1,char *s2)`

  We have already seen the operation of the function `strcpy` in the previous example. It copies all the characters pointed to by `s2` to the area pointed to by `s1`, until the null character is copied. There must be enough space for them on the area pointed to by `s1`. As an example:

  ```
  char a[12];
  char b[] = "C is nice";

  strcpy(a,b);
  printf("a: %s\n",a);
  ```

  will print:

  ```
  a: C is nice
  ```

- `void strcat(char *s1,char *s2)`

  This function concatenates the characters pointed to by `s2` to the string pointed to by `s1`. There must be enough space in the area pointed by `s1` to store the characters from both strings. As an example:

  ```
  char a[12];

  strcpy(a,"C is ");
  strcat(a,"nice");
  printf("a: %s\n",a);
  ```

  will print

```
a: C is nice
```

- `int strcmp(char *s1,char *s2)`

  This function allows to compare lexicographically strings `s1` and `s2`. It returns 0 if both strings are equal, a negative value if `s1` is before `s2` and a positive value if `s1` is after `s2`.

- `int strlen(char *s1)`

  This function will return the number of characters in the string `s1` without considering the null character. As an example:

  ```
  char a[] = "C is nice";

  printf("length of a: %d\n",strlen(a));
  ```

  will print

  ```
  length of a: 9
  ```

- `int sprintf(char *s,char *format,...)`

  This function works like `printf`, but the actual output goes to the string `s` instead of the standard output. The notation ... indicates that the number of arguments is variable, and in this case, it depends on the number of format specifiers in the format string. Let us see an example:

  ```
  char a[20];
  int i = 5;
  float f = 3.5;

  sprintf(a,"%d -- %f",i,f);
  printf("a: %s\n",a);
  ```

  will print

  ```
  a: 5 -- 3.5
  ```

6 Using strings

It is possible to define an array of strings, and initialize it at the same time. For example:

```
char *a[3] = { "C" , "is" , "nice" };
```

In this example, a is an array of three pointers to characters, or, an array of three strings. The memory may be as is shown in figure 9.

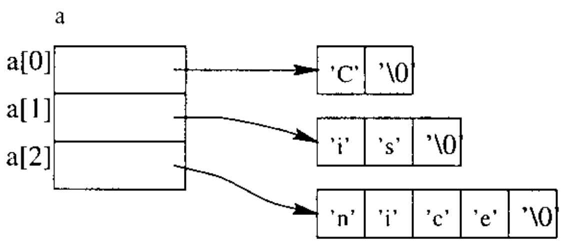


Figure 9: an array of strings

the expression `a[i]` can be used to access the i-th string. As an example:

`strlen(a[2])` returns 2.

Strings can also be used as fields in structures. For example, a structure that represents a person with his or her name and age, could be defined as follows:

```
typedef struct {
  char name[10];
  int age;
} person;
```

and created and initialized by:

```
person x;

strcpy(x.name,"John");
x.age = 30;
```

Note that we must define the string as an array of characters, and not as a pointer. If we would have defined it as a pointer to characters, there would not have been space for the characters to be copied by `strcpy`.

If we would like to reserve just the exact amount of characters needed by the name of the person, we can define the field as a pointer, and ask for memory in a dynamic way. This point will be introduced in the next section.

7 Dynamic memory administration

Dynamic memory administration is the process by which memory can be allocated and freed at any point during the execution of the program.

C provides some functions in its standard library related to dynamic memory administration. The two most important ones are:

```
void *malloc(size_t n);
void free(void *p);
```

The malloc function asks for a memory block of size n (in bytes). If n consecutive bytes are available, it returns a pointer to the first byte. Otherwise, it returns the constant NULL.

As an example, if we want to copy the string b into a, we can reserve space for the exact amount of characters, and then copy the string:

```
char *a;
char *b = "a string";

if ((a = (char*)malloc(strlen(b)+1)) == NULL) {
  printf("not enough memory\n");
  exit(1);
}
```

Note that we must consider also the null character when we ask for memory space.

Let us suppose we need to obtain space for n integers during the execution of the program. The malloc function requires the size expressed in bytes. To know how many bytes an integer uses, we can use the operator sizeof, which takes as argument the name of a type or an object, and returns its size in bytes. The following piece of code allocates dinamically space for an array of n integers, and initializes all its components to zero.

```
int *arr,n,i;

scanf("%d",&n);

if ((arr = (int*)malloc(n * sizeof(int))) == NULL) {
  printf("not enough memory\n");
  exit(1);
}
for(i = 0;i < n;i++) arr[i] = 0;
```

As another example, to reserve memory for n structures person (as defined in the previous section), we can execute:

```
person *p;

scanf("%d",&n);

if ((p = (person*)malloc(n * sizeof(person))) == NULL) {
  printf("not enough memory\n");
  exit(1);
}
for(i = 0;i < n;i++) {
  strcpy(p[i].name,"");
  p[i].age = 0;
}
```

The standard library function free, is used to return back the memory that was obtained by calling malloc. For example, to return back all the memory that was dynamically assigned in the examples in this section, we can execute:

```
free((void*)a);
free((void*)arr);
free((void*)p);
```

8 A bigger example

In section 6 we have seen that an array of strings could be defined and initialized in a very simple way. For example:

```
char *a[3] = { "C" , "is" , "nice" };
```

However, if we want to build a structure like this in a completely dynamic way, it is not so easy. Remember that in this example, a is a pointer to pointers of characters, because a is the name of an array, and the name can be considered as a pointer to the first element.

We must begin with an empty structure

```
char **b;
```

First, we need to create the array of pointers.

```
if ((b = (char**)malloc(3 * sizeof(char*))) == NULL) {
  printf("not enough memory\n");
  exit(1);
}
```

then we can ask memory for the individual strings and we are ready to copy them

```
for(i = 0;i < 3;i++) {
  if ((b[i] = (char*)malloc(strlen(a[i])+1)) == NULL) {
    printf("not enough memory\n");
    exit(1);
  }
  strcpy(b[i],a[i]);
}
```

The three steps we have followed are shown in figure 10.

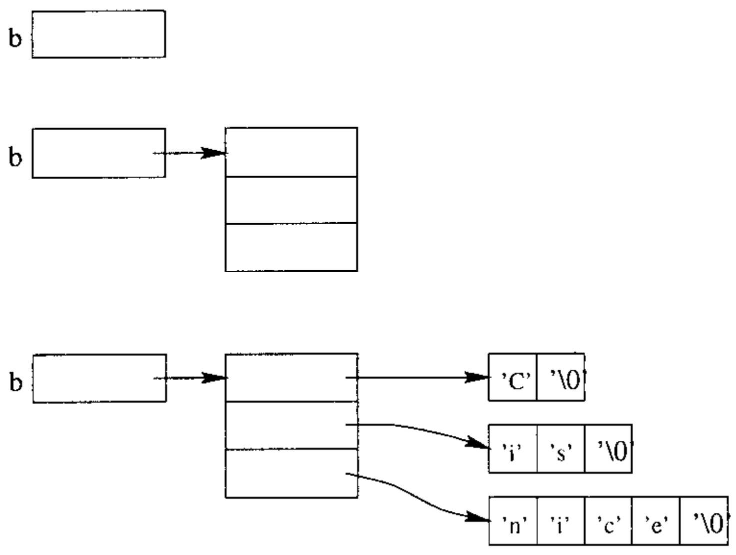


Figure 10: steps in dynamic memory allocation

To release the area that was dynamically allocated, we must follow the opposite procedure:

```
for(i = 0;i < 3;i++)
  free((void*)b[i]);
free((void*)b);
```

9 Program arguments

It is possible from the C program, to access the arguments that are passed in the command line. For example, if our program is called `program`, it can be executed from the shell prompt with a series of arguments, like for example:

```
$ program file.tex -b 123
```

The program arguments can be accessed through two parameters of the `main` function, named by convention `argc`, the argument count, and `argv`, the argument vector. `argc` is an integer, and `argv` is an array of strings, like the one we have been discussing in the previous sections. In this example, the values of the parameters are shown in figure 11.

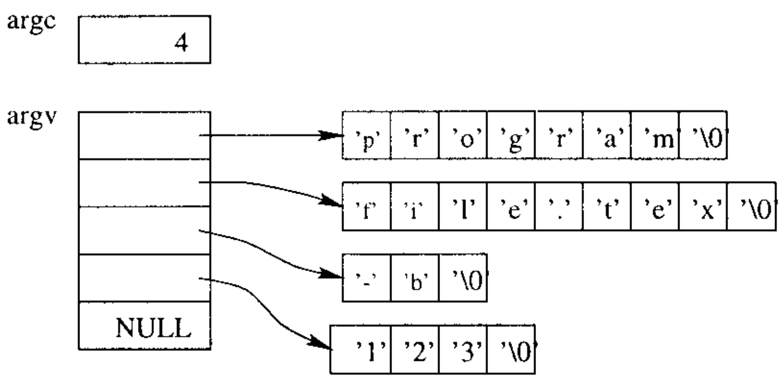

argc
4
argv
NULL
'p' 'r' 'o' 'g' 'r' 'a' 'm' '\0'
'f' 'i' 'l' 'e' '.' 't' 'e' 'x' '\0'
'-' 'b' '\0'
'1' '2' '3' '\0'

Figure 11: program arguments

The following program prints all the strings that are passed as arguments to the program.

```
#include <stdio.h>
int main(int argc,char *argv[]) {
  int i;

  for(i = 0;i < argc;i++)
    printf("%s\n",argv[i]);
  exit(0);
}
```

If we compile this program under the executable name `program`, and we execute it as it was shown before, we will obtain:

```
program
file.tex
-b
123
```

Note that the name of the program is the first string in the **argv** argument. Also the ANSI standard guarantees that argv[argc] is a NULL pointer. The notation `char *argv[]` is equivalent to `char **argv`, and can be used just when defining the arguments of a function.

We will see now an example, which will show how a program can deal with parameters as the standard Unix commands do. The program expects a file name as argument, and it has two options: `-a` and `-b`. The usual notation for this is the following:

```
program [[-a][-b]] <filename>
```

So, the program can be called, for example, as follows:

```
$ program a.tex
$ program -a a.tex
$ program -b a.tex
$ program -a -b a.tex
$ program -b -a a.tex
```

The code for the program is the following:

```
#include <stdio.h>
int main(int argc,char *argv[]) {
  int a_option = 0,b_option = 0;
  char **p_to_arg = &argv[1];

  while (--argc && (*p_to_arg)[0] == '-') {
    if ((*p_to_arg)[1] == '\0') {
      printf("invalid option\n"); exit(1);
    }
    switch((*p_to_arg)[1]) {
      case 'a': a_option = 1; break;
      case 'b': b_option = 1; break;
      default:  printf("invalid option\n"); exit(1);
    }
  p_to_arg++;
  }
```

```
    if (argc != 1) {
      printf("invalid arguments\n"); exit(1);
    }
    printf("a option: %s\n", (a_option) ? "yes" : "no");
    printf("b option: %s\n", (b_option) ? "yes" : "no");
    printf("file: %s\n",*p_to_arg);
    exit(0);
  }
```

In the program, `p_to_arg` is a pointer initialized to point to the second entry in the `argv` argument, the one that corresponds to the first program argument.

The `while` loop processes the optional arguments. In every iteration the pointer `p_to_arg` is advanced to the next argument, and the argument count is decremented. This last operation is valid because `argc` is a local variable. The condition stands for: continue iterating while there are still arguments and the first character of the current argument is a `-`. At the end of the loop, `p_to_arg` will point to the filename, if there is one in the input line.

10 Pointers to functions

The functions are also stored in memory, and in C, pointers are allowed to point to them. As an example, let us define a pointer to functions, that take two integers as arguments and return an integer value:

```
int (*p)(int,int);
```

and define two functions with these characteristics:

```
int add(int x,int y) {
  return x + y;
}

int sub(int x,int y) {
  return x - y;
}
```

The pointer `p` can point to each of them. With the expression:

```
p = add;
```

the pointer p will point to add. This function add can be called through the pointer, by de-referencing it:

```
printf("%d\n",(*p)(2,3));
```

The important point is that the function pointed to by p is evaluated. So, if we make p a pointer to the function sub

```
p = sub;
```

and we return back to execute the printf function, sub will be called.

A pointer to a function can be passed to another function as a parameter and can be used within the function to call the function which it is pointing to. It is not permitted to increment or add a constant to a function pointer.

As an example, the following function do_op receives an integer n as argument, two arrays of integers with n elements in each and a pointer to a function:

```
int do_op(int n,int x[],int y[],int (*f)(int,int)) {
  int i,sum = 0;

  for(i = 0;i < n;i++)
    sum += (*f)(x[i],y[i]);
  return sum;
}
```

This function can be invoked, for example, as follows:

```
int a[3] = { 2 , 1 , 5 };
int b[3] = { 1 , 3 , 4 };

printf("%d\n",do_op(3,a,b,add));
printf("%d\n",do_op(3,a,b,sub));
```

The first printf will print 16 (2+1 + 1+3 + 5+4), and the second 0 (2-1 + 1-3 + 5-4).

11 Linked lists

We will see now an example in which we will combine dynamic memory allocation with structures.

A linked list has a pointer to access the first node, this node contains a pointer that points to the second one, and so on. The last node contains a null pointer. Each node keeps some information. In our example, we will assume it contains an integer data.

A node is an ideal candidate to be implemented with a structure. It must contain a field to store data, and the pointer to the next node. Note that this structure is recursively defined:

```
typedef struct {
  int data;
  node *next;
} node;
```

The list will be a pointer to the first node:

```
typedef node *list;
```

A list `l` could be defined as follows.

```
list l;
```

An example of a list built with the previous structures is shown in figure 12.

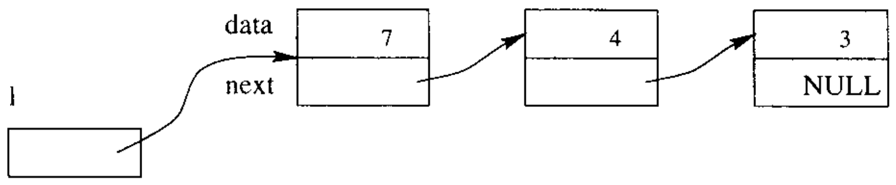


Figure 12: a linked list

A list like this could be traversed printing the integer information by using the following function:

```
void print(list l) {
  node *p;

  for(p = l;p != NULL;p = p->next)
    printf("%d\n",p->data);
}
```

The pointer p begins by pointing to the first node, and if its value is not NULL, the data field of the node pointed to by p is printed. Then p is advanced to the next node, by using the address stored in the next field. This process continues until p is NULL.

To create the list, we can define a function to create and insert a node into the list l in the position pos with a data value. Note that this function must allocate dynamically space for the node, and modify the involved pointers. If the node must be created in the first position, the pointer l that points to the first element of the list must be modified. So, its address is passed as argument (not the value).

```
void insert(list *first,int value,int pos) {
  node *p = *first,*prev = NULL;
  node *new_node;

  /* a new node must be created */
  if ((new_node = (node*)malloc(sizeof(node))) == NULL) {
    printf("not enough memory\n"); exit(1);
  }

  /* advance the pointers to reach insertion position */
  while (--pos) {
    prev = p;
    p = p->next;
  }

  if (prev == NULL) {  /* first position */
    *first = new_node;
    new_node->next = p;
  } else {             /* other position */
    prev->next = new_node;
    new_node->next = p;
  }
}
```

The pointer p is used to point to the node which is currently at the pos position. The pointer prev (previous) is used to point to the previous position. If we want to insert a node in the first position, the pointer p will point to this position, and the pointer prev will be NULL (no position to point). This situation is shown in figure 13. In this case, the pointer to the first element must be modified to point to the new first one. The new pointers are drawn with a dotted line.

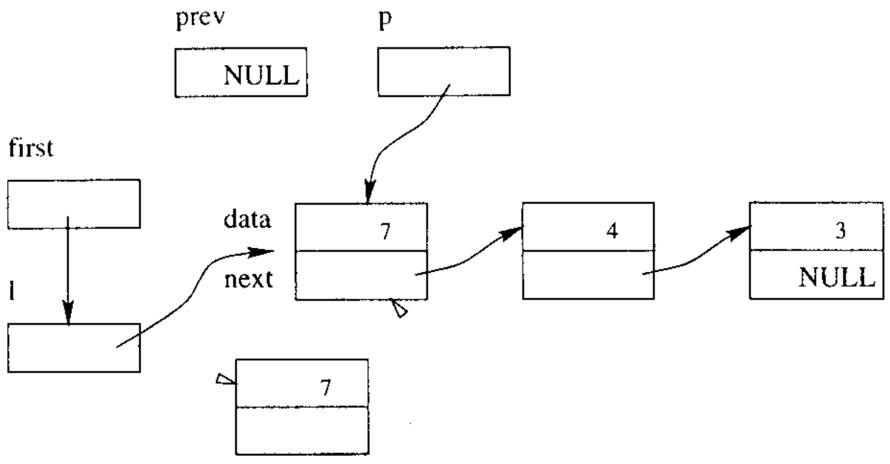


Figure 13: insertion in the first position

A situation in which the node to be inserted is not the first is shown in figure 14.

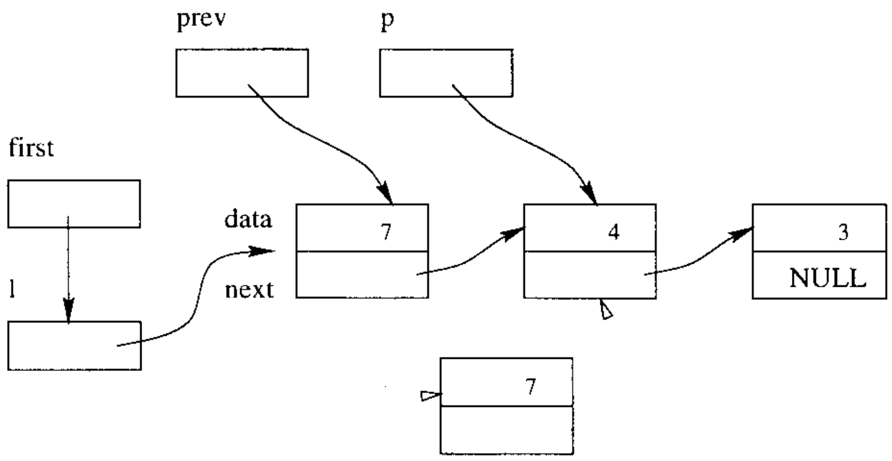


Figure 14: insertion in another position

12 Files

There are two possibilities to work with files in C. The first one is called *unbuffered I/O*. It is not part of the ANSI C standard, but it is part of POSIX.1. The term *unbuffered* refers to the fact that each *read* or *write* invokes a system call in the kernel. The other one, usually called, the *standard*

I/O routines belongs to the ANSI C standard, and provides higher level services.

12.1 Unbuffered I/O

All open files are referred to by file descriptors. A file descriptor is a non-negative integer. When we open a file, the kernel returns a file descriptor to the process. When we want to read or write a file, we identify the file with this integer value.

By convention the Unix shell associates the file descriptor 0 to standard input, file descriptor 1 to standard output and file descriptor 2 to standard error. In POSIX.1 these numbers are replaced by the constants `STDIN_FILENO`, `STDOUT_FILENO` and `STDERR_FILENO`.

The functions available for file I/O are five: `open`, `read`, `write`, `lseek` and `close`. We will look at them right now:

- `int open(char *pathname,int oflag)`

  The `pathname` is the name of the file to open or create. The value to be passed to the argument `oflag` is obtained from one of the following constants:

`O_RDONLY`	Open for reading only.
`O_WRONLY`	Open for writing only
`O_RDWR`	Open for reading and writing

  optionally OR'ed with constants from the following set (Not all the possibilities are shown):

`O_APPEND`	Append to the end of file on each write.
`O_CREAT`	Create the file if it does not exist. This option requires a third argument specifying the access permission bits of the new file.
`O_TRUNC`	If the file exists, and its open mode allows write operations, truncate its length to 0.
`O_NONBLOCK`	Sets the non blocking mode.

- `int close(int filedes)`

  Close the file with file descriptor `filedes`.

- `off_t lseek(int filedes,off_t offset,int whence)`

  Every open file has an associated 'current file offset'. It is a non negative integer that measures the number of bytes from the beginning of the file. The interpretation of the `offset` argument depends on the value of the `whence` argument.

  - If `whence` is `SEEK_SET`, the offset of the file is set to `offset` bytes from the beginning of the file.
  - If `whence` is `SEEK_CUR`, the offset of the file is set to its current value plus the offset. The offset can be positive or negative.
  - If `whence` is `SEEK_END`, the offset of the file is set to the size of the file plus the `offset`. The offset can be positive or negative.

  The offset of the file can be greater than the current size, in which case, the next `write` to the file will extend it.

- `ssize_t read(int filedes,void *buff,size_t nbytes)`

  This function read `nbytes` from the file and store them in memory beginning at the address pointed to by `buff`. It returns the number of bytes successfully read.

- `ssize_t write(int filedes,void *buff,size_t nbytes)`

  This function writes `nbytes` from the address pointed to by `buff` to the file. It returns the number of bytes successfully written.

12.2 Some examples

Let us see some examples. The following program, opens a file named `file.data`, writes a string and closes it.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int main() {
  int fd;
  char *str = "some data";
  int n = strlen(str)+1;

  if ((fd = open("file.data",O_WRONLY | O_CREAT | O_TRUNC,
                                          S_IRUSR | S_IWUSR)) < 0) {
      perror("can not open");
      exit(1);
    }

    if (write(fd,str,n) != n) {
      perror("can not write");
      exit(1);
    }

    exit(0);
  }
```

The file is opened only for writing; it is truncated to zero length if it existed, and if it is created, permissions to read and write are granted to the user. The function `perror` prints the string it receives as argument and then print the system error message. Note that a `close` is not necessary at the end, because all files are closed automatically when the program exits.

The following program can read from the file just created. Ten characters are read from the file to the area pointed to by `str`. Note that we must have enough space for the characters read.

```
  #include <sys/types.h>
  #include <sys/stat.h>
  #include <fcntl.h>

  int main() {
    int fd;
    char str[10];

    if ((fd = open("file.data",O_RDONLY)) < 0) {
      perror("can not open");
      exit(1);
    }

    if (read(fd,str,10) != 10) {
      perror("can not read");
      exit(1);
    }

    exit(0);
  }
```

The following program can create an empty file with 1KB size:

```
#include <sys/types.h>
#include <unistd.h>
#include <sys/stat.h>
#include <fcntl.h>

int main() {
  int fd;
  char c = '\0';

  if ((fd = open("file.data",O_WRONLY | O_CREAT | O_TRUNC,
                             S_IRUSR | S_IWUSR)) < 0) {
    perror("can not open");
    exit(1);
  }

  if(lseek(fd,1024,SEEK_END) != 1024) {
    perror("can not seek");
    exit(1);
  }
  if (write(fd,&c,1) != 1) {
    perror("can not write");
    exit(1);
  }

  exit(0);
}
```

12.3 Standard I/O library

The ANSI C standard I/O library handles details such as buffering allocation and performing I/O in optimal-sized chunks.

When we open a file, the standard I/O function returns a pointer to a `FILE` object. This object contains all the information required by the other library functions. We never use this structure directly. We pass a pointer to this structure to the other standard functions, in the same way as we were using the file descriptors before.

Some of the functions available are described now. Note that just a short description is presented. For more information, please refer to man pages.

- `FILE *fopen(char *pathname,char *type)`

This function opens the file `pathname`. `type` could be one of the following values:

"r" open for reading.

"w" open for writing.

"a" append; open for writing at the end of the file, or create for writing.

"r+" open for reading and writing.

"w+" truncate to 0 length or create for reading and writing.

"a+" open or create for reading and writing at the end of file.

If the file can be successfully opened, a pointer to a `FILE` structure is returned. If there is a problem, a `NULL` pointer is returned.

- `int fclose(FILE *fp)`

  Closes the file specified by `fp`.

- `int fgetc(FILE *fp)`

  Reads one character from the file specified by `fp`. `getc` is equivalent to `fgetc`, but it is implemented as a macro. It returns `EOF` to indicate an error condition.

- `int ungetc(int c,FILE *fp)`

  Push back the character `c` into the stream specified by `fp`. This means that it will be available on a subsequent reading.

- `int fputc(int c,FILE *fp)`

  Write the character `c` in the file specified by `fp`. `putc` is equivalent to `fputc`, but it is defined as a macro. It returns `EOF` to indicate an error condition.

- `char *fgets(char *buff,int n,FILE *fp)`

  This function reads at most `n-1` characters into the area pointed to by `buff` from the file specified by `fp`. The reading is stopped after an EOF or a newline. It returns `NULL` to indicate an error condition.

- `int fputs(char *str,FILE *fp)`

  This function writes the string pointed to by `str` to the file specified by `fp`. It returns `EOF` to indicate an error condition.

- `int fflush(FILE *fp)`

  This function causes any unwritten data to be passed to the kernel. If `fp` is NULL, all output streams are flushed.

  The output cannot be directly followed by input without an intervening `fflush` or `fseek`. The same is true in the other way.

12.4 Some examples

Let us see some examples. The following program opens a file and writes three strings.

```
#include <stdio.h>
int main() {
  FILE *fp;
  char *str[] = { "one" , "two" , "three" };
  int i;

  if ((fp = fopen("file.data","w")) == NULL) {
    perror("can not open");
    exit(1);
  }

  for(i = 0;i < 3;i++)
    if (fputs(str[i],fp) == EOF) {
      printf("can not write");
      exit(1);
    }
  exit(0);
}
```

There exists three predefined file pointers: `stdin`, `stdout` and `stderr`. These pointers can be used with all the functions shown here. As an example, the following program copies its standard input into its standard output:

```
#include <stdio.h>
int main() {
  int c;

  while ((c = fgetc(stdin)) != EOF)
    if (fputc(c,stdout) == EOF) {
      perror("can not write");
      exit(1);
    }
  exit(0);
}
```

12.5 Binary I/O

Most of the functions shown operate with one character at the time or one line at a time. If we are doing binary I/O we would like to read or write an entire structure at a time. Two functions are provided for this purpose:

- `size_t fread(void *ptr,size_t size,size_t nobj,FILE *fp)`

  Reads `nobj` objects of size `size` from the file specified by `fp` and stores them in memory starting at the address pointed to by `ptr`. It returns the number of objects successfully read.

- `size_t fwrite(void *ptr,size_t size,size_t nobj,FILE *fp)`

  Writes `nobj` objects of size `size` into the file specified by `fp` copied from the memory address pointed to by `ptr`. It returns the number of objects successfully written.

We can write the elements 3 through 6 of a floating point array into a file by executing:

```
float data[10];

if (fwrite(&data[3],sizeof(float),4,fp) != 4)
  perror("can not write");
```

or write a complete structure as follows:

```
struct person {
  char name[10];
  int age;
};
struct person x;

if (fwrite(&x,sizeof(x),1,fp) != 1)
  perror("can not write");
```

There are two functions related to the file position:

- `long ftell(FILE *fp)`

  Returns the current file position.

- `int fseek(FILE *fp,long offset,int whence)`

  with the same semantic of the `lseek` function.

12.6 Formatted I/O

The formatted I/O functions allow to read or write from a file in a similar way as `scanf` and `printf` work with standard input and output. The information is always written in *ASCII* code into the file. The great advantage is that they are simple to use, and the files can be read directly with a text editor. The disadvantage is that files are usually bigger.

- `int fprintf(FILE *fp,char *format,...)`

  It works like `printf`, but the output goes to the file specified by `fp`.

- `int fscanf(FILE *fp,char *format,...)`

  It works like `scanf`, but the input data comes from the file specified by `fp`.

As an example, the following program writes into a file called `numbers` the integers from 0 to 5.

```
#include <stdio.h>

int main() {
  FILE *fp;
  int i;

  if ((fp = fopen("numbers","w")) == NULL) {
    perror("can not open");
    exit(1);
  }

  for(i = 0;i < 6;i++)
    if (fprintf(fp,"%d\n",i) != 1) {
      perror("can not write");
      exit(1);
    }
  exit(0);
}
```

If we list the contents of this file, we will see the numbers.

```
$ cat numbers
  0
  1
  2
  3
  4
  5
```

13 Bibliography

Brian W. Kernighan and Dennis M. Ritchie. *The C programming language.* Prentice Hall, 1988.

Leendert Ammeraal. *C for programmers.* John Wiley and Sons Ltd., 1986.

W. Richard Stevens. *Advanced programming in the Unix Environment.* Addison Wesley Professional Computing Series, 1992.

Tim Love. *ANSI C for Programmers on Unix Systems.* Cambridge University Engineering Department.

Available on ftp::/svr-ftp.eng.cam.ac.uk:misc/love_C.ps.Z.

Embedded Systems

Sixth College on Microprocessor-based Real-time Systems in Physics

Abdus Salam ICTP, Trieste, October 9–November 3, 2000

Chu Suan Ang
Kuala Lumpur
Malaysia

email: csang@pc.jaring.my

Abstract

A cursory survey of embedded systems is first given. Embedded system development in both software and hardware is then introduced. This is followed by examples of embedded processors suitable for small and medium scale embedded system applications.

1 Introduction

Embedded systems have been around since the early days of computers. When a chemical plant used an IBM mainframe computer for process control in the 1960s, the mainframe was really an embedded processor, albeit a big and expensive one. When a physicist used a PDP11 minicomputer in the '70s to control and monitor his cryogenics experiments, he had built an embedded system. However, in those days, the number of such systems was not very large, basically because of the cost of hardware. How many PDP11s can a cryogenics laboratory possess?

With the advent of microprocessors/microcontrollers and their prices tumbling down in recent years, there is a tremendous growth in the number of embedded systems. The cost change for embedded controller is phenomenal in the last two decades - from $10,000 in 1970s to $10 in 1990s which is three orders of magnitude change. Based on the well-known fact that an order of magnitude change of price would have large impact on its use and importance, one can see that *embedded systems* will proliferate virtually every where. The subject of *embedded systems* is now a prominent one, at least in the Internet! A recent Internet **infoseek** search on 'embedded systems' produces 226,063 entries! With such a vast amount of information available, this short series of lectures can at best only give a cursory introduction to the subject.

1.1 What are Embedded Systems?

An embedded system is one with a built-in or embedded processor or computer, typically for carrying out some kind of real-time applications. The computer in such a system is not used as a general purpose computing machine. An embedded processor may or may not have a standard keyboard and video monitor, but it will always have some kind of connection to the *outside world* be it a synchrotron, an air-conditioner or a handphone. While it is possible to cite many examples for which the time of response is not critical, there are far more applications of embedded systems which are time critical. Thus the study of real-time aspects of embedded systems becomes an important issue - which is what this college is all about.

It is the application rather than the hardware itself that defines the embedded system. A PC used as a general purpose computer, as those in the computer room and in your office or home is not an embedded system. The same type of PC used in the laboratory to log data or control thus forming an integrated equipment is an embedded processor. Peripheral interface will be used, but then again, in a simple case, it may involve only the standard serial (COM Port) and parallel (Printer Port) interface of the PC.

There are numerous examples of embedded systems around us. Basically the ubiquitous embedded processors can be found in a large number of applications and situations:

- **Laboratory** - test equipment, data acquisition systems, control systems, dedicated equipment. The use of embedded systems in laboratories has been going on for a long time. In '60s and '70s researchers in laboratories used minicomputers as embedded processors. Now standard PC and microcontrollers are typically used. Test and laboratory equipment manufacturers are among the first major users of microprocessors in embedded systems. The predecessor of this Real-time College was a college on the use of microprocessors in embedded systems in laboratories.

- **Process industry** - process control systems. This is the grand daddy of real-time embedded systems. Early examples are the closed-loop control system at a Texaco refinery in Texas in 1959 and a similar system at a Monsanto Chemical Company ammonia plant in Louisiana. As the industry is able to pay, they are the ones that use mainframe computers as embedded processors. It is interesting to note that the use of computers in the process industry more or less charts out the history of computer engineering and computer science. Practically all the hardware and software techniques have been used by this industry in one way or the other.

- **Manufacturing industry** - production line assembly equipment, automatic test equipment, robots. Manufacturing industry benefits tremendously from embedded processors especially in the area of automation or robotics. Without the use of embedded systems, you would not be paying the current price of about $1000 for your PC which is really more powerful than a minicomputer of the '70s, let alone the ENIAC (Pennsylvania, 1945, 19,000 vacuum tubes, 200kW, 10 decimal digits, 0.2 ms addition, 2.8 ms multiplication.) or the EDSAC (Cambridge, 1949, 3,800 vacuum tubes, 500kHz mercury delay lines, 256 words, 35 bits, 1.5 ms addition, 6 ms multiplication.)! In 1996, assembly plants in Malaysia, Mexico, Philippines, Thailand, China and other countries are churning out more than 3 billions microcontroller ICs worth more than 10 billion dollars! This is only possible when large amount of embedded systems with clever software are used in the assembly and production lines.

- **Automotive** - engine controls, anti-lock braking, lamp, indicator and other controls. It turns out that the automotive industry is one of the most important customers of the embedded processors. In 1996, the average amount spent by a car manufacturer on a car in microelectronics is more than one thousand dollars. This industry stipulates high requirements; electronics used must be highly reliable while able to withstand severe conditions of temperature, vibration and electromagnetic interference. Some processors were initially specifically designed for the automotive industry and latter only modified for general purpose use.

- **Consumer goods** - audio-visual equipment, household electronics (microwave ovens, washing machines, dishwashers, air-conditioners), electronic toys and gadgets, etc. The list of products in this category is very large and is expanding continuously as the costs of embedded controllers drop. It is inconceivable now to operate a new television set without an IR remote controller. This is of course easily made possible when the price of 4-bit microcontrollers drops to a dollar each. (Whether one needs a remote controller to turn on a channel is a different story.)

- **Office & banking equipment** – autotellers, counting machines, weighing machines, photocopiers, fax machines. In many parts of the world, fax machine is an essential equipment in running a business or operating an office. It speeds up business transactions significantly. While e-mail is taking over facsimile service in many situations, the latter is still an essential piece of office equipment. (I had to send my accommodation form to ICTP housing section by fax from Kuala Lumpur.) Modern banking equipment are of course using a large number of embedded processors, ranging from the very powerful one in autoteller machines to simpler ones in currency notes counters and others.

- **Computer peripherals** - printers, keyboards, visual display units, modems. A computer system consists of a number of peripheral devices besides the CPU box. Peripheral devices inevitably use embedded processors to either reduce cost or enhance performance. As the volume of PCs produced is no longer trivial, the use of embedded processors in their peripheral devices cannot be overlooked either.

- **Telecommunications** - pagers, telephones, wireless phones, handphones. This is yet another major area of embedded processor application. With the rapid growth in the telecommunications especially in the area of cellular phone, the telecommunications manufacturers have been pushing the advancement of embedded processors in terms of size, cost and complexity. With the requirement of integrating analogue and digital circuitry, they are encouraging the chip designer and manufacturer to push towards the limits of this technology.

Although there is an infinite variety of embedded systems, the principles of operation, system components and design methodologies are essentially the same. A typical system consists of a *computer* and an *interface* to the physical environment, which may be a chemical plant, a car engine or a keyboard, for example. In some applications, *standard input/output* devices such as the VDU, keyboard and printer are present, as in the case of process controller in a chemical plant. In others there are no standard I/O devices, as in the case of car fuel injection control. In the former case, it is likely that a general purpose computer such as a PC or a more powerful workstation PC will be adapted as the embedded processor. In the latter, microcontrollers designed together with dedicated electronics will be used.

We shall deal with the development of such systems in general, with emphasis on a class of embedded systems using *microcontrollers* which is currently the most prevailing form of computer used in laboratory and many other situations.

1.2 What are Real-time Embedded Systems?

It was mentioned earlier that embedded systems are typical used to carry out real-time applications. What are real-time systems? The Oxford Dictionary of Computing defines a real-time system as "Any system in which the time at which the output is produced is significant. This is usually because the input corresponds to some movement in the physical world, and the output has to relate to that same movement. The lag from input time to output time must be sufficiently small for acceptable timeliness."

The above definition covers a wide range of systems - from UNIX workstations to aircraft engine control systems. When a command is entered in a UNIX workstation, we typically get a response on the screen 'with a sufficiently small time lag'. In an aircraft engine control system, the response to commands and other input parameters has to be within certain time limits. There is however a subtle difference between the UNIX workstation and the aircraft engine control system in terms of timeliness.

An alternative definition of a real-time system can be as follows: "a real-time system receives inputs and sends outputs to the target system at times determined by the target system operational considerations - not at times limited by the capabilities of the computer system." This further defines the meaning of response time and it distinguishes between the UNIX workstation and the engine controller. In a UNIX workstation, occasionally when we issue a command, we may not get the response in a time to our liking because the CPU is running some other higher priority tasks or simply overloaded. In this case, the UNIX workstation no longer qualifies as a real-time system according to the more stringent definition mentioned above.

A real-time program is thus one for which the correctness of operation depends both on the logical results of the computation and the time at which the results are produced. The main objective of this Real-time College is to deal with the various techniques and methodologies in achieving the above.

In view of the fact that not all embedded systems require very rigid response times, real-time systems may be classified broadly into three categories:

- **Clock-based (cyclic, periodic)** - e.g. process control systems. Generally all process control related systems would require a clock-based system. The real-time program is conscious of time by means of a **system clock**. Actions are taken at the precise moments of time. When a stimulus is present or when a limit is reached the system must respond within a certain clock cycles (time).

- **Event-based and Interactive** - e.g. alarm systems, autoteller. An event based system such as an alarm system in your house generally does not have the sense of 'time'. When a contact is opened because the house is broken in, the siren is triggered or the police is notified, to within an acceptable time limit.

Strictly based on time constraints, real-time systems can be grouped into:

- **Hard real-time** - must satisfy deadlines on each and every occasion, e.g. temperature controller of a critical process.
- **Soft real-time** - occasional failure to meet deadlines acceptable, e.g. autotellers.

While real-time embedded systems have received a lot of attention in recent years, the earliest proposal of using a computer in real-time applications for controlling a plant actually dates back to 1950 when Brown and Campbell published their paper:

- Brown, G.S., Campbell, D.P., 'Instrument engineering: its growth and promise in process-control problems',*Mechanical Engineering*, 72(2): 124 (1950).

A couple of early industrial installations of embedded systems are listed below:

- September 1958 by Louisiana Power and Light Company for *plant monitoring* at a power station in Sterling, Louisiana.
- First industrial *computer control* installation was by Texaco Company for a refinery at Port Arthur in Texas in March 1959.

The above systems, as well as many other early systems were *supervisory control* systems that used steady-state optimisation calculations to determine the set points for standard analogue controllers. In other words, the digital computer was used to compute and to send simple commands to many standard analogue controllers which had been in use for a longer time in the industry. These analogue controllers were generally expensive, complicated and required periodic calibrations. Later, *direct digital control* which allowed the direct control of plant actuators was added and analogue controllers were not required.

The early real-time programs were written in *machine code* which was manageable when the tasks were well defined and the system small. However, in combining supervisory control with direct digital control, the complexity of programming increased significantly. The two tasks have very different time scales and interrupting the supervisory control is necessary. This led to the development of general purpose **real-time operation systems** and high-level languages for such systems.

2 Design and Development of Embedded Systems

There are four major steps involved in the design and development of embedded systems:

- System design.
- Design and build hardware.
- Design and develop software.
- Integrate software into target system.

For very small projects involving only one person, the above tasks are carried out sequentially in that order. However, for bigger projects, it is often possible to develop the hardware and the software in parallel. This calls for a thorough system design in the first place.

2.1 Designer's Skills

In order to carry out the task effectively, the designer of embedded system must possess several skills:

- **Good knowledge of the** *microcontroller resources.* This should include the architecture, the instruction set, the addressing modes and the on-chip resources. The knowledge should generally extend beyond the simplified and idealised devices. For example, a good designer must know how the microcontroller handles interrupts and related timing issues so as to handle real-time activities effectively.
- **Good knowledge of** *real-time control.* The real-time requirement of the target system must be clearly understood before an effective solution may be found.
- **Good knowledge of** *software techniques.* The amount of software effort needed for an embedded system often far exceed that of hardware nowadays. A good designer thus must possess good knowledge of languages, operating systems, and software building blocks in handling various requirements and tasks of the target system. Many experienced programmers found that collecting useful algorithms and software tools is very helpful for future projects.

For example, it may be an advantage to represent a system by a state machine. In this case, how can the state machine be implemented in software easily? In an embedded system where a keyboard is used, how does one handle the keyboard parsing?

- **Good knowledge** of **hardware I/O components or sub-modules.** To be able to design a good embedded system, knowledge of the state-of-the-art peripheral devices is helpful. For example, the technology of output devices including LED, LCD and CRT has progressed significantly. Manufacturers have implemented very sophisticated device drivers for some displays and it is a good idea to consider using them whenever possible.

- Many embedded systems involve the use of ADC or DAC. Again, a good knowledge of accuracy, resolution, and speed of conversion is essential. If a target system is expected to measure 1 millidegree in 100 degrees, it is useless to design a system with a 10-bit ADC, for example. Other components such as drivers, position control and position encoding are often used and should be included in the repertoire of hardware skill.

- **Good knowledge of** *development tools* . Development of embedded system requires both hardware and software development tools. Hardware tools: multimeter, oscilloscope, logic probe, pulser, EPROM programmer, logic analyzer, in-circuit emulator, development system. Software tools: editor, cross compiler, cross assembler and linker, simulator, development system.

2.2 System Design

Designing of embedded system is no different from designing any other computer based system and it is important that one applies a good design and engineering methodology. Many different approaches have been advocated and there are many books written on the subject but basically the objective is to apply a system approach so that the target system may be built to specification functionally and it is easy to maintain.

First of all, define the functions and requirements of the target system. The *problem* must be well defined. Otherwise there is no *solution.* Difficulties arise when the scope of the work is not rigidly known or when the designer is uncertain of the capabilities of the various hardware and software resources.

This may happen in the initial phases of a project and as time goes on, one must have a clear idea of all the requirements and *freeze the specifications* before embarking on the next phase of work.

In general, once the first phase is over, one can specify the *interface* to the target system clearly, for example:

- Number and type of parallel I/O needed for interacting with the target system.
- What kind of real-time requirement is needed?
- Any serial communication needed? If so, what is the distance of communication?
- Is the target system localised or distributed over a wide area?
- Any ADC and DAC requirement? If so, what are the requirements on resolution, accuracy and sampling rate?

Is it a networked or a stand-alone system? In the case of distributed or networked application, define the type of networking facility to use. This usually depends on the data rate and response time. For example,

- If the data rate requirement is kbps and below and the response time requirement is around a second, a low cost serial link based on RS232 or RS422 interfaces may be used.
- If a high data rate up to Mbps is needed, use a standard LAN-type link, Ethernet or Token Ring for example.

Specify the *user interface.* Is it an instrument panel-type interface? Or is it a *graphical user interface* (GUI)? In either case design a friendly user interface.

2.3 Choosing An Embedded Processor

When the functional requirements of an embedded system is defined, one can choose an appropriate microcontroller/microprocessor. The choice really depends on many factors, amongst them are:

- Unique functional requirements of the target system. It may be that the ADC requirement calls for a particular processor, or the temporary buffer needed dictates another. Other applications may require a microcontroller with EEPROM as a non-volatile storage.

- Production volume of the target system. A one-off laboratory embedded system may use an expensive or oversized processor whereas a system that has to be produced in quantity may be very cost sensitive. One may have to use a \$1 processor with masked ROM instead of \$50 processor with EEPROM.

- Experience of the designer.

- Availability of the devices.

- Your boss says 'use microcontroller xyz'.

Besides using a microcontroller and building the target system from scratch, there is yet another alternative - obtain or purchase general purpose embedded computers with the necessary I/O and build only the interface to the *outside world*. This is an attractive option if you can afford it. There are manufacturers producing a wide variety of embedded computers ranging from 8-bit microcontroller-based systems to full-fledged 486 PC with 1.44MB ROM disk on a single expansion card.

However, the importance of embedded system design really arose from the availability of a wide range of microcontrollers. And knowing these microcontrollers well is a necessary skill of an embedded system designer.

2.4 Microcontrollers (MCU)

If you ever wonder why we should study microcontrollers, please look at the following table of the total number and value in USD of microcontrollers shipped by manufacturers in 1996 alone:

MCU	Quantity (Millions)	Value (Million USD)
4-bit	1,100	1,800
8-bit	2,100	6,500
16-bit	200	1,600

The evolution of microprocessor has been along two different paths. One has been the development of powerful CPU with 16– and 32–bit data bus and very large memory space (e.g. gigabytes). These processors are used in personal computers and workstations which form the backbone of computing facilities in home, commercial, educational, engineering and research environments.

The power and speed of the 16–and–32–bit CPU of course do not limit them to the domain of stand-alone computers. They are used as embedded computers as well. In fact they are used in many applications where sophisticated control or high speed operation is needed, e.g. HP Laserjet printers.

However, it is true that for a large number of laboratory and other applications, the tasks can often be performed by a range of smaller processors – the 4–and–8–bit *microcontrollers.* In this short series of lectures, we shall not deal with the development of embedded systems using 16- and 32-bit CPUs because of the complexities of such systems. However, their use as cross-development tools for microcontroller-based embedded systems will be elaborated.

The second evolution path of microprocessor is along the line of microcontrollers which on a single chip the processor is integrated with RAM, ROM, EPROM, EEPROM, timers, serial and parallel I/O facilities. These microcontrollers are most suited for real-time embedded systems or used as real-time modules in large systems.

It is noted that the 8-bit microcontrollers is the main workhorse in embedded systems and this trend is likely to continue. However, the 4-bit smaller brother has its part to play too, with shipment of about half that of the 8-bit. There is really no point in putting an 8-bit MCU in a TV remote control when a 4-bit version would do the job efficiently at a lower cost. This is of course due to that fact, that more powerful microcontrollers normally require complex hardware. Cost considerations can be very important in high volume applications. The price range is wide - from low cost (~USD1) 4-bit chips to high performance 16-/32-bit chips at (USD50-100).

Choosing a microcontroller for use is not a simple task if you are a serious user because there are many manufacturers offering a wide variety of seemingly similar devices. Besides the few points mentioned earlier, one has to look at several other factors:

- Development tool and technical support. This applies to your local agent support really. It is no good to you when the catalogue lists some superb development tools at low prices but the local agent is unable to get it for you or provide the necessary technical information.

- Documentation. Can you get full data book, reference manuals, application notes?

- Does the manufacturer produce all the supporting chips? If not, are they readily available? Is there a second source for the MCU?

- Does the series have a one-time-programmable (OTP) version? What about EEPROM, and windowed EPROM?

The major suppliers of microcontrollers are: Motorola, Mitsubishi, NEC, Hitachi, Philips, Intel, SGS-Thomson, Microchip, Matshushita, Toshiba, National Semiconductor, Zilog, Texas Instruments, Siemens, and Sharp. Motorola, the leading supplier of microcontrollers, shipped more than 350 millions units in 1993 while the last in the above list shipped more than 17 million units.

We shall look at two microcontrollers in greater detail later. In this section, a brief survey of some commonly used microcontrollers is given.

- **6805 (Motorola)** - This is a popular family of microcontrollers by Motorola based loosely on the 8-bit 6800 microprocessor which has a von Neumann architecture where instructions, data, I/O and timers all share the same memory space. Some members of this family include on chip serial I/O, ADC, and PLL frequency synthesizer. There are EPROM and mask ROM versions. Expanded and single chip modes are available.

- **6811 (Motorola)** - This is another popular 8-bit microcontroller by Motorola which is more powerful than the 6805 and is a CMOS device drawing typically less than 20mA. It has most of the features and peripheral devices of a microcontroller including digital I/O ports, programmable timers, ADC, PWM generator, pulse accumulator, asynchronous and synchronous communication ports and watchdog circuit. We shall use this device to design a small embedded system in this College.

- **683xx (Motorola)** - These are high performance (32-bit) microcontrollers capable of very high processing speeds and addressing large memory space. They are produced by incorporating various peripheral devices into the 68000 family core processor. The 68331 for example has a 68020 core and about the same processing power as an Intel 80386.

- **8048, 8051 (Intel and others)** - Two very famous series of 8-bit microcontrollers by Intel. The 8048 is a first generation microcontroller and is still popular because of the wide range of software available and its low cost. The 8051 is a second generation microcontroller which rules the microcontroller world of the 8-bit class of embedded systems at the moment. It is not as orthogonal as the Motorola counterpart, but it is powerful and can be easy to program and design if you are familiar with the architecture.

The 8051 has a modified Harvard architecture with separate address spaces for program and data. The program space is 64K(bytes), with the lower 4 or 8 K residing on chip. It uses indirect addressing to access up to 64K of external data memory. It has 128 bytes of on-chip RAM (256 bytes in 8052) plus several special function registers. I/O is mapped separately into its own space as in the other Intel processors.

It has the capabilities of performing Boolean operation on bits just about anywhere in the system and then carry out relative jumps based on the results. There are large amount of software available for this microcontroller and there are many other chip manufacturers that second source this device with many different variants if the customers so desire. Finally, probably the most important of all, it is more readily available than others and perhaps cheaper than other chips in many parts of the world.

- **80C196 (Intel)** - This is a third generation Intel microcontroller featuring 16-bit operation and CMOS fabrication (though the original version 8096 is NMOS). As a high-end microcontroller, it has 40 digital I/O, high speed ADC, serial communications, 8 priority interrupts, PWM generator, watchdog timer, hardware multiplication and division.

- **80186, 80188 (Intel)** - These are the microcontroller versions of the famous 8086 and 8088 used in the PC. There are a number of variants available but they all have 2 DMA channels, 2 counters or timers, programmable interrupt controller, and dynamic RAM refresh output. The use of the same CPU as the PC means that a lot of programs are readily available and that one can use standard development tools for PC to develop applications for this microcontroller. This may cut down the learning curve drastically if one is previously familiar with the editors, assemblers and compilers in PC. Of course it is basically a very powerful processor to use.

- **80386EX (Intel)** - This is the microcontroller version of the 386 processor of Intel. As in the case of 80186 and 80188, the major advantage is compatibility with the 386 PCs. The chip has serial I/O, DMA channels, power management, counters or timers, programmable interrupt controller, and dynamic RAM refresh output. This is of course a even more powerful chip to be used as microcontroller. It is worth noting that in this case the effort of designing your own 386 microcontroller embedded system versus buying a standard ready built 386 PC as your embedded PC has to be weighed carefully. The latter may turn out to be a better solution.

- **COP400 (NS)** - This is a 4-bit microcontroller from National Semiconductor which features 512 to 2K ROM, 32 to 160 4-bit RAM with many different packaging (DIP/SO/PLCC) from 20 to 28 pins. It can operate from 2.5 to 6 volts. A wide range of applications call for this type of low end chips, especially when its price goes under 50 cents in quantity.

- **COP800 (NS)** - This is a 8-bit microcontroller from National Semiconductor which features static memory, and voltage range of 2.5V to 6V. It has a memory mapped architecture as in the Motorola series of microcontrollers.

- **HPC (NS)** - This High Performance microController family from National Semiconductor is a 16-bit chip with von Neumann architecture operating at 3.3V. It has hardware multiplication and division capabilities. Other features include HDLC for data communications, multiply/accumulate unit for low to medium DSP applications.

- **Z8 (Zilog)** - The Z8 family of microcontroller is from Zilog and is loosely related to the Z80 MPU. It has a rather unique architecture with three memory spaces for program, data and registers. Standard features include digital I/O (up to 40 lines), serial communications, timers, DMA, fast interrupts. One member has a ROM Basic. Another one (Z86C95) has 256 registers and an internal 16-bit Harvard architecture DSP. The DSP registers are accessible as additional registers. ADC and DAC are also included.

- **HD64180 (Hitachi)** - This is a microcontroller family from Hitachi that is compatible with Z80 but runs in fewer clock cycles. It has digital I/O, asynchronous and synchronous serial communication channels, timers, interrupt controls, DMA. Hardware multiplication and a few other instructions have been added.

- **TMS370 (TI)** - This microcontroller family by Texas Instruments is similar to 8051 and has large number of on-chip devices such as RAM, ROM (mask, OTP, or EEPROM), timers, watchdog, SCI, SPI, ADC and interrupts. Instructions are mostly 8 bits with a few 16-bit ones. Hardware multiplication and division included.

- **PIC (Microchip)** - This is a family of first RISC microcontrollers which is gaining popularity recently. The predecessors of this family have been around for more than 20 years under the name General Instruments. The new PIC series are fabricated in CMOS with enhanced features and more family members.

  The chip features a Harvard architecture with fewer instructions than other microcontrollers (33 for the 16C5X versus over 90 for the 8048). Simplicity in design allows more features to be added. The major advantages of this chip are small size, small pin count, low power consumption and low cost.

3 Hardware Design and Development

Once the system requirements are well defined and the type of embedded processor chosen, one can embark on the task of hardware design and development. If the choice is a standard PC or ready built hardware as the embedded processor, then the hardware design step is simplified to that of designing the interface board or circuitry to the target system. Although there can be an infinite variety of target systems, the interface requirements however can be grouped into just a few standard categories - digital I/O, analogue I/O, serial data communications and parallel data communications. Many of the interface requirements are normally provided for by the embedded processor hardware. Perhaps signal conditioning circuits (instrumentation amplifiers, precision attenuators, current drivers, etc.) are needed in the case of analogue I/O or special actuators or sensors.

We shall look at the case where the embedded processor is not already available but built. This is more likely the case for embedded system designers! Ten or fifteen years ago, one would build a microprocessor based system using a handful of chips including microprocessor, memory, peripheral devices and other glue chips. And to do that effectively, certain basic skills have to be acquired. In fact, the earlier Microprocessor College at ICTP spent four weeks trying to achieve just that.

Nowadays, we may still build microprocessor-based embedded system. The 6809 system used in the laboratory of this College is one such example. There are many good reasons for doing so. First of all, it generally has more memory resources than a single chip microcontroller. This facilitates the use of more sophisticated resident firmware including a full featured monitor or a real-time kernel, for example. Often, there are many readily available software for a popular microprocessor such as the 6809. The designer may already be familiar with a well-known microprocessor and need not learn to use a new one.

The trend however, is to use single chip microcontrollers whenever possible. The beauty of designing embedded systems using microcontrollers is the relative ease and simplicity. You no longer have to be a 20-year-experienced-electronic-engineer to be able to design the hardware. As you may be aware, the topic of embedded system in this College has been reduced to six lectures!

Whether we use microprocessors or microcontrollers, there is a set of good design rules or practices that one should adhere to. Amongst them, one that has often been over looked is that the design must incorporate facilities for debugging and testing. Small tests or diagnostics, switches or indicators, added during the designing stage cost very little, but help tremendously in the later stages.

Once the circuit design is completed, the next step is circuit board layout and fabrication. Unfortunately the hardware development process does not end there. In most cases, a certain degree of hardware testing and debugging must be done.

3.1 Outline of Hardware Test Procedure

To carry out these tasks, it would be advantageous if sophisticated tools such as development system, in-circuit emulator and logic analyser are available. However, it is possible to test and debug with the basic electronics laboratory equipment such as multimeter, oscilloscope, logic probe and function generator alone, if a systematic approach is adopted.

- Printed circuit board (PCB) inspection for track continuity and possible bridging. This is a step that is often overlooked. However, it is a vital step because easily locatable faults if left undetected, usually cause much more debugging efforts at a later stage.

- Power up the bare PCB and check voltages.

- If it is a microprocessor-based system, such as the 6809, or a microcontroller-based system operating in *expanded multiplexed* mode, test the address bus and (partially) the data and control bus on the *hardware kernel* which is the processor itself. This step is skipped if the system is single-chip, micocontroller-based.

  In the case of 6809, this is done by forcing a NOP ($12) on the data bus by pulling up D1 and D4 to 5V via resistors and grounding all other data lines. It causes the continuous execution of NOP for all memory locations. This in turn results in A0 toggling at half the system clock rate, A1 toggling at half the rate of A0 and so forth. The address bus can thus be checked easily with an oscilloscope. In this test, data bus and control bus are partially verified.

  The above test procedure is actually making use of the 1-byte instruction of the microprocessor in a unintended manner. For Z80, 8085 and 8088 similar techniques can be used. In Z80 and 8085, RST 7 ($FF) instruction is used whereas in 8088 either the 1-byte INT 3 or PUSH instructions may be similarly used.

- If a logic analyser is not available, implement a tight loop program in the EPROM or EEPROM such as a branch-to-itself loop (LOOP BRA LOOP). For 6809, this consists of two bytes ($20 $FE) and takes three machine cycles to execute. A two-byte reset vector is also needed in the ROM. The execution of this very short program can be followed cycle by cycle on an oscilloscope and thereby confirming the proper operation, at least partially, of the data and control bus.

- It is a good idea to include DIP switches and LED indicators in the hardware even if they are not required in the final target system. Test routines for I/O ports which have these input switches and output indicators can be written and tested. Commonly used routines include incrementing the binary value of the output port at a slow rate for visual inspection, reading status of switches and sending it to the output port. This stage of testing serves to verify the operation of I/O ports and to provide users with function selection. Normally on power up the system is programmed to check the status of the input switches and jump to appropriate test routines or the main program.
- Small test routines for other components in the system are then implemented. This includes testing the serial link, the timers, ADC and the memories.
- In some embedded systems where the memory is not very small, a monitor program or kernel is then implemented.
- At this stage most of the hardware testing is done and the task moves on to application software testing and debugging. However, there is one type of hardware bug which is not detected by the testing mentioned above. These are problems caused by intermittent faults, glitches or external interference. These are detected by means of logic analyser or in-circuit emulator running in surveillance mode.

3.2 Some Hardware Development Tools

While one can get by with the basic tools for small embedded system development, nevertheless it will help if a number of other hardware development tools are available, especially when one is dealing with more sizeable projects or when problems such as intermittent faults, external electromagnetic interference, and glitches arise as mentioned above. It is impossible to give a thorough treatment of various hardware tools in detail here. However, a number of more important ones are introduced below.

- **Oscilloscope** - The oscilloscope really needs no introduction other than listed here for completeness sake. It is noted that while the conventional dual-trace 20MHz cathode ray oscilloscope (CRO) is still the faithful workhorse in the lab, there exists in the market now digital oscilloscopes with liquid crystal display (LCD) at a reasonable price. Often it combines the function of a digital (memory) oscilloscope with a logic analyzer. The importance of the oscilloscope cannot be over-emphasized - after all the HP and Tektronix logic analyzer designers used their oscilloscopes to debug their embedded systems in the '70s!

- **Logic Analyzer** - The two traces of an oscilloscope is ready rather inadequate or impossible when it comes to *simultaneously* monitoring the 40 or so lines of a typical microprocessor or microcontroller circuit. Logic analyzers capture 48 or more signals and display them in multiple traces or in coded form. Being a powerful embedded system itself, the logic analyzer can perform a number of other things that expedite the debugging of embedded systems.

  It allows a trigger condition (data, address and control bus pattern) to be set up and captures the cycle by cycle information in memory (typically few thousand cycles deep) when the trigger condition is met. The captured data can be viewed as traces, in binary/hex form or in mnemonics of the target processor after being disassembled. This provides a very power tool for monitoring what's going on at a very low level non-intrusively - at least while the embedded system is running at its normal speed.

  Most logic analyzers also provide *timing analysis* whereby the traces are sampled at rates higher than the system clock and hence glitches or other irregular waveforms may be detected.

- **Emulator** - First introduced by Intel, now in-circuit emulators are used in large number of embedded system development. This tool brings the debugging of hardware one step higher than using the logic analyzer alone. Basically it not only allows the target system to be monitored, but also has the ability to stop execution in a controlled manner, change memory and register contents and resume execution. This is achieved by replacing the target system CPU with a more elaborate system typically containing the same type of CPU but having other resources which can carry out the actions mentioned above. In theory the system *emulates* all the CPU's functions in real time.

  The major features of the in-circuit emulators are breakpoint, real-time trace, RAM overlay, and performance analysis. Breakpoint setting, as mentioned above, allows us to stop execution, for example, at the end of a function and monitor the return value. When the code does not behave as expected, real-time trace can be used to *look* at what the code is doing. Embedded systems often have their code stored in ROM or EPROM. To change the code during debugging is tedious. RAM overlay is a technique to circumvent this difficulty. Instead of running the code in the target system ROM or EPROM, RAM in the emulator which can be easily modified is used. Performance analysis deals with the problem of code not able to deliver the performance required, such as keeping up with external events. The analysis allows the programmer to scrutinize the execution of his code carefully and find remedies if possible.

In the case of microprocessor-based systems, the target microprocessor is replaced by an emulating processor which has overall control over the data, address and control bus and thus the operation of the entire system. In the case of microcontroller-based systems, it is more complicated. Typically, the emulator operates the microcontroller in the expanded mode so as to gain access to the internal bus. It must also have:

- extra RAM to hold the application software during development,
- a monitor program, and
- rebuilt ports to replace those lost in the expanded mode.

Other features available in an emulator are:

- communication facility between the monitor program and a host computer,
- ability to download object code from the host computer to the target system,
- ability to display and change RAM contents and processor status of the target system,
- single stepping and breakpoint features, and
- execution of the application program at full speed.

The emulator is almost an indispensable tool in the development of embedded systems but the downside is that it is generally not cheap. Good emulator can run to tens of thousands of dollars. Fortunately there are a number of low-cost emulators typically produced by chip manufacturers themselves to promote the sales of their microcontrollers. These are often sold under the name of evaluation board of system. They lack the sophistication of full featured emulators but nevertheless are very useful for small projects.

One such example of a low-cost standalone in-circuit emulator is the M68HC11EVM designed for developing 68HC11 embedded systems. It has the following features:

- Emulate both the *single-chip* and *expanded-multiplexed* modes of operation.
- Code may be generated using the resident assembler/disassembler, or may be downloaded through a host or terminal.
- Microcontroller ROM is simulated by write-protected RAM during program execution.
- Two serial links for host and terminal communication.

The system operates in either one of two memory maps - the *monitor* map and the *user* map. Two types of memory map switching are possible. *Temporary* map switching allows modification of user memory, and *permanent* map switching allows execution of user programs.

• **ROM Emulator** - ROM emulators are like RAM overlays mentioned above, used to temporarily replace the target system firmware. A ROM emulator consists of RAM and associated circuit, a connection to the ROM socket in the target system and a link to a host computer. The host computer downloads the data into RAM which is then used by the target system as its ROM memory. This relatively simple tool is very effective in embedded system development because it reduces the iteration time significantly.

4 Software Design and Development

Software design and development for embedded systems is no difference from most other software project design and development.

- First of all write down the software specifications before anything else. Resist the temptation to start programming before the overall software design is done. How often do you see an electronic engineer grab a soldering iron the moment he has a rough specification of an amplifier to build? As far as possible, adopt a top-down approach.

- The major task in software design is the breaking up of the entire project into smaller manageable modules or components. Ideally modules and components should not be longer than 2 or 3 pages. The longer it is, the more difficult to debug. Write comments on your code, not just a few token lines haphazardly thrown in to satisfy your manager or instructor. On each routine, write a detailed header describing the algorithm, strategy, calling procedure, return value, etc. After 20 years of pleas, coaxing and threatening, I am sure we can produce better commented code.

- What programming language to use? Most people agree that one should use a high level language (HLL) to develop embedded systems. Amongst the HLLs, C is known to be a good choice for embedded systems. However, other HLL have not fallen entirely into oblivion yet. Interpretative HLLs such as BASIC and FORTH are used by some. PL/M from Intel is also being used.

- Besides knowing C, an embedded system programmer usually has to learn the assembly language as well. For very small projects, assembly language is still a good choice in view of the memory constraint. Even when one writes in C, a small amount of code such as the interrupt routines and sometimes the device drivers are still implemented in assembly language. Source code debugging is nice, but occasionally, one may have to debug at a lower level, especially when hardware debugger such as logic analyzer is used. In which case, a good knowing of the assembly language is needed.

- One important point in designing software for embedded system is to design with debugging in mind. More often than not, your code won't work the first time. Unlike hardware development, the time taken in testing and debugging during software development can be surprisingly long if you are not careful. Well organized code is a must if you want to minimize debugging time. Well commented code mentioned above is another cardinal virtue in programming.

Basically, one must adhere to good software engineering methodology. We shall look at a number of issues pertaining to software development for embedded systems. Ideally a development environment system for embedded system work should have the following three components:

- **Host computer** - This is typically a PC which runs the editor, linker and compiler. PC has become the de facto standard as development platform for embedded systems because of its availability and the amount of commercial and public domain software tools obtainable. Traditional embedded system vendors have designed their development tools with the PC in mind. This also encourages a large number of third party software vendors to use the PC platform for their software tools.
- **Debugging engine** - This refers to the component that allows you to *look* into your target system in terms of code execution. It may be in the form of an in-circuit emulator or in smaller projects a monitor program resident in the target system itself. This debugging engine allows you to open a window in the host computer and monitor the execution of your code or status of your processor in the target system. For any serious work, it is no longer acceptable to compile your code, program the EPROM, plug it in and hope that it will work!
- **Source-level debugger (SLD)** - This is a piece of software running in the host PC which allows you to debug your code at source level, in conjunction with the debugging engine. Not only does it communicate with the debugging engine or target system, it also provides intelligent assistance in the debugging stage. For example it displays the source code (actual C statement instead of assembly code) at which the target is at, resolves symbolic references, examines in the high level format, allows breakpoint to be set at source level, single step through the code again at source code level, etc. Generally a good SLD will provide all these features in very neat multiple window environment, thus making debugging a much easier task than if it is done at assembly code or machine code level.

4.1 Cross Development

As mentioned above, mainly because of the ubiquitous position, the PC is almost universally used as the platform for embedded system development.

In which one would be doing cross development running a host of cross software - cross assemblers and linkers, cross interpreters, cross compilers. Unless of course one is developing an embedded system with the same CPU as the PC used (e.g. 80186, 80188, 80386EX or the PC itself used and embedded processor.).

Cross development is necessary for a number of other reasons:

- Many microcontrollers used in embedded systems are just too small to be used as processors in development systems. Native or resident assemblers and compilers may not be available for such systems.
- Existing computer facilities are readily available and with the appropriate cross-development software tools, are suitable for carrying out the task of software development. This is considered an important advantage because no extra hardware is needed and software tools such as editors are already available.
- Nowadays, one can find cross-development software tool for almost any processor in the market. Some manufacturers are supporting their products with a dial-up facility or through Internet which allows users to download cross-assemblers and cross-compilers to the PC.

Thus, cross assemblers are programs that run on a computer with a different processor from that of the target system, and *assemble* programs written for the target system into *relocatable object code.* The linkers then relocate, usually with other object modules such as library modules, to the desired execution addresses for the target machine. Common features of cross assemblers are: (1) provision for using macros in program, thus *macro-assembler,* (2) conditional assembly, (3) assembly time calculations and (4) listing control.

Similarly, cross compilers are programs that run on a computer with a different processor from that of the target system, and *compile* high level language programs written for the target system typically into assembly language programs. The use of a cross compiler can reduce program development time significantly for large projects. It also makes programs more portable, since they are written in a high level language such as C. A typical cross compiler consists of: (1) macro preprocessor, (2) parser, (3) optimizer and (4) code generator.

4.2 Simulation

Simulation is a way of using software to model the target system including the target processor itself. The program can *see* his system running in the stable environment of his host computer which run the simulation program. This is used when the target system is not available, when the target prototype is still unreliable, or when the programmer has to access the low level status of the system not normally accessible in embedded systems.

While it sounds like a great idea, unfortunately good simulators for embedded systems are not readily available. This is due to the fact that the simulator has to deal with real-time events and sometimes rather complex I/O. How can you get a general purpose simulator to understand your obtuse or ingenious interface to the solar tracking system? How do you simulate real-time, asynchronous events? To duplicate the data stream coming from the outside world is not easy either.

Nevertheless, there are simulators available for many processors. One successful category of simulators seems to be the microcontrollers such as the 8051. When most of the I/O are integrated on a single chip, they are well defined and thus can be simulated more readily.

5 Other Techniques for Embedded Systems

Armed with the above, one can embark on the actual coding, compiling, downloading and debugging of the embedded system. Elegant structuring of the program is very important in embedded system design, as in all other software design. A monitor program tugged in the EPROM of an embedded system is not too much to ask for nowadays. This will help in the debugging process tremendously. In structuring your program, however, there are two other techniques that have been used by many designers and found to be very useful. These are (1) state machine technique and (2) real-time kernel.

5.1 State Machines and State Tables in Embedded Systems

For small systems, *sequential organization* of the program is often used. The entire function of an embedded system is represented by a flowchart and implemented accordingly using a single main loop. When external inputs or events arrive, the program branches off to some modules to carry out the required actions.

There are however a number of shortcomings using the above method:

- Testing of a monolithic program is often difficult.
- When the loop becomes large as more functions are added, life becomes complicated. When single large loop is used, there is a tendency to produce *spaghetti* code.
- Subsequent modifications of system function, like adding another control switch, are tedious because the entire flowchart has to be revised and often re-implemented entirely.

For many embedded systems, the complexities often justify a more systematic approach of designing the software. Representing the function of a system by a **state machine** is such a approach. The power of state machine representation comes from the fact that it can subsequently be represented by a **state table** which is well suited for microcontroller and microprocessor implementation, even at assembly language level.

Using the state table method of implementing the functions of a system, it is natural that the job be broken down into small, more manageable and often independent modules, called the *action routines*. Such routines are more easily tested and often reusable.

However, the single most important advantage of state table implementation really lies in the ease of function modification. In most cases, only the state table is modified together with the necessary new routines, while most of the old code would be intact.

5.2 An Example of State Machine Representation

A simple example of a system with keyswitches and display is given here to illustrate the method of state machine representation.

- Suppose we have a keypad with ten numeric keys 0 to 9 and two function keys $\boxed{\text{ENTER}}$ and $\boxed{\text{DELETE}}$ and a 4-digit numeric LED display.
- On power up, the display shall show 0.
- Numeric values can be entered on the keypad and as each digit is entered, it is scrolled into the display from the rightmost digit. During this mode, the display blinks to indicate *digit entering mode.*
- The *digit entering mode* is terminated with either the $\boxed{\text{ENTER}}$ key or the $\boxed{\text{DELETE}}$ key.
- If $\boxed{\text{ENTER}}$ is pressed, the display stops blinking.
- If $\boxed{\text{DELETE}}$ is pressed, the display stops blinking and shows 0.

There are 3 possible states in this example:

State	Name	Description
S0	Initial	Power-on state or after **DELETE**, display shows **0** in steady mode.
S1	Data Entry	Digit entry mode, display shows digits in blinking mode.
S2	Display	Final display mode, display shows final value in steady mode.

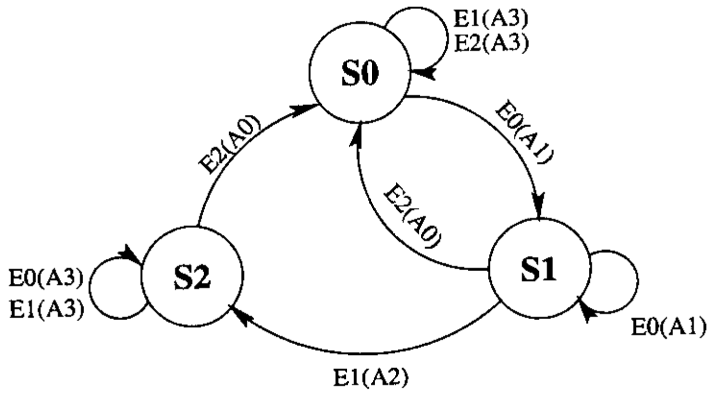
There are 3 types of events:

Event	Name	Description
E0	Number	Entry of any numeric key.
E1	Enter	**ENTER** key is pressed.
E2	Delete	**DELETE** key is pressed.

There are three action routines needed:

Action	Name	Description
A0	Reset	Display 0.
A1	Build digits	Build up display buffer from right while numbers are entered and blink display.
A2	Steady display	Show steady display.
A3	Null	No action.

The specification mentioned earlier is represented by a state diagram.



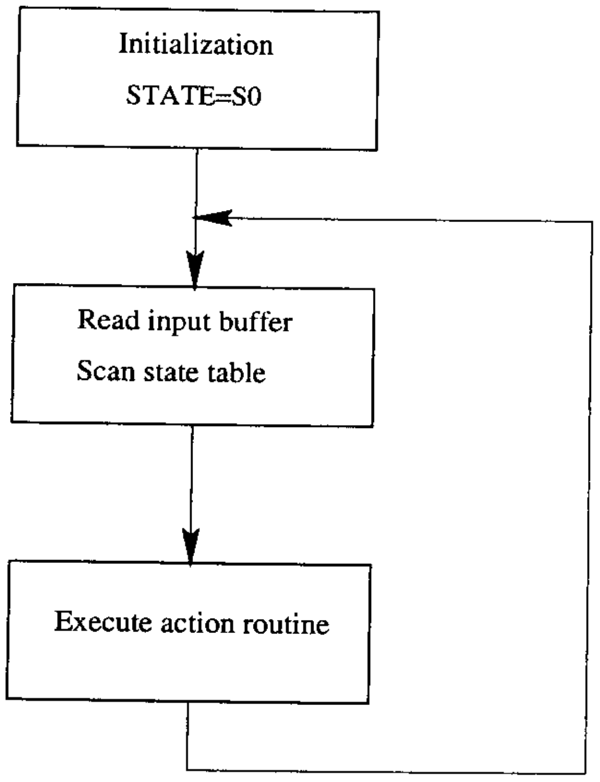
The above state diagram can be easily transformed into a state table representation.

Present State	Event	Action	Next state
S0	E0	A1	S1
	E1	A3	S0
	E2	A3	S0
S1	E0	A1	S1
	E1	A2	S2
	E2	A0	S0
S2	E0	A3	S2
	E1	A3	S2
	E2	A0	S0

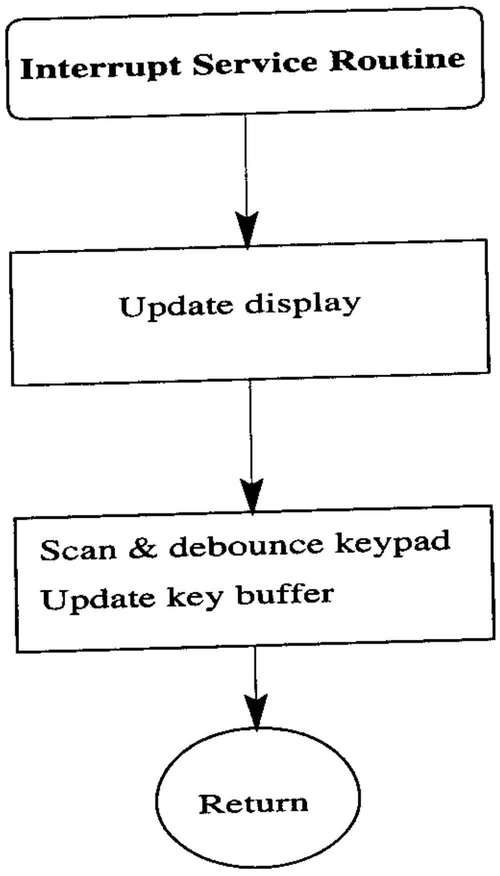
The complexity of the system has thus been broken down into:

- A number of action routines.
- A service routine to scan the keypad and update display.
- A state stable.
- A very small main program.

The main program structure is represented below:



The keypad and display service routine may be implemented as an **interrupt service routine** based on 10-ms clock ticks from a programmable timer module, for example:



5.3 Task Scheduler in Embedded System

An application in real-time embedded system can always be broken down into a number of distinctly different tasks. For example,

- Keyboard scanning
- Display control
- Input data collection and processing
- Responding to and processing external events
- Communicating with host or others

Each of the above tasks can be represented by a state machine. However, implementing a single sequential loop for the entire application can prove to be a formidable task. This is because of the various time constraints in the tasks - keyboard has to be scanned, display controlled, input channel monitored, etc.

One method of solving the above problem is to use a simple **task scheduler**. The various tasks above are handled by the scheduler in an orderly manner. This produces the effect of simple multitasking with a single processor. A bonus of using a scheduler is the ease of implementing the *sleep* mode in microcontrollers which will reduce the power consumption dramatically (from mA to μA). This is important in battery operated embedded systems.

There are several ways of implementing the scheduler - preemptive or cooperative, round robin or with priority. In a cooperative or non-preemptive system, tasks cooperate with one another and relinquish control of the CPU themselves. In a preemptive system, a task may be preempted or suspended by different task, either because the latter has a higher priority of the time-slice of the former one is used up. Round robin scheduler switches in one task after another in a round robin manner whereas a system with priority will switch in the highest priority task.

For many small microcontroller based embedded systems, a cooperative (or non-preemptive), round robin scheduler is adequate. This is the simplest to implement and it does not take up much memory. Ravindra Karnad has implemented such a scheduler for 8051 and other microcontrollers. In his implementation, all tasks must behave cooperatively. A task waiting for an input event thus cannot have infinite waiting loop such as the following:

```
While (TRUE)
{
 Check input
 ...
}
```

This will hog processor time and reprieve others of running. Instead, it may be written as:

```
If (input TRUE)
{
 ...
}
Else (timer[i]=100ms)
```

In this case, *task i* will check the input condition every 100 ms, set in the associated *timer[i]*. When the condition of input is false, other tasks will have a chance to run.

The job of the scheduler is thus rather simple. When there is clock interrupt, all task timers are decremented. The task whose timer reaches 0 will be run. To simplify things, the state *status* of the task is used by the scheduler to decide where to pass control to.

The greatest *virtue* of the simple task scheduler ready lies in the *smallness* of the code, which is of course very important in the case of microntrollers. The code size ranges from 200 to 400 bytes.

5.4 Real-time Kernel in Embedded Systems

Real-time operating system (RTOS) is the central theme of this College and it would be nice if we can incorporate such an OS in our embedded systems. Unfortunately, more often than not, the memory and other resources of most embedded systems we build do not permit this. There is however an alternative - that of using a subset of the RTOS to solve the problem of embedded systems. If the I/O and file handling is removed from the fully fledged RTOS, we are left with a kernel which deals with tasks handling. This turns out to be a powerful tool in dealing with real life embedded system applications, such as the state machine technique.

In embedded systems, interrupts are used to respond to external events and in doing so avoid the waste of CPU time by constant polling for such events. However, interrupt handling can be rather complex if there are many processes to be handled simultaneously. In many situations, embedded systems run more or less independent programs which share some common resources. A very large intertwined program will result if we use simple interrupt handling technique. Real-time kernel (RTK) will help the programmer to deal with such circumstances by thinking in terms of concurrent tasks instead of individual routines that execute when certain events occur.

Real-time kernels come in a great variety of types. Many of the small RTKs are implemented in assembly language; others are implemented in HLLs such as C. A recent survey shows that there are more than 40 RTK manufacturers producing kernels for 8-, 16- and 32-bit processors including proprietary and open market ones. The price tag of these commercial RTKs ranges from USD$100 to USD$10,000.

There are also a small number of real-time kernels appearing in journals, magazines and books, which are normally available in source code. Later in this series of lecture, we shall look at one designed by Jean J. Labrosse called μC/OS, which is implemented in C with full source code available to the user.

6 The 68HC11 Microcontroller

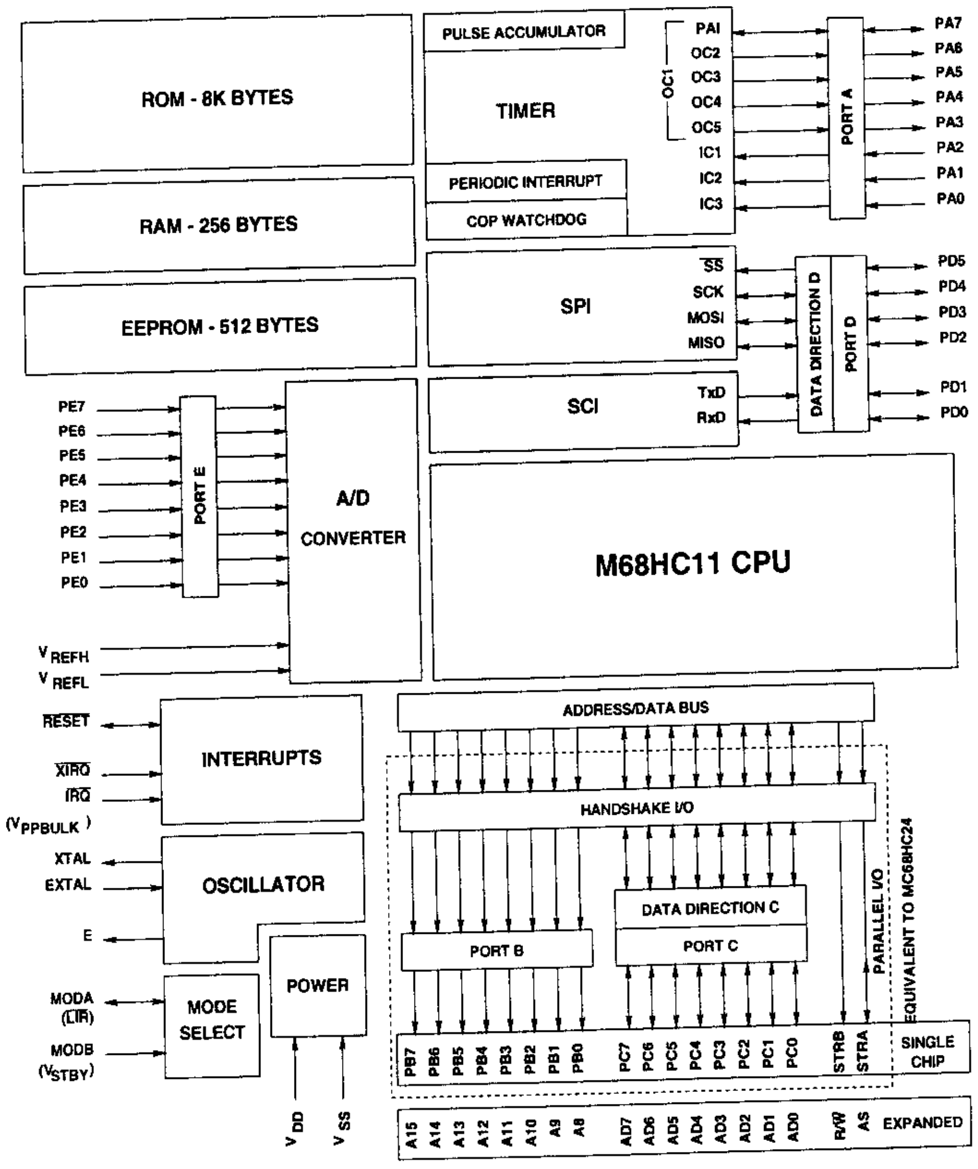
We have mentioned earlier that there are now many microcontrollers available in the market. We shall look at one of them, the 68HC11, in this section. It is a family of microcontrollers with members providing different I/O and memory facilities. They can be used in *single-chip* or *expanded mode.*

The main features are:

- **Parallel I/O** - 40 I/O lines arranged as five 8-bit ports, two general purpose and three fixed direction.
- **ADC** - 8-channel, multiplexed-input, successive approximation with sample and hold. Conversion time 16 μs for 2 MHz system.
- **Serial communications** - A full-duplex two-wire asynchronous serial communications interface (SCI) with baud rate ranges from 75 bps to 131 Kbps. A full-duplex three-wire synchronous serial peripheral interface (SPI) with a maximum master bit frequency of 1 MHz.
- **Programmable timer** - 16-bit with four stage prescaler, three capture functions and five output compare functions.
- **Memories** - ROM (4K, 8K or 12K), EPROM (4K or 12K), EEPROM (512, 2K or 8K), RAM (256, 512 or 1K).
- **Interrupts** - Nonmaskable interrupt (XIRQ) and maskable interrupt (IRQ). IRQ is either level-sensitive or falling-edge-sensitive.
- **Pulse accumulator** - A 8-bit counter used for event counting or gated-time accumulation.
- **COP watchdog** - A computer operating properly watchdog is used to detect error in the system. When it is used, the program is responsible for keeping an internal free-running watchdog timer from timing out. If the watchdog times out, the MCU will be reset. This is an important feature in embedded systems as most of them are running unattended. In the case where watchdog is not built in, an external watchdog circuit using a couple of monostable multivibrators is often used.
- **Low power modes** - In single chip mode, 15 mA for normal operation, 6 ma in WAIT mode and 100 μA in STOP mode.

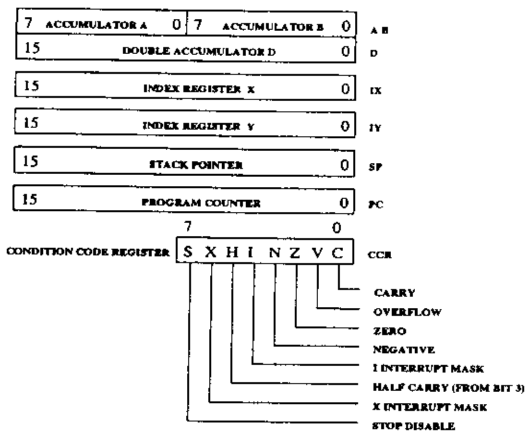
6.1 Architecture of the 68HC11

A simplified diagram of the architecture of the 68HC11 is shown in the figure below.The parallel I/O subsystem consisting of ports PB, PC and STRA and STRB is lost if the MCU is used in the expanded mode. A MC68HC24 port replacement unit can be used to regain the functions of the ports and the control lines. The functions are restored such that there is no distinction between the two. Thus an expanded system with an MC68HC24 and an external EPROM can be used to develop software intended for single-chip application.



6.2 Programming Model

The 68HC11 has 91 new opcodes in additions to those of 6800 and 6801. Now it has a total of 109 instructions. Both multiplication and division are possible now. Bit manipulation instructions are also available.



c

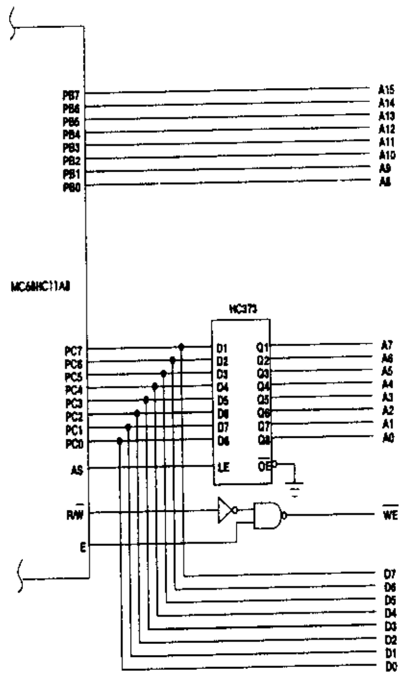
6.3 Modes of Operation

There are 4 hardware controllable modes of operations that are available:

Mod A	Mod B	Mode of Operation
0	1	Single Chip
1	1	Expanded
0	0	Bootstrap
1	0	Special Test

- **Single-chip mode**. The chip functions as a monolithic microcontroller without external address or data bus.

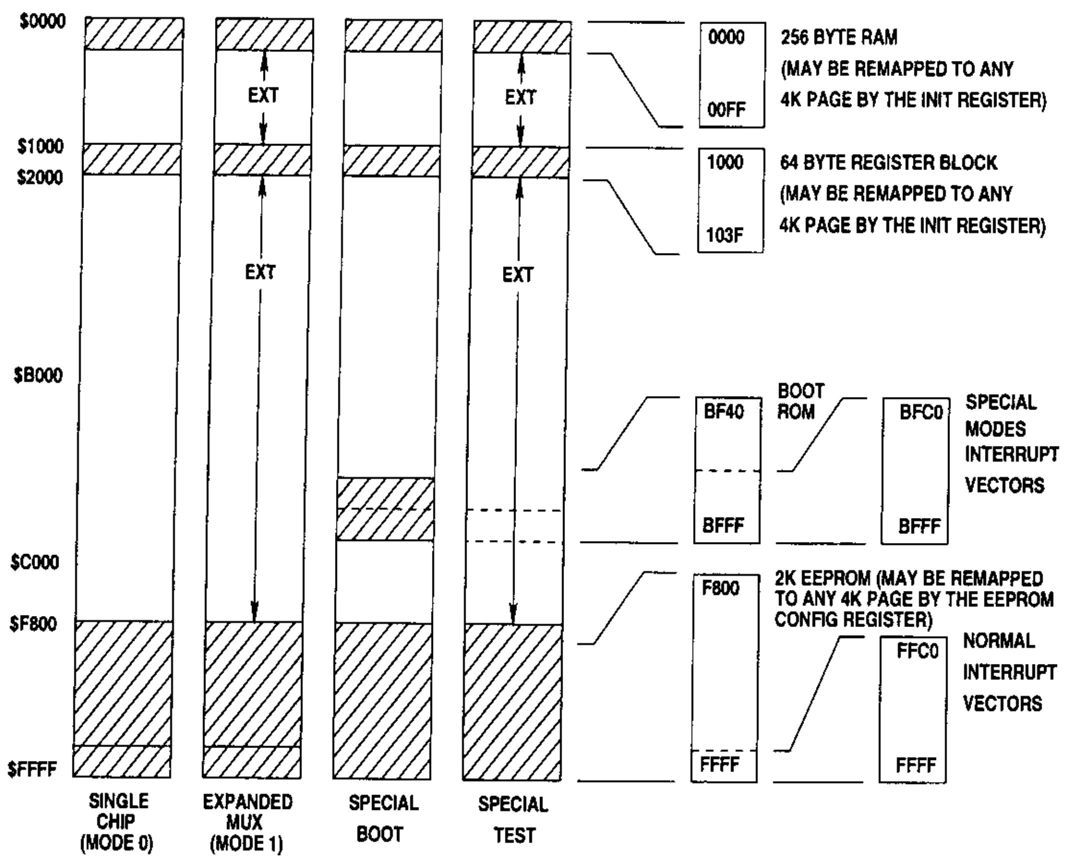
- **Expanded-multiplexed mode**. The chip can access a 64KB address space. The total address space includes the on-chip memory addresses. The expansion is made up of port B and port C, and control signals AS and R/W.



- **Bootstrap mode**. A special operating mode that uses a boot loader program in the bootstrap ROM to load program into RAM via SCI. This is how you get your program loaded into the MCU memory.
- **Special test mode**. This is a factory testing mode similar to the expanded- multiplexed mode except that the reset and interrupt vectors are fetched from external memory locations.

6.4 Memory Maps

The memory maps of the four different modes are shown below. In expanded mode, the areas not used internally are for external memory and I/O. If an external memory or I/O device is located to overlap an enabled internal resource, the internal resource will take priority.


$0000
$1000
$2000
$B000
$C000
$F800
$FFFF
EXT
EXT
EXT
EXT
SINGLE CHIP (MODE 0)
EXPANDED MUX (MODE 1)
SPECIAL BOOT
SPECIAL TEST
0000
00FF
256 BYTE RAM (MAY BE REMAPPED TO ANY 4K PAGE BY THE INIT REGISTER)
1000
103F
64 BYTE REGISTER BLOCK (MAY BE REMAPPED TO ANY 4K PAGE BY THE INIT REGISTER)
BF40
BFFF
BOOT ROM
BFC0
BFFF
SPECIAL MODES INTERRUPT VECTORS
F800
FFFF
2K EEPROM (MAY BE REMAPPED TO ANY 4K PAGE BY THE EEPROM CONFIG REGISTER)
FFC0
FFFF
NORMAL INTERRUPT VECTORS

NOTE:

1. Either or both the internal RAM and registers can be remapped to any 4k boundary by software.

7 A Design Example Using the 68HC11

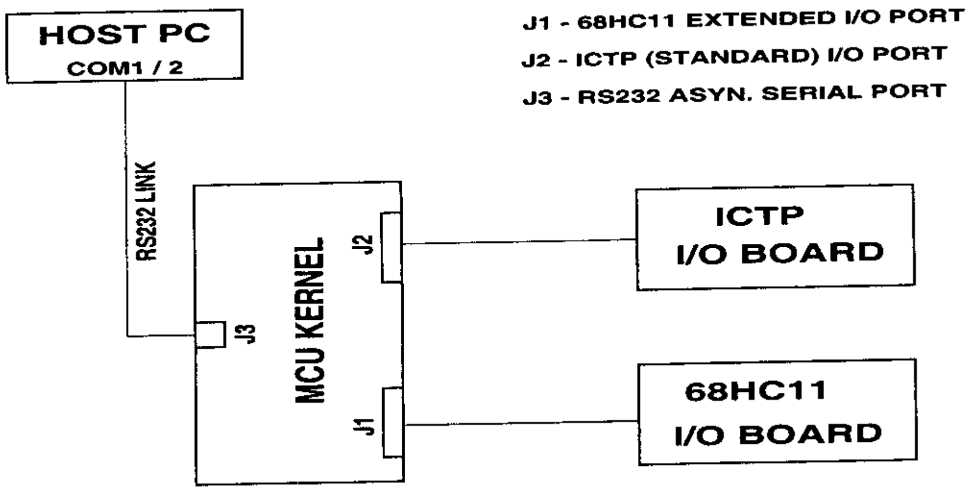
7.1 System Overview

The 68HC11 embedded system is one of several designed in this College to demonstrate the concepts of real-time embedded systems and the technique of cross development of such systems. In this particular one, simplicity of hardware and development tool is emphasised. In fact, besides the micro-controller, only one other chip, the RS232 interface driver, is essential in the system, making it a really *minimal* system. It is conceivable that every participant can go home with one such system, or at least the PCB for such a system.

However, it is noted that though very small, it is nevertheless a fully functional simple development system working in conjunction with a host station such as a PC and the appropriate software. Only a standard RS232 serial link between the host station and the target system is needed. Assembled or compiled object code can be downloaded to the target system and stored permanently in the EEPROM without requiring an external EEPROM programmer or other hardware. Uploading of target system code can also be done if necessary.

As a simple system, in circuit emulation and debugging facilities such as those provided by the Motorola Evaluation Module M68HC11EVM are not available. This however is not a serious hindrance in learning the cross development of a real-time embedded system.

A block diagram of the 68HC11 system is shown below followed by description of the various sub-units in other sections.


HOST PC
COM1 / 2
J1 - 68HC11 EXTENDED I/O PORT
J2 - ICTP (STANDARD) I/O PORT
J3 - RS232 ASYN. SERIAL PORT
RS232 LINK
MCU KERNEL
J1
J2
J3
ICTP I/O BOARD
68HC11 I/O BOARD

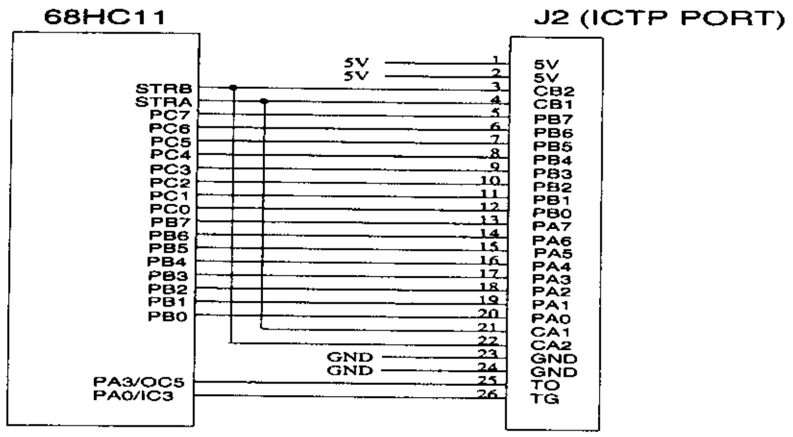
7.1.1 Host PC

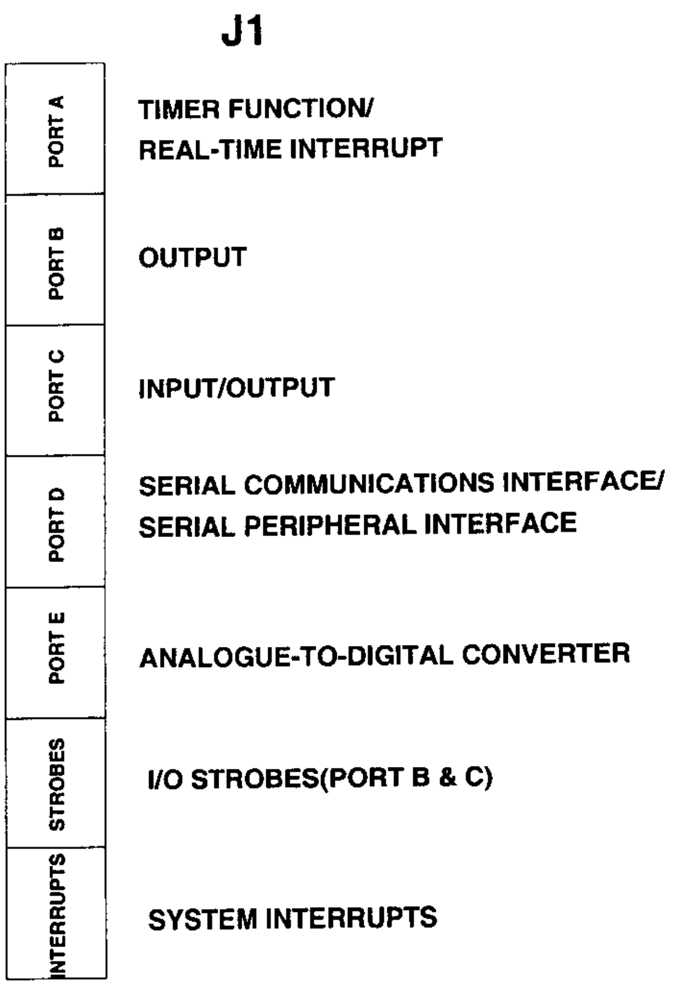
The external host is typically a PC running Linux or other operating system with suitable cross development software for the 68HC11. A COM port on the PC is used to communicate with the 68HC11 target system. This link serves as a code downloading or uploading channel during the development stage. During the running or execution stage, the serial link may be used for data communications between the PC and the target system if necessary. Or it may be used by the target system to communicate with an external instrument or equipment.

7.2 HC11 Microcontroller Kernel

The **HC11 Kernel** is a small board capable of communicating with a host and interfacing to different target I/O subsystems. The entire board consists of merely a 68HC811E2 microcontroller, an RS232 driver, a 5-V regulator, an 8-MHz crystal, a low voltage inhibitor (for reset), pull-up resistors, capacitors and connectors. It highlights the capabilities of a typical microcontroller. The main features of this board are as follows:

- **ICTP PORT** – A 26-pin *standard ICTP* I/O port (J2) to interface with ICTP I/O board or other similar boards. However it does not fully conform to the specification of the ICTP Port which is essentially based on the ports of a Motorola peripheral interface adapter (PIA). PA0–7 of J2 is connected to Port B of the 68HC11. This port is a *output only* port. PB0–7 of J2 is connected to Port C of the 68HC11. This is an I/O port. CA1 and CB1 of J2 are connected to input strobe pin (STRA) whereas CA2 and CB2 are connected to the output strobe (STRB) of the microcontroller. There are functional differences between the PIA strobe lines and those of the 68HC11.





- **HC11 PORT** - A 40-pin extended I/O port (J1) to bring out most of the peripheral lines for use with a 68HC11 I/O board. This connector consists of the following:
  - Timer function and real-time interrupt port (Port A).
  - General purpose output port (Port B).
  - General purpose I/O port (Port C).
  - Serial communications interface (SCI) and serial peripheral interface (SPI) port (Port D). This port may be used as general purpose I/O.
  - ADC or general purpose input port (Port E).
  - Input and output strobes (STRA, STRB).
  - Interrupts (IRQ, XIRQ)

- **RS232 Serial Port** - An RS232 serial communications port (J3). This port uses the TxD and RxD of Port D for asynchronous serial communications. A Maxim RS232 driver/receiver chip operating at single 5V supply is used.

- **Power Consumption** - The board is powered either by a regulated 5V DC supply or an unregulated DC supply ranging from 7 to 12 V which is readily available in the form of AC adaptor. For the latter a 5V regulator is used to produce the 5 V required by the MCU and other components. The regulated 5V is also brought to the 68HC11 I/O board through connector J1. Current consumption of the microcontroller (MC68HC811E2) is 15 mA which is relatively small. Other components in the board have low power consumption too. The current consumption of the I/O varies a bit depending on the states of the LED lamps. An overall 200 mA should suffice for this system.

- **Clock frequency** - An 8-MHz crystal is used to produce a MCU clock frequency of 2 MHz.

- **Reset circuit** - A low voltage inhibit device (MC34064) is used in the RESET set to drive the RESET low when the supply is below legal limits. This will prevent the unintentional corruption of the on-chip RAM and EEPROM. Of course the manual RESET button is still there.

- **Boostrap/Normal mode selection** - A *bootstrap/Normal* mode selection switch is connected to MODB pin of the MCU. In the bootstrap mode, the resident ROM bootstrap loader which will download a 256-byte program into the RAM. This feature together with the on-chip EEPROM programming capability make the board a small self-contained development station.

7.2.1 ICTP I/O Board (or Colombo I/O Board)

ICTP I/O boards may be connected to the HC11 Kernel board via J2. As mentioned earlier the J2 pins are not exactly the same as those specified. However, the original ICTP (or Colombo) I/O board should pose no problem in its present form. This is because PA0-7 are used as outputs for connecting to four 7-segment LEDs and not as a general purpose I/O port. Strobe lines do behave differently and program/driver should be modified accordingly.

Other I/O boards requiring J2 connection may be used as long as PA0-7 are not required to function as inputs.

7.3 HC11 I/O Board

There are innumerable ways of designing the I/O board. It is hoped that several I/O boards may be constructed to demonstrate the versatility of the microcontroller. It is envisaged that participants may subsequently wish to design and construct their own I/O subsystems which are more specific to their problems. For example, an experiment that requires counting of events, measurement of pulse width or precise pulse generations would make full use of the timer and real-time interrupt offered by the entire Port A of the 68HC11. Similarly, a situation where 4 ADCs are required may call for the design of a different I/O subsystem with the appropriate signal conditioning circuits.

For a start a rather basic board is built. It is intended to demonstrate the basics instead of showing the full capabilities of the microcontroller. Some functions and components presently available in the ICTP I/O board are not duplicated. Others that are simple to incorporate and considered useful in learning the cross development of an embedded system are included. The first HC11 I/O board consists of the following:

- **LED indicators** – Small LED lamps are connected to Port B. These can act as general purpose indicators but they are considered important in the development of embedded system as a debugging aid for reporting status.
- **DIP switches** – A 8-way DIP switch module is connected to Port C to act as simple digital input devices. These switches, as the LED lamps, are important aid in the development of embedded system. They allows the user to interact with his system easily.
- **Pulse input** – A push-button switch is connected to the input strobe pin (STRA).
- **Strobe output** – An LED lamp is connected to the output strobe pin (STRB) through a buffer. This output also select either the LCD mode or LED/SW mode. A HIGH selects the LCD.
- **Analogue input** – A miniature multiturn potentiometer providing 0-5V is connected to one of the ADC inputs.
- **External analogue input** - Provisions are made of connecting external sources to ADC inputs.
- **LCD panel** - A 16-character by 1-line LCD display panel is connected to Ports B and C. This is a more sophisticated output device capable

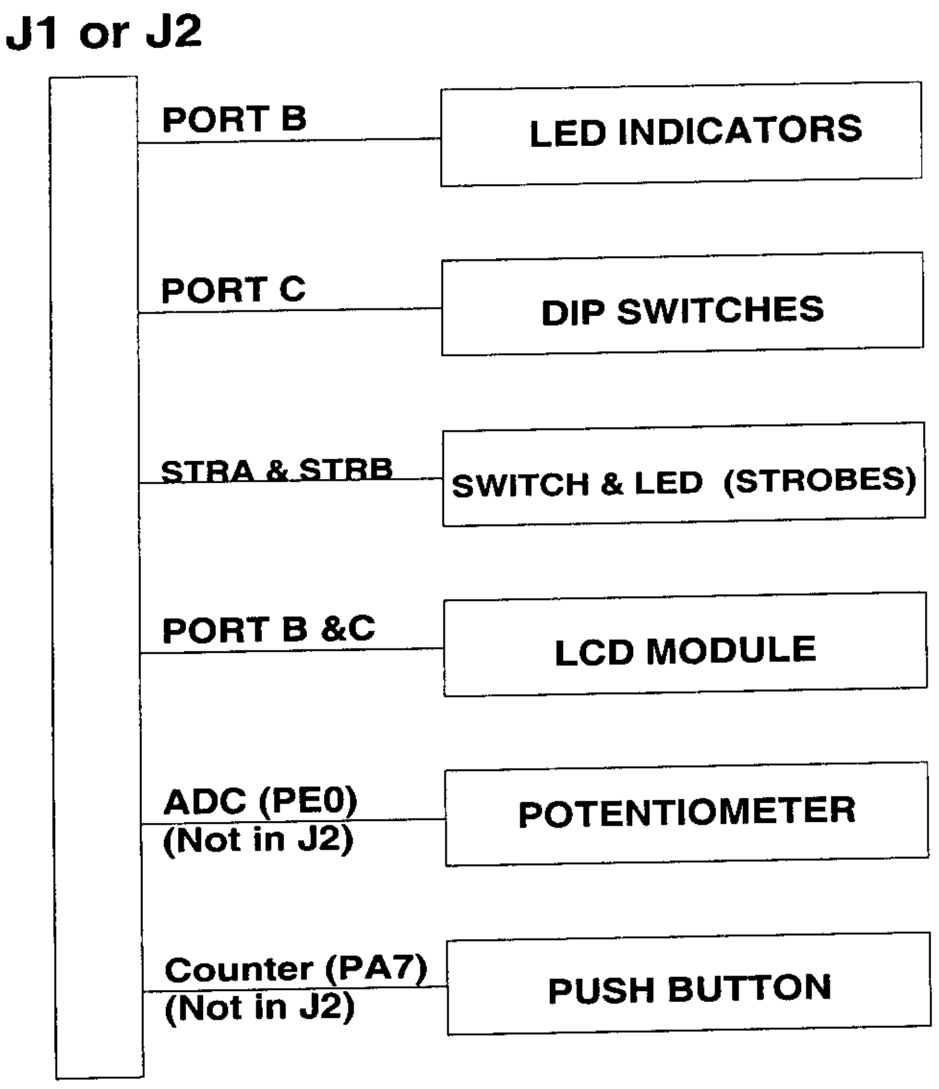
of displaying simple text messages. It is a rather useful*user interface* in a standalone system.

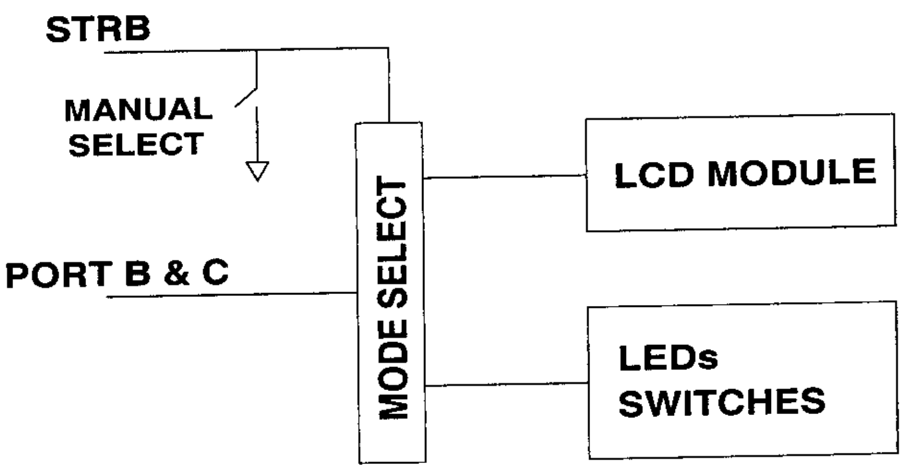
7.3.1 LCD & LED/SW Mode Selection

There are two ports (J1 & J2 connectors) brought out of the HC11 I/O Board which match those on the HC11 Kernel Board. J1 (HC11 PORT) is a 40-way connector which carries most of the HC11 I/O lines. J2 (ICTP PORT) is a 26-way standard ICTP I/O port. Most of the I/O devices mentioned above (with the exception of analogue input and pulse counter) can operate with either the ICTP PORT or or HC11 PORT. This is a constraint because in doing so we have only Port B and Port C of the HC11 only. Consequently, the LCD and LEDs/Switches cannot function simultaneously. A selection of either the LCD or the LED/SW has to be made. This is done either by the STRB signal or manually using a jumper through the use buffers. However, LEDs connected to PB3-7 are not required by the LCD and hence can be used during the LCD mode. Please refer to the appended circuit diagram for details.

7.3.2 LED Panel

This itself is an embedded sub-system consisting of a twisted nematic mode reflective liquid crystal dot matrix display and an embedded controller and driver in bare chip form directly attached on the PCB. The display appears as a 16-character by 1-line alphanumeric display while internally, as far as the controller is concerned, it is connected as 8 characters by 2 lines. The controller chip is a Samsung (or equivalent) dot matrix LCD controller KS0066. Please refer to the manufacturer's data sheet for programming this chip. If you don't see any pattern at all the panel, please adjust the contrast control (potentiometer).

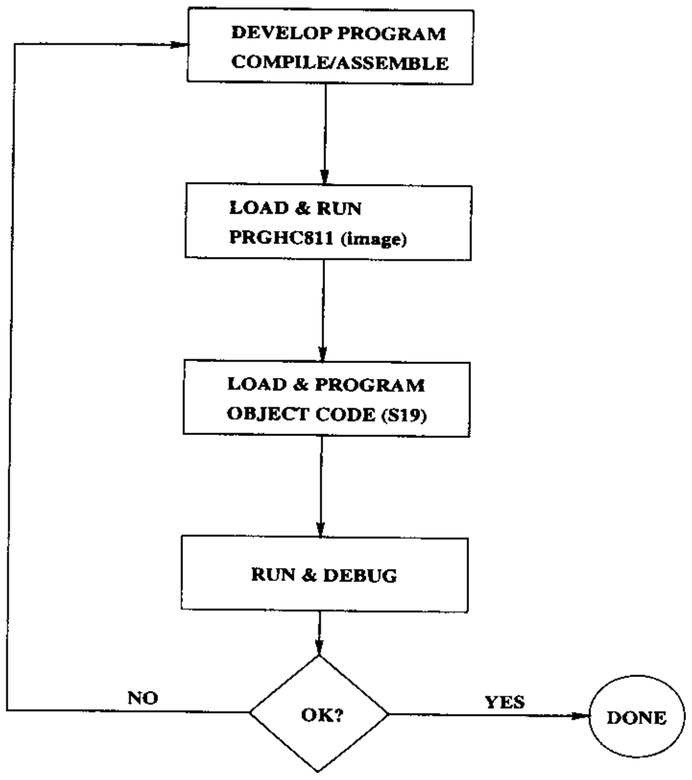

J1 or J2
PORT B
LED INDICATORS
PORT C
DIP SWITCHES
STRA & STRB
SWITCH & LED (STROBES)
PORT B &C
LCD MODULE
ADC (PE0)
(Not in J2)
POTENTIOMETER
Counter (PA7)
(Not in J2)
PUSH BUTTON


STRB
MANUAL
SELECT
PORT B & C
MODE SELECT
LCD MODULE
LEDs
SWITCHES

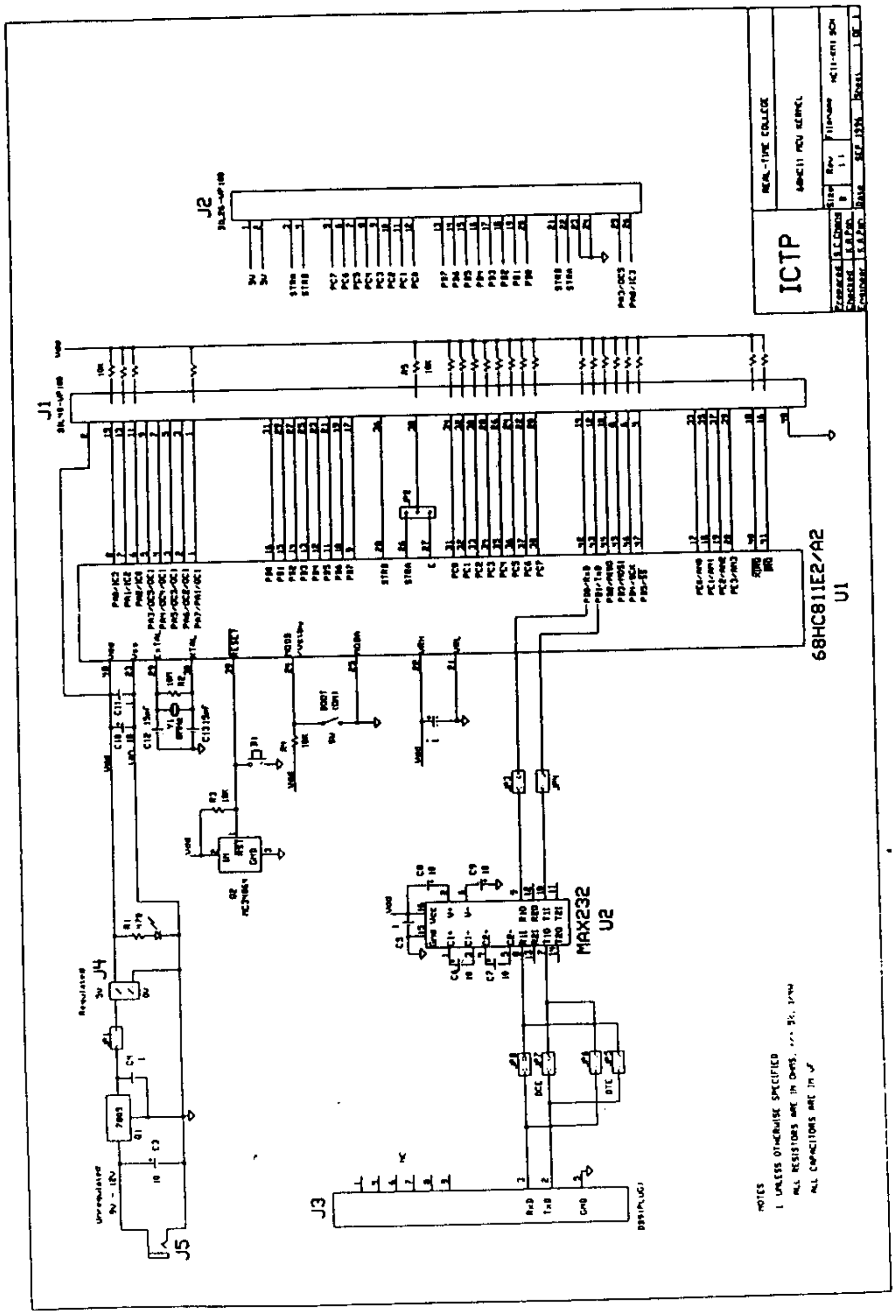
7.4 Program Development

Program development for the 68HC11 Embedded System consists of the following steps:

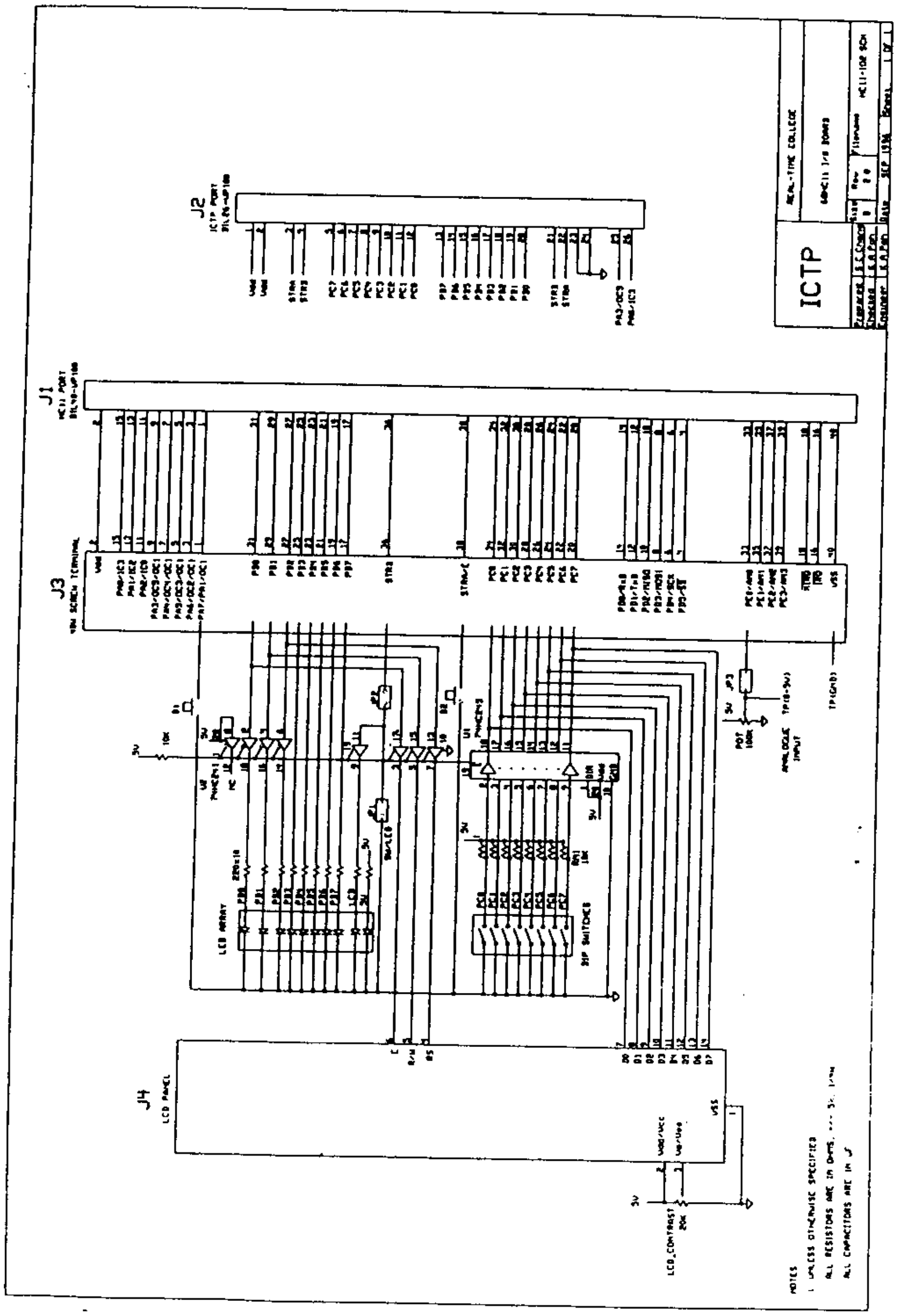
- Develop source program in host environment either in high level language or in HC11 assembly language.
- Compile or assemble source into HC11 object code in S19 format.
- Download and run HC811 programmer (PRGHC811) memory image code to the 68HC11 Embedded System RAM using the bootstrap mode.
- Download the application in S19 format into the system and program the EEPROM using the PRGHC811 which is now running.
- Reset and run the loaded target program.
- Repeat from the first step if target program does not behave as required.



7.5 Circuit Diagram of the 68HC11 Kernel Board



7.6 Circuit Diagram of the 68HC11 I/O Board



8 The Z80180 Microprocessor

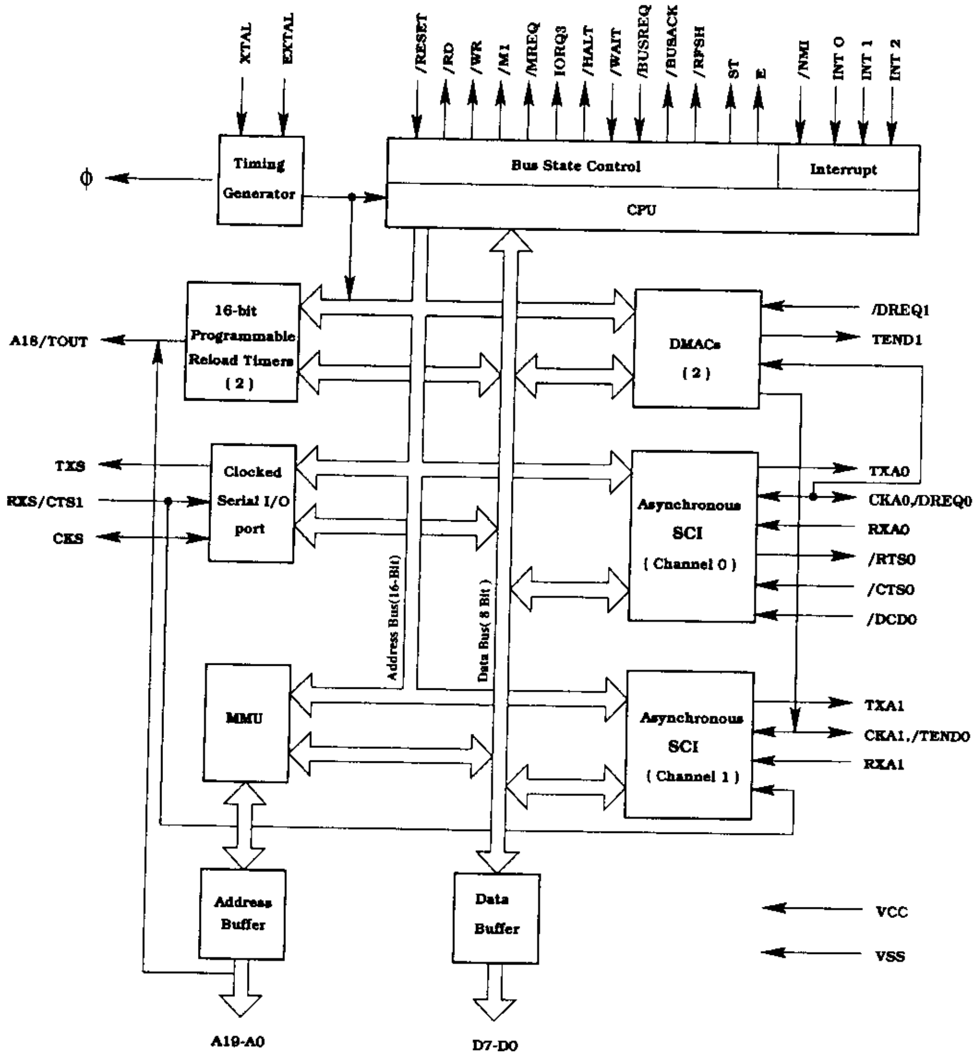
The Z80180 is listed as a microprocessor in the catalogue but it is rather close to a microcontroller and is a good candidate
a microcoded execution unit in CMOS, this chip offers rather high performance and maintains compatibility with a large amount of existing Z80 programs.

The main features are:

- **Improved performance** - Higher performance than the Z80 is obtained by reduced execution times, an enhanced instruction set, and high operating frequencies. Up to 33 MHz at 5 V or 20 MHz at 3.3 V is available.
- **Large memory space** - An on-chip memory management unit (MMU) supports extended address space of up to 1 MB of memory.
- **DMA channels** - Two direct memory access channels provide high speed transfer of data between memory and I/O devices using either request, burst or cycle-steal mode. Transfer can be effected between memories, between I/Os or between memory and I/O.
- **Serial communications channels**- Two full-duplex asynchronous serial communication channels (UART) each with a programmable baud rate generator and modem control. Some versions offer break detection and generation. A clocked serial I/O (CSIO) provides a half-duplex serial transmitter and receiver, which can be used for high speed data transfer.
- **Programmable timers** - Two 16-bit programmable timers. One can be used as a waveform generator.
- **Z80 MPU** - Code compatible with Z80 MPU.
- **Low power consumption** - Power consumption at 10 MHz is 25 mA in normal operation and 6 mA in STOP mode. Versions that provide STANDBY mode consumes less than 10 μA in this mode.

8.1 Architecture of the Z80180

The architecture of the Z80180 is shown below. Basically it has a CPU core with number of system and I/O resources. The core has a clock generator, bus state controller, interrupt controller, memory management unit and a central processing unit. The integrated peripheral resources consist of direct memory access controls, asynchronous serial communication interface and clocked serial interface and programmable timers.



8.2 Programming Model

The Z80180 is object code compatible with the Z80 MPU. Thus one can refer the Z80 technical data for the full instruction set and programming model. It has three groups of registers:

- **Register Set GR** – This consists of a 8-bit Accumulator (A), a 8-bit Flag Register (F) and three general purpose registers (BC, DE and HL) which may be treated as 16-bit or 8-bit registers depending on the instruction.
- **Register Set GR'** – An alternate set of registers to the GR. They are not directly accessible but the contents may be exchanged with the GR set at high speed.
- **Special Registers** – These consist of an 8-bit Interrupt Vector Register (I), an 8-bit R Counter (R), two 16-bit Index Registers (IX and IY), a 16-bit Stack Pointer (SP), and a 16-bit Program Counter (PC).

Besides the Z80 instructions, a number of new ones have been added. They include instructions to enter sleep mode, 8-bit multiplication, I/O manipulation, etc.

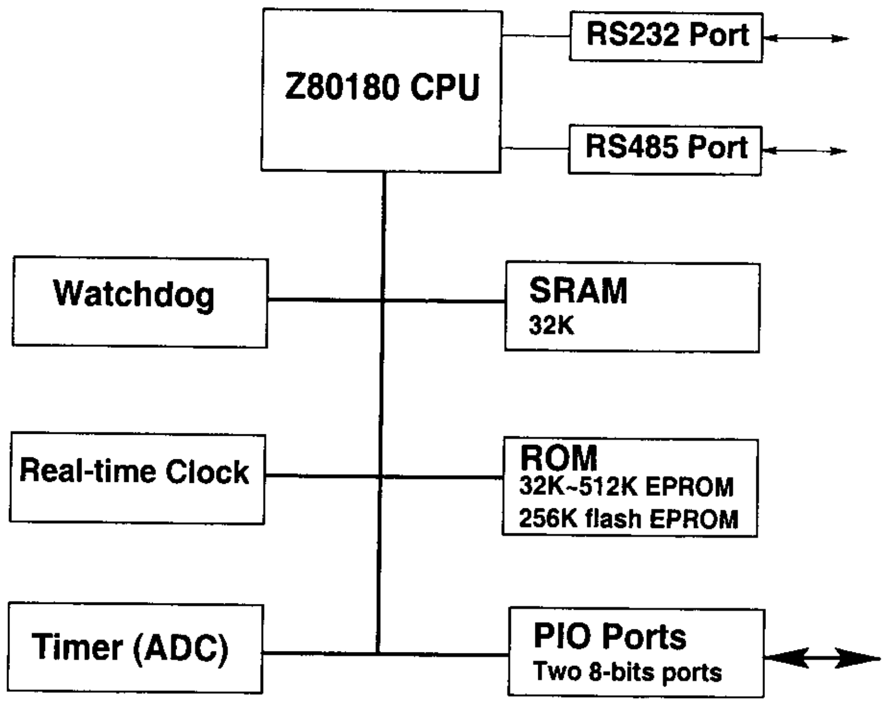
9 A Z80180 Embedded System

9.1 System Overview

We have earlier introduced a very small embedded system using the 68HC11 single chip microcontroller. The elegance of that design is in its simplicity - a mere two-chip board. Come with the simplicity is resource limitation, essentially in memory size. The version we used has only 2KB of EEPROM and 256 bytes of RAM.

In this section, we shall introduce a larger embedded system using the Z80180 MPU. This has a memory capacity of 1 MB which is more than adequate for really a large number of the embedded system applications. We shall look at a Z80180-based Micro Genius developed by Z-World Engineering. Together with a C cross-compiler running in the PC, this embedded controller provides a rather powerful system for real-time applications. The commercial version is compact in size (3.2" by 2") and relatively low cost (USD89). Another essential feature of this system is the provision of real-time multitasking capability by means of *costatements* and/or a *real-time kernel.*

9.2 System Hardware Configuration



A diagram of the hardware configuration is shown above. It has the following sub-units:

- A Z80180 which is an enhanced Z80 microprocessor outlined earlier.
- An RS232 port with the following features:
  - With RTS & CTS handshaking.
  - 9600, 19200 or 57600 baud.
  - It is used to communicate with PC during program development and can be subsequently programmed for other use.
- An RS485 port which provides half-duplex serial communication using balanced differential drives for distances up to 4 km.
- 32K bytes of RAM.
- Up to 512K bytes of EPROM or up to 256K bytes of flash EPROM. Flash EPROM is non-volatile and can be written under program control.
- The following parallel I/O (PIO) are available:
  - Two 8-bit ports, A and B.
  - Port A with handshaking.
  - 4 lines of port B are pre-assigned for real-time clock and RS485 use.
- A watchdog circuit restarts the system if software fails to reset the watchdog timer every 1.2 seconds. It also resets when Vcc falls below 4.62V.
- A 555 timer is used as an analogue-to-digital converter for interfacing with external resistive sensors.
- A real-time calendar clock acts as a timekeeper. It also provides 31 bytes of scratchpad RAM.

9.3 Program Development

Programs for the Micro Genius are developed using *Dynamic C* which is an integrated editor-compiler-debugger, run in Windows or DOS environment. When a program is compiled, it is downloaded directly to the RAM of the target system that is connected to one of the COM ports of the PC. Serial communication is at 9600, 19200 or 57600 baud.

When the program development is finished, the entire program may be compiled for EPROM. An EPROM may then be programmed in a separate process and plugged into the target system to run.

Three modes of program development are available:

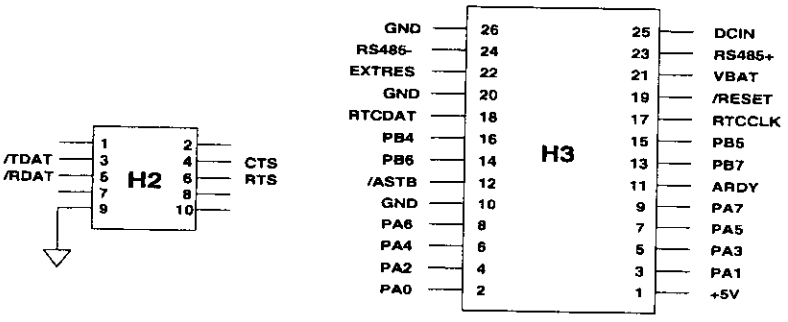
- **Use target system with EPROM**. Use the target system with a Dynamic C BIOS EPROM and connect the RS232 port directly to the PC. The RAM provides up to 32K of code and data space.

- **Use target system with flash EPROM**. Use the target system with a 256K flash EPROM and connect the RS232 port directly to the PC. In this case the flash EPROM provides 256K of program space and the RAM 32K of data space.

- **Use target system with a separate development board**. A development board that plugs into the EPROM socket of the target system, provides its own RS232 port for communicating with the PC and emulates the BIOS EPROM as well as providing 504K bytes of program space in addition to the 32K data space on the target RAM. In this case, both the serial ports of the target machine are free.

9.4 Interface Description

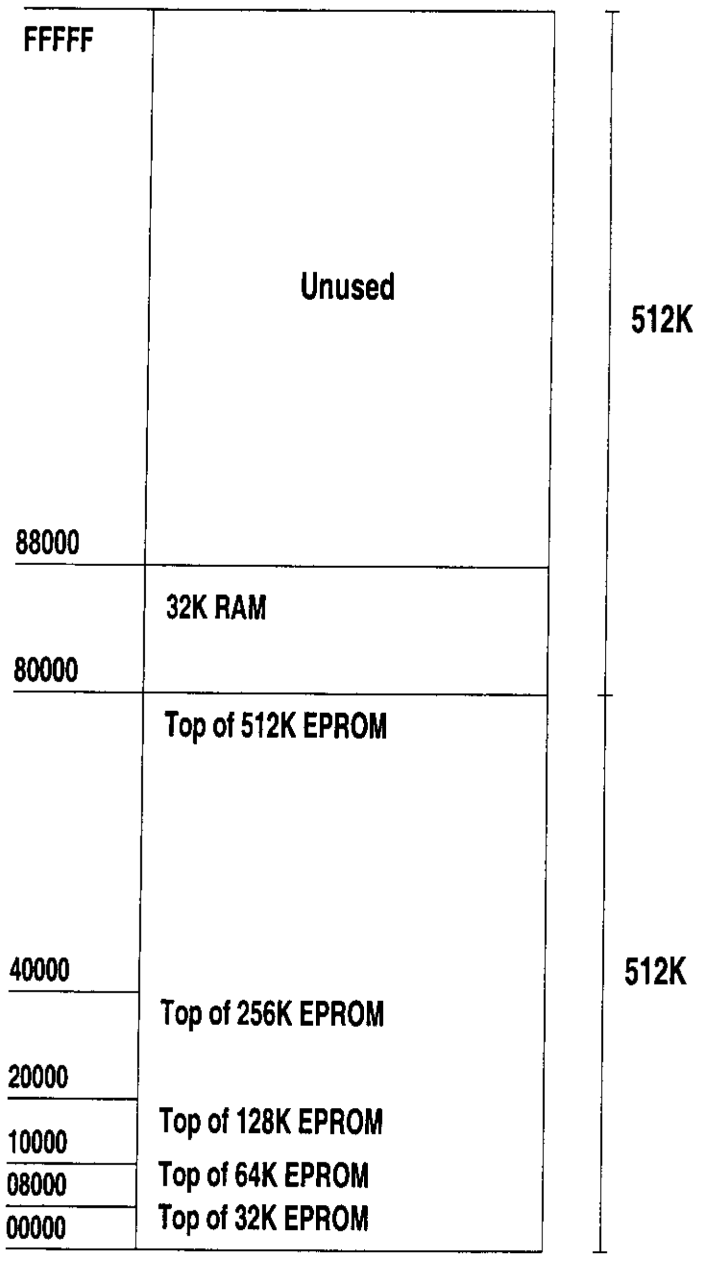
The interface of Micro Genius consists of bit- and byte-wise parallel I/O, serial ports, precision timer, and real-time clock. They are arranged in two headers (H2 and H3) as shown below:



- **RS232 & programming port** A 10-pin header (H2) provides a 5-wire RS232 interface. This interface can also be used as the programming port, in which case the communication port temporarily lost to the user program.
- **RS485 port** – An RS485 driver chip provides a half-duplex RS485 interface. An RS485 serial communication channel can be used to create a network of embedded systems with links spanning several kilometres. The RS485 signals are available on pins 23 and 24 of header H3.
- **Supervisor** – A supervisor (DS1232) provides a watchdog timer that guards against system or software faults by resetting the processor if software does not *hit* (by calling `hitwd`) the timer at least 1.2 seconds. It also resets the entire system on power-up or when Vcc falls below 4.62V.
- **Real–time clock** A real-time clock (DS1302) provides time and date function, plus 31 bytes of scratchpad RAM. An external battery (connected to VBAT) is used to retain data when power is down. Data are clocked using $\overline{\text{RTCCLK}}$ and $\overline{\text{RTCDAT}}$. $\overline{\text{RTCRST}}$ resets the real-time clock.
- **Timer** – A timer (555) is used to measure external resistance, such as a thermistor, control potentiometer, or a position sensor. It behaves like an analogue input channel. The resistance of the input device is deduced from the timer value using the following formula:
  $$\Delta = 1.1RC \text{ seconds where } C = 4.7\mu\text{F}$$
- **Parallel input/output ports** A PIO chip is used to provide parallel input/output ports.
  - Port A (PA0-7, ARDY, $\overline{\text{ASTB}}$) is a full I/O port with handshaking lines. PA0-7 are TTL compatible.
  - PB4-7 are available to user applications. Each line can supply up to 1.5mA at 1.5V to drive Darlington transistor.
  - PB0-3 are used as $\overline{\text{RTCRST}}$, EN485, RTCDAT and RTCCLK respectively.
  - Impedance of the I/O lines are 80Ω for sinking current and 160Ω for sourcing current.
  - Port A may be programmed to operate in mode 0 (strobed byte output), mode 1 (strobed byte input) or mode 3 (bitwise I/O). Port B is in mode 3.

9.4.1 Memory Map

- The memory map of the Micro Genius is as follows:



9.5 Driver Software For I/O Devices

An extensive set of C functions for programming the interfaces is available from Z-World Engineering. The following table forms a partial list.

FUNCTION	DESCRIPTION
void setPIOCA(byte mask)	Set port A control register.
void resPIOCA(byte mask)	Reset port A control register.
void setPIODA(byte mask)	Set port A data register.
void resPIODA(byte mask)	Reset port A data register.
void setPIOCB(byte mask)	Set port B control register.
void resPIOCB(byte mask)	Reset port B control register.
void setPIODB(byte mask)	Set port B data register.
void resPIODB(byte mask)	Reset port B data register.
int tm_rd(struct tm *t)	Read the RTCinto the structure *t.
int tm_wr(struct tm *t)	Write the values in the structure *t.
int WriteRAM1302(int ram_loc, byte data)	Write data to any of the 31 RAM locations of the DS1302.
int ReadRAM1302(int ram_loc)	Read data from any of the 31 RAM locations of the DS1302.
void WriteBurst1302(void*pdata, int count)	Write count bytes, in burst mode,to the DS1302.
void ReadBurst1302(void*pdata, int count)	Read count bytes, in burst mode,to the DS1302.
void Write1302(int reg, byte data)	Write data to a specific register of the DS1302.
int Read1302(int reg)	Read data from a specific register of the DS1302.
charger1302(int on_off, int diode, int resistor)	Turns the trickle charger on the DS1302 on.
void Set555(uint max_count)	Trigger the 555 circuit and start the Z180 timer.
int Read555(uint *lapse_count)	Read Z180 timer.

```
struct tm {
  char tm_sec;   //0-59   char tm_min;   //0-59
  char tm_hour;  //0-23
  char tm_mday;  //1-31
  char tm_mon;   //1-12
  char tm_year;  //0-150 (1900-2050)
  char tm_wday;  //0-6 where 0 means Sunday
};
```

9.6 Serial Communication Software

The serial communication library includes the following functions:

- Initialization of the serial ports.
- Monitoring and reading a circular receive buffer.
- Monitoring and writing to a circular transmit buffer.
- An echo option.
- CTS (clear to send) and RTS (request to send) control.
- XMODEM protocol for downloading and uploading data. Downloading of data is in multiple of 128 bytes. Uploaded data is written to specified area in RAM.
- A modem option.

Serial communication is done by a background interrupt routine that updates receive and transmit buffers. Using the CTS/RTS option, the RTS will be pulled high when the receive buffers has reached 80% of its capacity. The RTS line is pulled low again when the received buffer has gone below 20% of its capacity.

The RS232 library supports communication with Hayes Smart Modem. The CTS, RTS and DTR lines of the modem are not used. They are tied together. The CTS and RTS lines on the Micro Genius are also tied together. A NULL connection is required for the TX and RX lines.

9.7 Master-Slave Networking

Functions for master-slave two-wire half-duplex RS485 9th-bit binary communication are also available. In a network, one system is configured as master (address 0) and the rest as slaves (address 1-255). The data transfer scheme is as follows:

- Z180 is initialized for RS485 communication.
- The master sends an enquiry and waits for a response.
- Slaves monitor for their address during the 9th-bit transmission. The slave that matches the address will listen to the rest of the message and reply to the master.

- The format of a master message:
  `[slave id] [len][   ] [   ] ...[   ] [CRC hi] [CRC lo]`
- The format of a slave message:
  `[len] [   ] [   ] ...[   ] [CRC hi] [CRC lo]`

9.8 Dynamic C Development System

As mentioned earlier, Dynamic C is an integrated development system comprising a C compiler, an editor, and a source-level debugger. In the Windows version, it has eight menu: File, Edit, Search, Compile, Run/Debug, Watch, Options, and Window. It compiles, links and downloads to the target machine under the same environment.

Embedded assembly language is supported (#ASM #ENDASM directives). C statements can be placed within assembly code by placing a C in column 1. It supports *hard* and *soft* breakpoints where the former disables interrupts whereas the latter leaves interrupts on so that higher priority tasks can continue to execute.

Debugging supported by printf and *watch* expressions. A watch expression is a C language expression that can include preprocessor substitutions, variables and function calls.

9.9 Extension To C for Extended Memory Data

Extension to C allows the access of extended memory data. Extended memory addresses are 20-bit physical addresses. Pointers are 16-bit machine addresses. Two non-standard keywords are used for this purpose: *xstring* and *xdata*.

xstring *name* { *string1, ... stringn* };

defines an array of string addresses. The term *name* is the name of the array, itself a 32-bit unsigned long integer whose lower 20 bits are the address of the array.

xdata *name* { *datum1, ... datumn* }

defines an array of addresses of initialized extended memory data. The data must be constant expressions.

xdata *name [n];*

defines a block of *n* bytes in extended memory.

9.10 Multitasking In Micro Genius

Both *preemptive* and *cooperative* multitasking are supported. In preemptive multitasking, tasks are interrupted and control is taken away involuntarily. A kernel is needed to monitor, regulate and dispatch tasks. A real-time kernel (RTK) included in the Dynamic C library supports prioritized preemption. As many priority levels as desired may be used.

A simplified real-time kernel (SRTK) is also available. There are only three levels of priority in this case. The top priority task executes at 25ms intervals, the low priority task executes at 100ms intervals. The background task executed when no other tasks are executing.

A special *fastcall* task is available that can execute as often as 1280 times per second. It preempts all other tasks.

In cooperative multitasking, each task voluntarily gives up control so that other tasks can execute. A kernel is not required. This method provides easier communications between tasks and is simpler to program. However it requires a *costatement* mechanism to function. Costatement mechanism is another extension to C provided by Dynamic C compiler.

9.11 Costatement Mechanism

Costatements are an extension to C that facilitate cooperative multitasking. Costatements are cooperative concurrent tasks that can suspend their own operation:

- They can **waitfor** event, condition, or the passage of time.
- They can **yield** temporarily to other costatements.
- They can **abort** their own operation.

Costatement can be active (ON) or inactive (OFF). For each costatement, there is a structure of type **CoData** associated with it. It maintains a position pointer to resume execution after being stopped. It also carries a start flag and other data in the following syntax:

costate[*name*[***state***]]**{**
[*statement*|**yield;**|**abort;**|**waitfor(*expression*);**] ...**}**

Three delay functions can be used by **waitfor**:

int DelaySec(ulong seconds);
int DelayMs(ulong milliseconds);
int DelayTicks(uint ticks);

9.12 A Real-time Problem Without Using Costatements

Consider the following sequence of events to be programmed:

- Wait for a push-button to be pushed.
- Turn on device 1.
- Wait for 60 seconds
- Turn on device 2.
- Wait for 60 seconds.
- Turn off both devices.
- Go to the beginning.

The above can be written in C without using costatements as follows:

```
// Normal C program without using costatement
        extern shared long time;
        long timer1, timer2;
        int state;
// Intialization:
        state=1;
        for(;;){
                if(state==1){
                        if(buttonpushed()){
                                state=2;
                                turnondevice1();
                                time1=time;
                        }
                } else if(state==2){
                        if((time-timer1)>=60L)}
                                state=3;
                                turnondevice2();
                                timer2=time;
                        }
                } else if(state==3){
                        if((time-timer2)>=60L{
                                state=1;
                                turnoffdevice1();
                                turnoffdevice2();
                        }
                }
        }
```

9.13 Real-time Problem Using Costatements

Now if the above sequence is just one of several tasks to be performed, the code above has to be modified, often involving changes to keep track of the state or time. Using costatement, the entire problem can be solved elegantly as follows:

```
// Using costatements
        for(;;) {
                costate {                                    // task 1
                          waitfor(buttonpushed());
                          turnondevice1();
                          waitfor(DelaySec(60L));
                          turnondevice2();
                          waitfor(DelaySec(60L));
                          turnoffdevice1();
                          turnoffdevice2();
                }
                costate {                                    // task 2}
                        ...
                }
                ...
                costate{                                     // task n}
                        ...
                }
        }
```

9.14 The Virtual Driver In Micro Genius

The virtual driver (invoked by VDInit) is a set of functions that provides the following services:

- Periodic time interrupts
- Second, millisecond and tick timers
- Synchronization of the second timer with the real-time clock
- Virtual watchdog timers
- Periodic drive for real time kernels
- A fastcall execution thread
- Global initialization

The virtual driver is called 1280 times per second by a clock interrupt. If no real-time kernel, fastcall, or virtual watchdog is in use, the virtual driver just updates the second, millisecond and tick timers.

If **#define RUNKERNEL 1** is included in a program that uses the virtual driver, it will call the RTK or SRTK every 25 milliseconds.

9.15 Real-time Kernels In Micro Genius

The RTK and SRTK allow program to be divided into prioritized tasks. Execution of these tasks is *interleaved* in time. An example of using SRTK is given below:

```
#use vdriver.lib                    // or include VDRIVER.LIB and
#use srtk.lib                       // SRTK.LIB in LIB.DIR

#define RUNKERNEL 1                 // use the kernel

int HCOUNT, LCOUNT;

main(){
        HCOUNT=LCOUNT=0
        VdInit();                   // Need virtual driver
        init_srtkernel();           // Initialize the SRTK
        while(1){ ... }
}

// This high priority task executes every 25 ms
        srtk_hightask(){HCOUNT++;}

// This low priority task executes every 100 ms
        srtk_lowtask(){
                LCOUNT++;
                costate{        // Print every 1/2 second
                        waitfor(DelayMs(500));
                        printf("%d %d\n", HCOUNT, LCOUNT);
                }
                costate{        // Reset when HCOUNT is large
                        waitfor(HCOUNT>=32000);
                        HCOUNT=0;
                        LCOUNT=0;
                }
        }
```

The costatements create two execution threads within the low priority task. Background tasks can be placed in the **while** loop in **main**. To use the RTK, three steps must be taken:

- define an array of task pointers
- specify the number of tasks
- **#define RUNKERNEL**

An example using the RTK is shown below:

```
#define NTASK 7
#define RUNKERNEL 1
#use RTK.LIB

// Task prototypes
        int heater(), pump(), sensor(), backgnd();

// Array of 4 task pointers
        int(*Ftask[4])()={heater,              // task 0
                             pump,             // task 1
                             sensor,           // task 2
                             backgnd};         // task 3

/****** WITH VIRTUAL DRIVER *****/
main(){
        VdInit();                              // initialize VD and RTK

        run_every(0,5);                        // run task 0 every 5 ticks
        run_every(1,15);                       // run task 1 every 15 ticks
        run_every(2,100);                      // run task 2 every 100 ticks

        backgnd();                             // run lowest priority task 3
}
```

Kernel functions related to the RTK are

- **void run_at(int tasknum, voidtime)**
- **int comp48(void*time1, voidtime2)**
- **void gettimer(voidtime)**
- **void run_after(int tasknum, long delay)**
- **void run_every(int taksnum, int period)**
- **void request(uint tasknum)**
- **void run_cancel(int tasknum)**
- **void suspend(uint ticks)**

10 A Real-time Kernel for Embedded Systems - μC/OS

10.1 Introduction

Jean J. Labrosse published an early version of μC/OS in *Embedded Systems Programming* magazine in June 1992. It was written in C with the initial goal for creating a small but powerful kernel for the 68HC11 microcontroller. It has since been extended to a portable system suitable for use with any microcontroller/microprocessor provided that it has a stack pointer and the processor status can be stacked and unstacked.

Labrosse has subsequently written the book describing μC/OS:

- Jean J. Labrosse, $\mu\, C/OSTheReal-TimeKernel$, R & D Publications, Lawrence, Kansas. ISBN 0-13-031352-1

The complete source listing of μC/OS is available in the book. It is also available in a companion disk.

The code is protected by copyright. However, you do not need a license to use the code in your application if it is distributed in object format. You should indicate in you document that you are using μC/OS.

10.2 Main Features of μC/OS

The main features of μC/OS are:

- **Portable** – It is written in C, with a small processor specific code in assembly to create task, start multitasking and perform context switching. For 80186/80188 the assemble language code is less than 4 pages.

- **ROMable** – The size and design of the kernel is such that it is suitable for storing in ROM or EPROM.

- **Priority driven** – It always runs the highest priority task that is ready.

- **Pre-emptive** – When a task makes a higher priority task ready to run, the current task is pre-empted or suspended and the higher priority task is immediately given control of the processor. Execution of the highest priority task is deterministic.

- **Multitasking** – Up to 63 tasks may be set up.

- **Interrupt feature** – Interrupts can suspend the execution of a task. If a higher priority task is awakened as a result of the interrupt, the higher priority task will run as soon as the interrupt completes. Interrupts can be nested up to 255 levels deep.

10.3 μC/OS Tasks

A *task* is an infinite loop function or one that deletes itself when it is finished. The infinite loop can be pre-empted by an interrupt that can cause a higher priority task to run as mentioned above. A task can also call the following μC/OS services:

- **OSTaskDel()**
- **OSTimeDly()**
- **OSSemPend()**
- **OSMboxPend()**
- **OSQPend()**

Each task has a unique priority, ranging form 0 to 62. The lower the value the higher the task priority.

10.4 μC/OS Task States

There are altogether six possible states for a task as listed below:

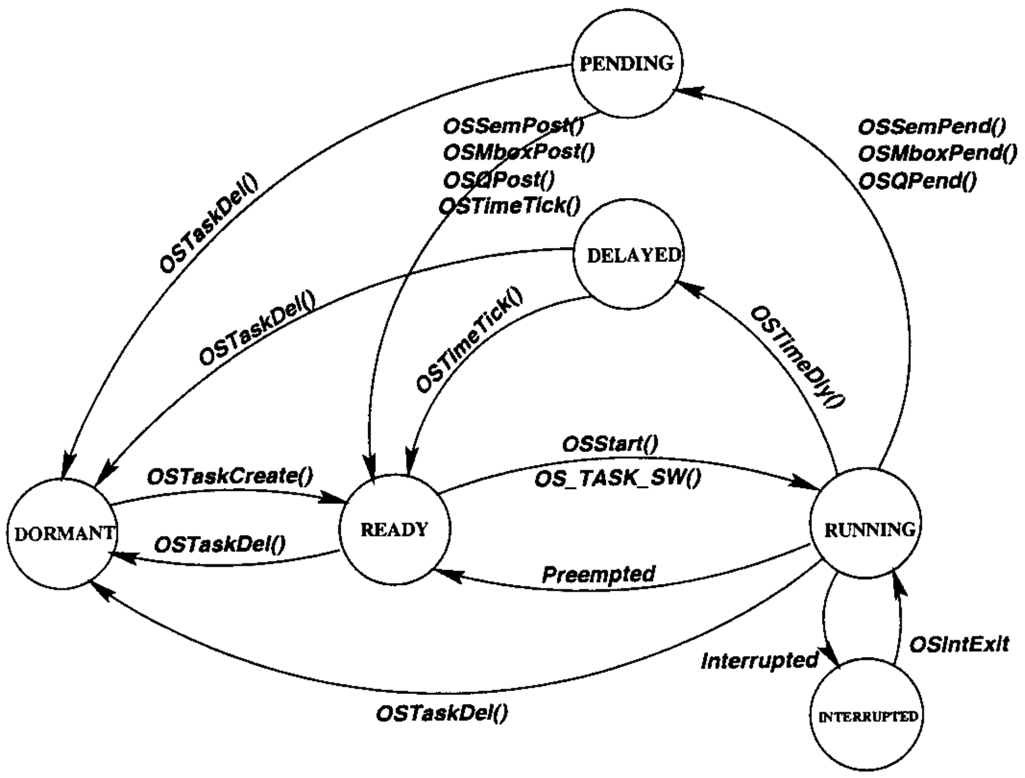
- DORMANT -The state when a task has not been made available to μC/OS.
- READY - When a task is created by calling **OSTaskCreate()**, it is in the READY state. Tasks may be created before multitasking starts or dynamically by a running task. If the created task has a higher priority than its creator, the created task is immediately given the control of the processor. A task can return itself or another task to the DORMANT state by calling **OSTaskDel()**.
- RUNNING - The highest priority task created is in the RUNNING state when multitasking is started by calling **OSStart()**.

- DELAYED -The running task may call **OSTimeDly()** and enters the DELAYED state. The next highest priority task then runs. The delayed task is made ready to run by **OSTimeTick()** when the desired delayed time expires.

- PENDING - The running may have to wait for an event by calling **OSSemPend()**, **OSMboxPend()** or **OSQPend()**. It then enters the PENDING state. The next highest priority task then runs. The task is made ready when the event occurs. The occurrence of an event may be signalled by another task or by an interrupt service routine (ISR).

- INTERRUPTED - A task may be interrupted and enters the INTERRUPTED state. The ISR then runs. The ISR may make one or more tasks ready to run. When all tasks are either waiting for events or delayed, an idle task **OSTaskIdle()** is executed.

10.5 μC/OS Task State Transition Diagram



10.6 Task Control Block

Each task has a task control block, **OS_TCB**, which is used by *mu*C/OS to maintain the state of the task when it is pre-empted. When the task regains control the **OS_TCB** allows it to resume execution properly.

Each **OS_TCB** has the following field:

- **OSTCBStkPtr** – points to the top of stack.
- **OSTCBStat** – state of the task. 0 - ready to run
- **OSTCBPrio** – task priority. 0 - 63
- **OSTCBDly** – number of clock ticks the task is to wait for an event.
- **OSTCBX, OSTCBY, OSTCBBitX, OSTCBBitY** – used for speeding up task handling by precomputing some parameters.

  OSTCBX = priority & 0x07;
  OSTCBBitX = OSMapTle[priority & 0x07];
  OSTCBY = priority >> 3;
  OSTCBBitY = OSMapTbl[Priority >>3];

- **OSTCBNext, OSTCBPrev** – to doubly link **OS_TCBs. OS-TimeTick()** uses this link to update **OSTCBDly** field for each task.
- **OSTCBEventPtr** – points to an event control block.

All **OS_TCBs** are placed in **OSTCBTbl[]**. The maximum number of task is declared in the user's code. An extra **OSTCB** is allocated for the idle task.

10.7 Creating a Task

Tasks are created by calling **OSTaskCreate()** which is target processor specific. Tasks can either be created prior to the start of multitasking or by another task at run time. A task cannot be created by an interrupt service routine.

OSTaskCreate() has four arguments:

- **task** – points to the task code.
- **data** – points to a user definable data area that is used to pass arguments to the task.

- **pstk** – points to the task stack area for storing local variables and register contents during an interrupt.
- **p** – task priority.

OSTaskCreate() calls **OSTCBInit()** which obtains an **OS_TCB** from the list of free **OS_TCBs**. If all **OS_TCBs** have been used, an error code is returned. If an **OS_TCB** is available, it is initialised.

A pointer the **OS_TCB** is placed in the **OSTCBPrioTble[]** using the task priority as the index. The **OS_TCB** is then inserted in a doubly linked list with**OSTCBList** pointing to the most recently created **OS_TCB**. The task is then inserted in the ready list.

If a task is created by another task, the scheduler is called to determine if the created task has a higher priority than its creator. If so, the new task is executed immediately. Otherwise, control is returned to its caller.

10.8 Deleting a Task

A task may return itself or another task to the DORMANT state by calling **OSTaskDel()**. However, the idle task cannot be deleted. The steps taken to removed a task is as follows:

- Removed from the ready list.
- **OS_TCB** is unlinked and returned to the list of free **OS_TCB**.
- If **OSTCBEventPtr** field in nonzero, the task must be removed from the event waiting list.

10.9 Task Scheduling

Task scheduling is done by**OSSched()** which determines which task has the highest priority and thus will be the next to run. Each task has a unique priority number between 0 and 63. Priority 63, the lowest, is assigned to the idle task when μC/OS is initialised.

Each task that is ready to run is placed in a ready list. The task scheduling time is constant irrespective of the number of tasks created. **OSSched()** looks for the highest priority task and verifies that it is not the current task to prevent unnecessary context switch. A context switch is then carried out by**OS_TASK_SW()**.

OSSched() runs in a critical section to prevent ISR from changing the ready status of a task.

10.10 Interrupt Processing

μC/OS requires an *interrupt service routine* (ISR) written in assembly language. Interrupts are enabled early in the ISR to allow other higher priority interrupts to enter.

OSIntEnter() is called on entering and **OSIntExit()** on leaving the ISR to keep track of the interrupt nesting level. There may be 255 levels.

μC/OS's worst case interrupt latency is 550 MPU clock cycles (80186/80188). μC/OS's worst case interrupt response time is 685 MPU clock cycles (80186/80188).

10.11 Clock Tick

Time measurement in suspending execution and in waiting for an event is provided by **OSTimeTick()**, which supplies the *clock ticks* or the heartbeats. **OSTimeTick()** also decrements the **OSTCBDly** field for each **OS_TCB** that is not zero.

The time between tick interrupts is application specific and is typically between 10 ms and 200 ms. **OSTimeTick()** increments a 32-bit variable OSTime since power up. This provides a system time.

10.12 Communication and Synchronisation

μC/OS supports message *mailboxes* and *queues* for communication. A task can deposit, through a kernel service, a message (the pointer) into the mailbox. Similarly, one or more tasks can received messages through a service provided by the kernel. Both the sending and receiving task have to agree as to what the pointer is pointing to.

A message queue is an array of mailboxes. μC/OS supports *semaphore* (0–32767) for synchronisation and coordination.

The above services are *events*. Thus, a task can signal the occurrence of an event **(POST)** or wait for an event to occur **(PEND)**. However, the ISR can **POST** an event but cannot **PEND** on an event.

When an event occurs, the highest priority task waiting for the event is made ready to run.

10.13 Event Control Blocks

The state of an event consists of the event itself and a waiting list for tasks waiting for the event to occur.

Each event is assigned an Event Control Block which has the following data structure:

- **OSEventGrp**
- **OSEventGrp**
- **OSEventTbl**[8]
- **OSEventCnt** for semaphore count
- **OSEventPtr** for mailbox or queue

10.14 Memory Requirements

The memory required for the program is less than 3150 for the 80186/80188 microcontroller. This can be reduced to if some of the services are not required. The RAM or data memory is as follows:

- 200
- + ((1 + OSMAX_TASK) * 16)
- + (OS_MAX_EVENTS * 13)
- + (OS_MAX_QS * 13)
- + SUM(Storage requirements for each message queue)
- + SUM(Storage requirements for each task stack)
- + (OS_IDLE_TASK_STK_SIZE)

10.15 Kernel Services

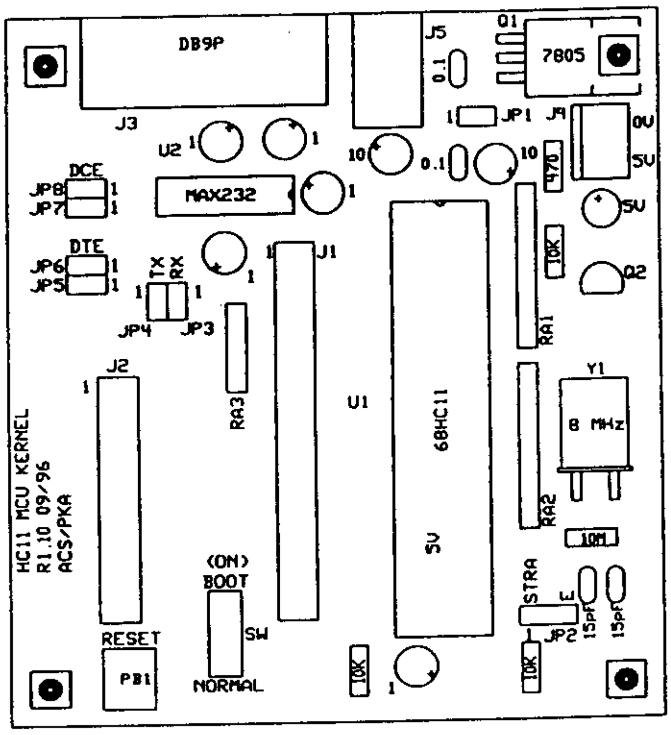
The kernel services are given in the following table:

#	SERVICE	DESCRIPTION
1	**OSInit()**	Initialise µC/OS
2	**OSIntEnter()**	Signal ISR entry
3	**OSIntExit()**	Signal ISR exit
4	**OSMboxCreate()**	Create a mailbox
5	**OSMboxPend()**	Pend for mrssage from mailbox
6	**OSMboxPost()**	post a message to mailbox
7	**OSQCreate()**	Create a queue
8	**OSQpend()**	Pend for message from queue
9	**OSQPost()**	Post a message to queue
10	**OSSchedLock()**	Prevent rescheduling
11	**OSSchedUnlock()**	Allow rescheduling
12	**OSSemCreate()**	Create a semaphore
13	**OSSemPend()**	Wait for a semaphore
14	**OSSemPost()**	Signal a semaphore
15	**OSStart()**	Start multitasking
16	**OSTaskChangePrio()**	Change a task's priority
17	**OSTaskCreate()**	Create a task
18	**OSTaskDel()**	Delete a task
19	**OSTimeDly()**	Delay a task for n system ticks
20	**OSTimeGet()**	Get current system time
21	**OSTimeSet()**	set system time
22	**OSTimeTick()**	Process a system tick

A HC11 Embedded System PCB Artwork

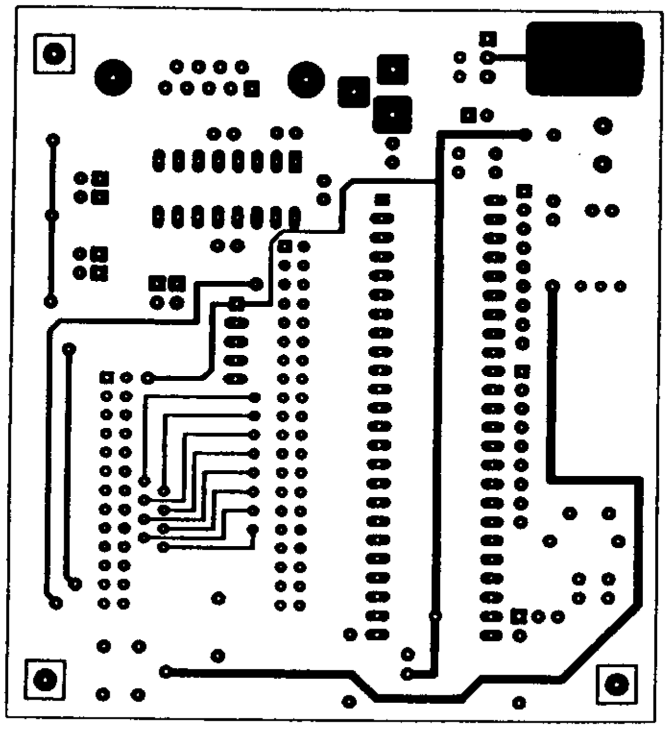
A.1 HC11 MCU Kernel Board

LEGEND

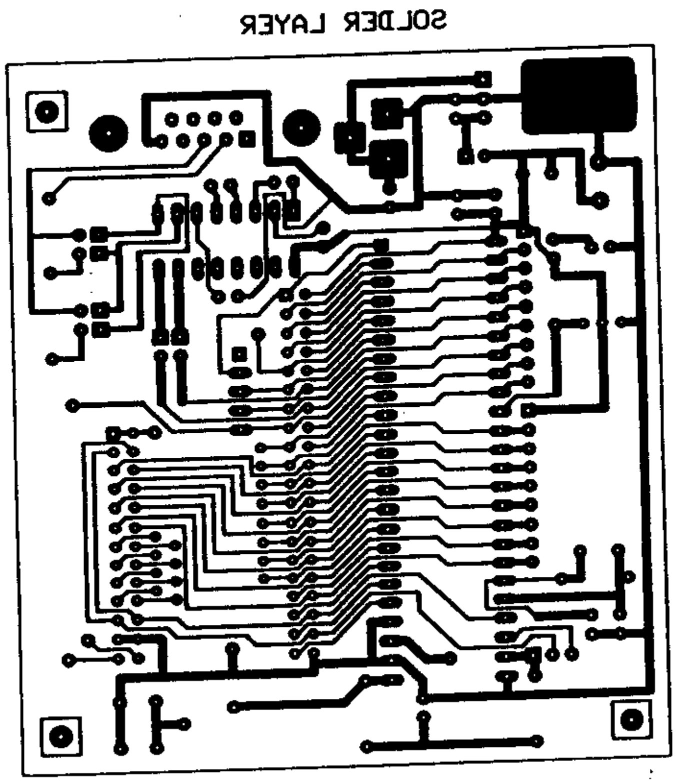


A.2 HC11 MCU Kernel Board

COMPONENT LAYER

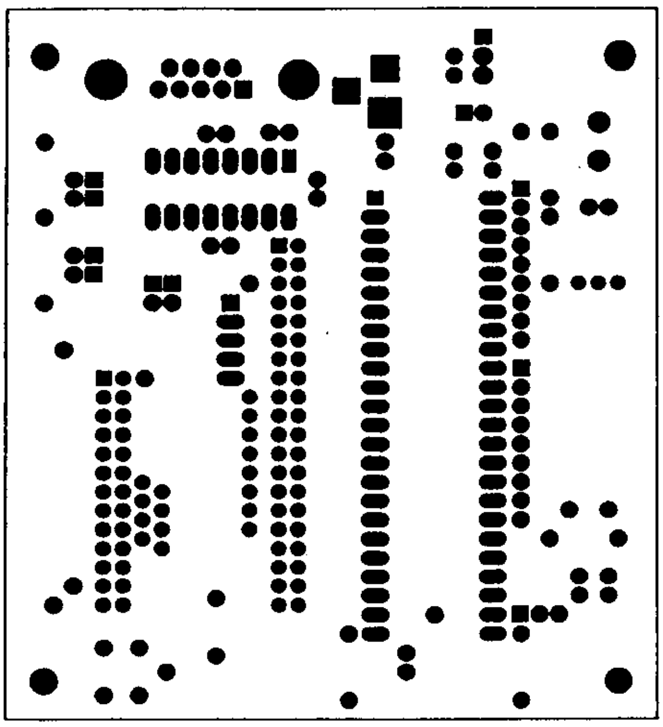


A.3 HC11 MCU Kernel Board

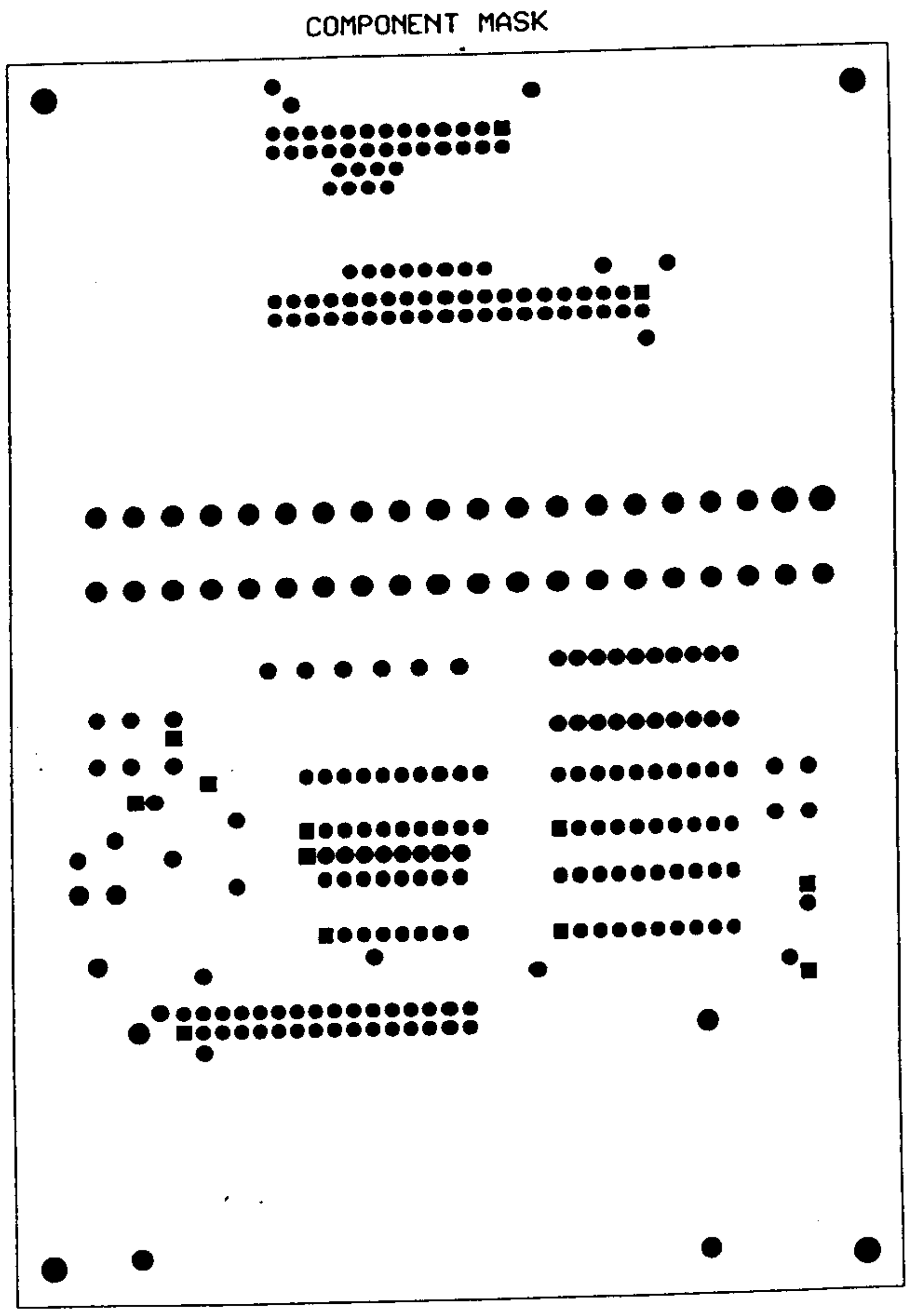


A.4 HC11 MCU Kernel Board

COMPONENT MASK

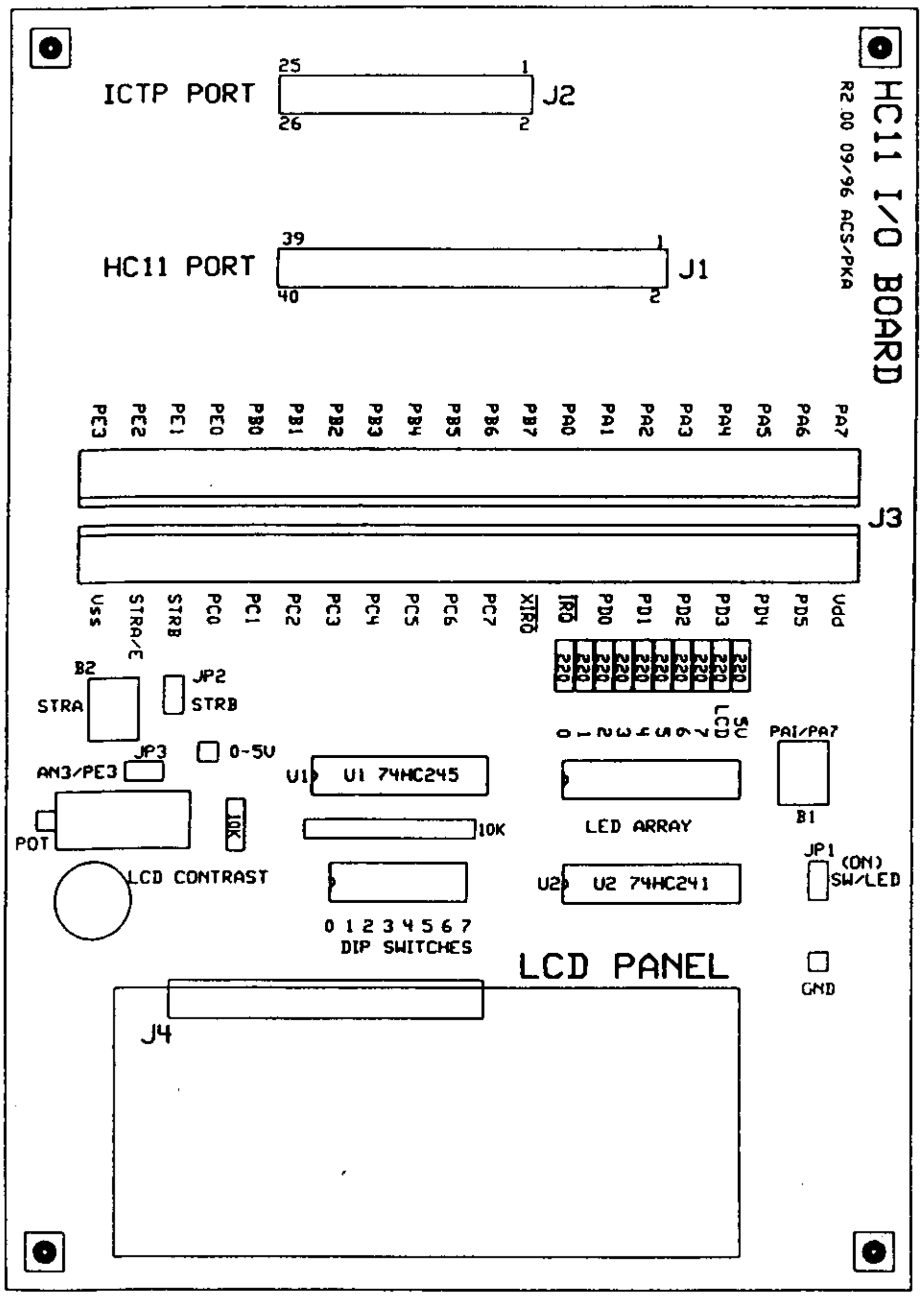


A.5 HC11 I/O Board



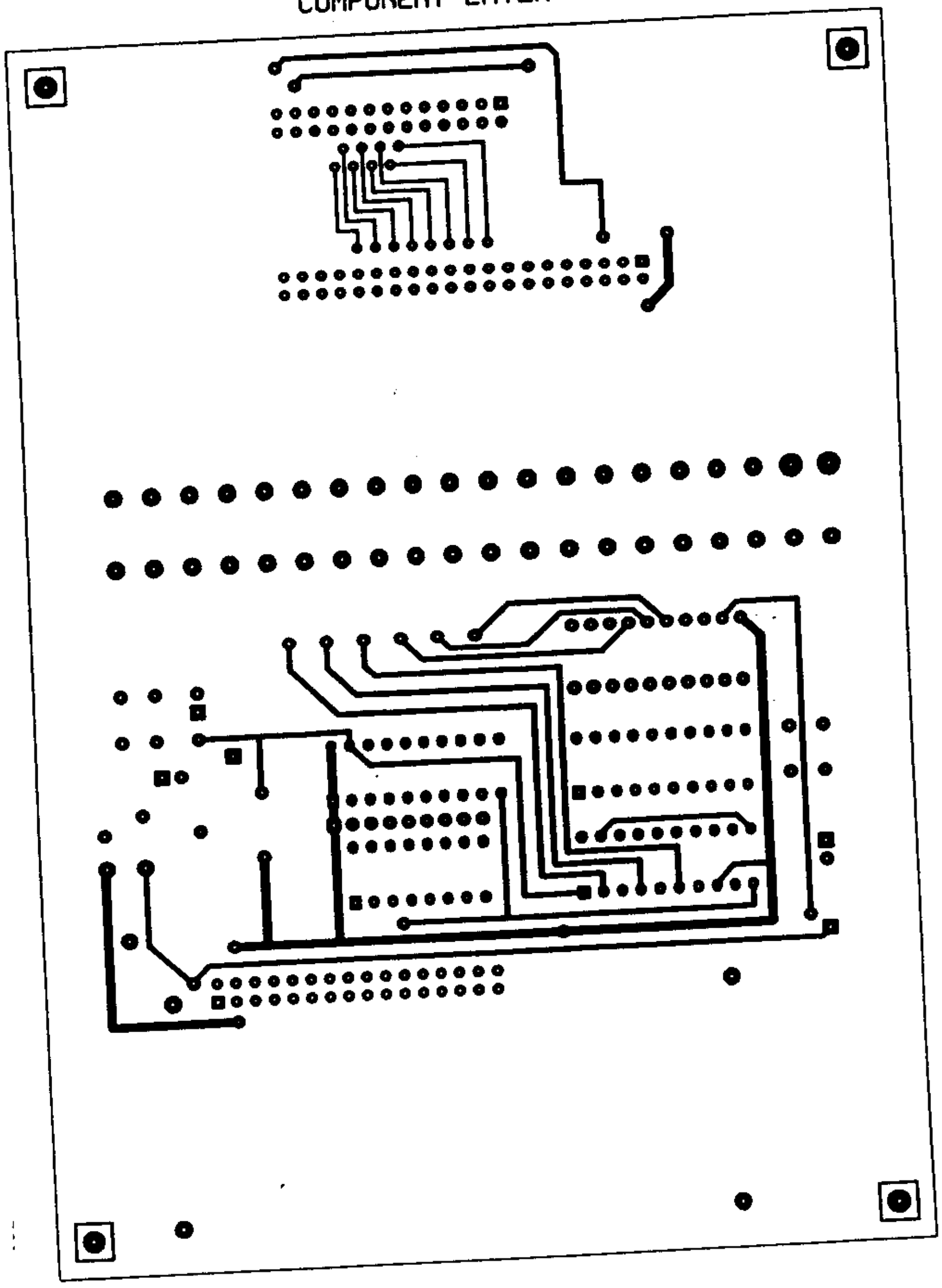
A.6 HC11 I/O Board

LEGEND

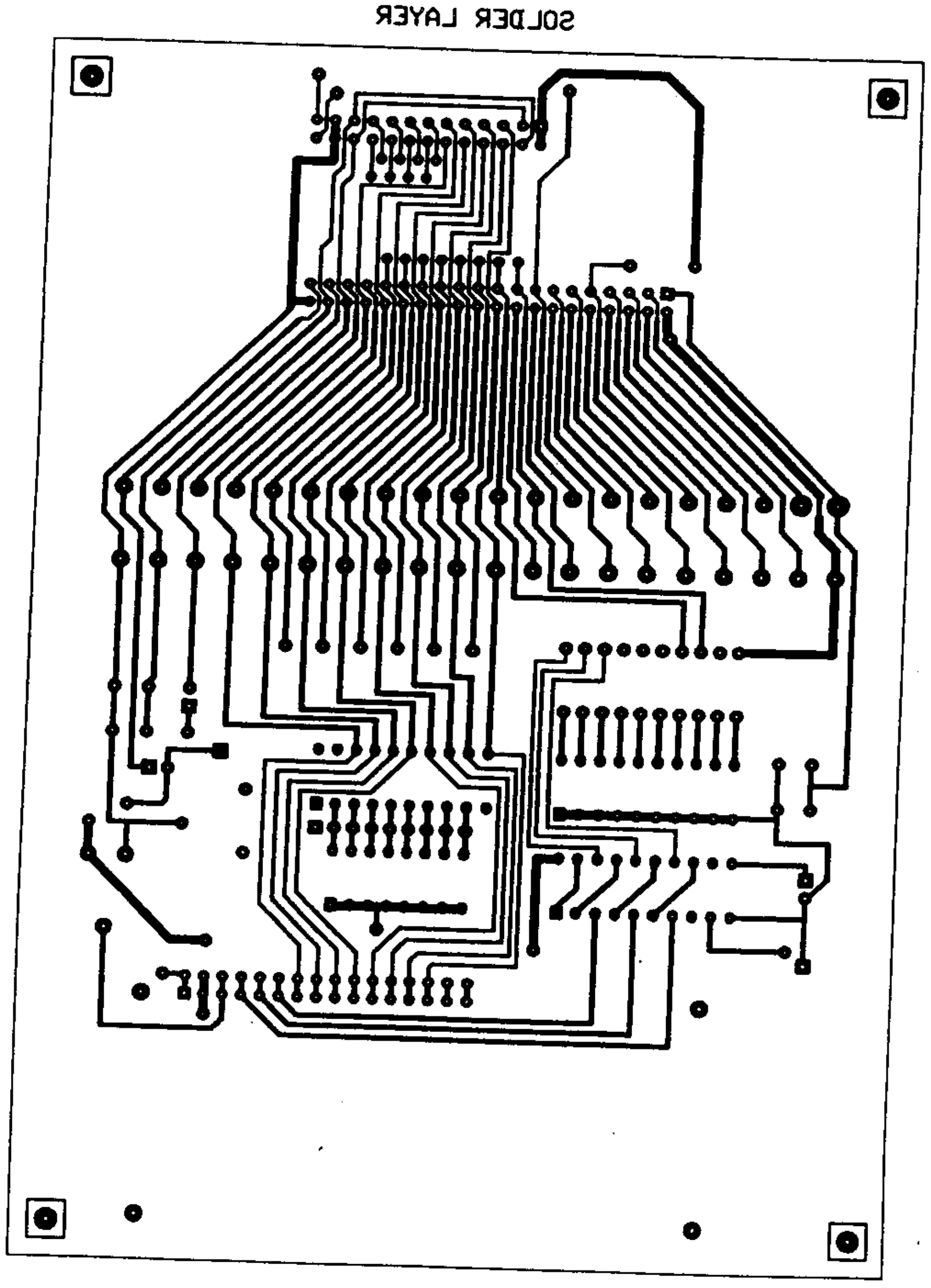


A.7 HC11 I/O Board

COMPONENT LAYER



A.8 HC11 I/O Board



B Software Utilities

Appended below are programm listing in the most primitive level for HC11 development.

B.1 PRGHC811 EEPROM Programmer

```
*******************************************************************************
* Program   : 68HC811E2 EEPROM programmer                                     *
* Filename  : PRGHC811.ASM                                                    *
* Version   : 1.00 on 12/9/90                                                 *
*           : 1.10 on 24/10/90                                                *
* Written by: K.A. Poh                                                        *
*                                                                             *
* Binary image of this program is downloaded to the 68HC811E2 in bootstrap    *
* mode.                                                                       *
* It then read in S19 file (application program) and program the EEPROM.      *
*******************************************************************************
*
***************
*   EQUATES   *
***************
RAMBS           EQU   $0000       start of ram
REGBS           EQU   $1000       start of registers
BOOTROM         EQU   $BF40       start of bootstrap ROM routines

***  Registers will be addressed with Ind,X mode  ***
BAUD            EQU   $2B         sci baud reg
SCCR1           EQU   $2C         sci control1 reg
SCCR2           EQU   $2D         sci control2 reg
SCSR            EQU   $2E         sci status reg
SCDR            EQU   $2F         sci data reg
BPROT           EQU   $35         EEPROM block protection reg
OPTION          EQU   $39         config option reg
PPROG           EQU   $3B         eeprom prog reg
HPRIO           EQU   $3C         highest priority reg
TEST1           EQU   $3E         test functions control reg
CONFIG          EQU   $3F         config reg
TDRE            EQU   $80
RDRF            EQU   $20
MDA             EQU   $20
SMOD            EQU   $40
mS10            EQU   10000/3      10mS delay
Null            EQU   0

***************
*     RAM     *
***************
```

```
               ORG   RAMBS
EE_OPT         RMB   1
MASK           RMB   1
TEMP           RMB   1
LAST_BYTE      RMB   1

               PAGE
***********************************
*  PRGHC811 PROGRAMS START HERE  *
***********************************
               ORG   RAMBS
               LDS   #$FF          init stack
               LDX   #REGBS
               CLR   SCCR1,X       8 data bits, 9600 baud
               LDD   #$300C
               STAA  BAUD,X
               STAB  SCCR2,X
Read_Opt       STS   EE_OPT        default EE_OPT=0, MASK=$FF
               BSR   Read_C        chk control byte
               CMPB  #'P'          program EEPROM ?
               BEQ   Load
               CMPB  #'V'          verify EEPROM ?
               BNE   Read_Opt
               DEC   EE_OPT

Load           EQU   *
               BSR   Read_C
               CMPB  #'S'          wait until S1 or S9 received
               BNE   Load
               BSR   Read_C
               CMPB  #'1'
               BEQ   Laod1
               CMPB  #'9'
               BNE   Load
               BSR   Rd_Byte       complete reading S9 record before ending
               TBA
               SUBA  #2            no.of bytes to read including chksum
               BSR   Get_Addr      get execution address in Y
Load9          BSR   Rd_Byte       discard remaining bytes
               DECA
               BNE   Load9
               BEQ   Read_Opt

Laod1          EQU   *
               BSR   Rd_Byte       read byte count of S1 record into ACCB
               TBA
               SUBA  #3            minus load addr & chksum from count
               BSR   Get_Addr      ge load addr into X
               DEY
```

```
                BRA   Load1B

Load1A          LDAB  EE_OPT
                BMI   Verify
Data_Poll       LDAB  ,Y
                EORB  LAST_BYTE
                ANDB  MASK
                BNE   Data_Poll
Load1E          DECA
                BEQ   Load
Load1B          BSR   Rd_Byte          read nx. byte
                INY                    nx. load addr
                TST   EE_OPT
                BMI   Load1D           if verifying then don't program byte
                BEQ   Prog             if internal EEPROM selected then program
Load1D          STAB  LAST_BYTE        save it for data polling
                BRA   Load1A

Verify          LDAB  ,Y
                CMPB  LAST_BYTE        if programmed byte is correct then
                BEQ   Load1E           read nx byte
                BSR   Write_C          else send bad byte back to host
                BRA   Load1E           before reading nx. byte

Read_C          EQU   *                ACCA, X, Y regs unchanged by this routine
                BRCLR SCSR,X RDRF *
                LDAB  SCDR,X
Write_C         BRCLR SCSR,X TDRE *
                STAB  SCDR,X           echo it back to host
                RTS

Rd_Byte         BSR   Read_C           read most significant nibble
                BSR   Hex_Bin
                LSLB
                LSLB
                LSLB
                LSLB
                STAB  TEMP
                BSR   Read_C
                BSR   Hex_Bin
                ORAB  TEMP
                RTS

Get_Addr        EQU   *
                PSHA                   save byte counte
                BSR   Rd_Byte          read MSB of addr
                TBA
                BSR   Rd_Byte          read LSB of addr
                XGDY
                PULA
                RTS

Hex_Bin         EQU   *
                CMPB  #'9'           if ACCB>9then assume it is A-F
                BLS   Hex_Num
                ADDB  #9
Hex_Num         ANDB  #$F
                RTS

Prog            EQU   *
                PSHA
                CLR   BPROT,X        remove protection on EEPROM
                CLR   PPROG,X
                CMPB  ,Y
                BEQ   ProgB          if same data then skip programming
                LDAA  #$16
ProgA           BSR   Program        erase byte
                LDAA  #2
                BSR   Program        program byte
ProgB           LDAA  #1F
                STAA  BPROT,X
                CPY   #CONFIG+REGBS
                BNE   ProgX
                LDAB  ,Y             load ACCB with old value to prevent hangup
ProgX           PULA
                BRA   Load1D

Program         EQU   *
                STAA  PPROG,X
                STAB  ,Y
                INC   PPROG,X        enable programming voltage
                PSHX
                LDX   #mS10          wait 10 mS
Wait            DEX
                BNE   Wait
                PULX
                DEC   PPROG,X        disable programming voltage
                CLR   PPROG,X
                RTS
                PAGE
***************
*   VECTORS   *
***************
                ORG   RAMBS+$F7
VILLOP          JMP   BOOTROM
VCOP            JMP   BOOTROM
VCLM            JMP   BOOTROM
```

B.2 HC11 Test Program

This is a simple program that tests or exercises all the I/O devices in the HC11 I/O board using the HC11 Kernel.

```
*****************************************************************************
* Program     : MC68HC11 MCU Kernel and I/O Board Test Program              *
* Filename    : HC11_TST.ASM                                                *
* Version     : 2.00 on 9/96                                                *
* Written by  : K.A.Poh, C.S.Ang                                            *
* Description:                                                              *
*                                                                           *
* This program tests the peripheral devices of the I/O Board with the       *
* following modes:                                                          *
*                                                                           *
* 1.  Welcome message - 'WELCOME TO ICTP.' is displayed for 2 seconds in this*
*                       mode.  Then it automatically enters the next mode.  *
*                                                                           *
*                                                                           *
* 2.  LED test mode   - A lit LED is rotated from right to left continuous. *
*                       LCD shows 'Rotating 1 bit. ' message.               *
*                                                                           *
*                                                                           *
* 3.  DIP switch mode - Status of DIP switch is shown on LED.               *
*                       LCD shows 'DIP switch mode.' message.               *
*                                                                           *
*                                                                           *
* 4.  ADC mode        - Analogue o/p from pontentiometer is shown on LCD.   *
*                                                                           *
* 5.  LCD test mode   - Display character set, one character at a time on   *
*                       LCD.                                                *
*                                                                           *
* 6.  Counter mode    - Pulse accumulator is tested by pressing PAI (B1)    *
*                     - button.                                             *
*                     - Counter value is shown on LCD.  Counter continues   *
*                     - to count even in other modes.                       *
*                                                                           *
* Press STRA (B2) button to enter the next mode, except for mode 1.         *
*                                                                           *
* LED lamp #10 (rightmost) shows 5V status.                                 *
*                                                                           *
* LED lamp #9 (2nd from right) shows LCD/SW mode.  Lit for LCD and off      *
* for switches.  The LCD/SW mode is controlled by STRB (with JP2 closed,    *
* JP1 open) or manually using JP1 (with JP2 open).                          *
*                                                                           *
* This program uses sequential flow in the main loop to switch mode.  No    *
* interrupts.                                                               *
*                                                                           *
*****************************************************************************
**********************************
*   Define Register Addresses    *
**********************************
```

```
RAMBS          EQU    $0000       start of ram
REGBS          EQU    $1000       start of registers
EEPROMBS       EQU    $F800       start of eeprom

***  Registers will be addressed in Ind,X mode  ***
PIOC           EQU    $02         parallel i/o ctrl reg
PORTA          EQU    $00         port a
PORTB          EQU    $04         port b
PORTC          EQU    $03         port c
DDRC           EQU    $07         data direction reg c
PORTCL         EQU    $05         alternate latched port C
PORTD          EQU    $08         port d
DDRD           EQU    $09         data direction reg d
PORTE          EQU    $0A         port e
TMSK2          EQU    $24         timer mask 2
TFLG2          EQU    $25         timer interrupt flag reg 2
PACTL          EQU    $26         pulse accumulator ctrl reg
PACNT          EQU    $27         pulse accumulator counter reg
BAUD           EQU    $2B         sci baud reg
SCCR1          EQU    $2C         sci control1 reg
SCCR2          EQU    $2D         sci control2 reg
SCSR           EQU    $2E         sci status reg
SCDR           EQU    $2F         sci data reg
ADCTL          EQU    $30         adc ctrl reg
ADR1           EQU    $31         adc result reg 1
ADR2           EQU    $32         adc result reg 2
ADR3           EQU    $33         adc result reg 3
ADR4           EQU    $34         adc result reg 4
OPTION         EQU    $39         option reg
COPRST         EQU    $3A         cop reset reg
PPROG          EQU    $3B         eeprom prog reg
HPRIO          EQU    $3C         highest priority reg
INIT           EQU    $3D         init reg
CONFIG         EQU    $3F         config reg
CONFIG_REG     EQU    $FF         EEPROM at $F800-$FFFF, cop disable

***  User Defined Constants  ***
ETX            EQU    $03         End of text
bit0           EQU    $01         define bit positions
bit1           EQU    $02
bit2           EQU    $04
bit3           EQU    $08
bit4           EQU    $10
bit5           EQU    $20
bit6           EQU    $40
bit7           EQU    $80
t8             EQU    bit6        T8 of SCCR1
rdrf           EQU    bit5        RDRF of SCSR
tdre           EQU    bit7        TDRE of SCSR
```

```
              PAGE
*********************
*  DEFINE I/O PINS  *
*********************
*PORTB :
e             EQU   bit0         control E of LCD
rw            EQU   bit1         control R/W of LCD
rs            EQU   bit2         control RS of LCD
              PAGE
********************
* DEFINE VARIABLES *
********************
              ORG   RAMBS
CHAR_CODE     RMB   1            character code for LCD
A_REG         RMB   1            tmp storage
CC_REG        RMB   1            tmp storage
MSG_PTR       RMB   2            message pointer
LCD_PTR       RMB   2            LCD pointer
BCD_BUF       RMB   3            00 00 00  -  99 99 99
MSG_BUF       RMB   17           16 character + ETX
              PAGE
***************************
* DEFINE CONFIG REGISTER  *
***************************
              ORG   CONFIG+REGBS
              FCB   CONFIG_REG
              PAGE
****************************************************************************
* BOOTSTRAP - Decide which test to perform                                 *
****************************************************************************
              ORG   EEPROMBS
BOOTSTRAP     EQU   *
              LDS   #$FF                     init stack
              JSR   PWR_UP_INIT              initialisation
TEST_LOOP     EQU   *
              LDY   #MSG_1                   load message 1
              STY   MSG_PTR
              JSR   DPLY_MSG
              LDX   #2000                    delay 2 seconds
              JSR   DELAY_IN_MS
              JSR   CLR_STAF                 clear unintended STRA
              LDY   #MSG_3                   load message 3
              STY   MSG_PTR
              JSR   DPLY_MSG
              JSR   LED_TEST                 rotate 1 bit in LED
              LDY   #MSG_4                   load message 4
              STY   MSG_PTR
              JSR   DPLY_MSG
              JSR   DIP_SW                   testing DIP switches
```

```
                JSR   ADC_TEST                    read and display ADC
                JSR   LCD_TEST                    cycle character pattern
                JSR   PA_TEST                     pulse accumulator test
                BRA   TEST_LOOP
                PAGE
*****************************************************************************
*  Messages                                                                 *
*****************************************************************************
MSG_1           FCC   'WELCOME TO ICTP.'
                FCB   ETX
MSG_2           FCC   '                '
                FCB   ETX
MSG_3           FCC   'Rotating 1 bit. '
                FCB   ETX
MSG_4           FCC   'DIP switch mode.'
                FCB   ETX
MSG_ADC         FCC   'ADC: 0.00 Volts '
                FCB   ETX
MSG_PA          FCC   'COUNTER(B1):    '
                FCB   ETX

*****************************************************************************
* LCD_MODE - Turn STRB high for LCD access                                  *
*****************************************************************************
LCD_MODE        EQU   *
                LDX   #REGBS
                PSHA              save A
                LDAA  #%00010100  full-input handshake, STRB high
                STAA  PIOC,X      write to ctrl reg
                PULA
                RTS

*****************************************************************************
* SW_MODE - Turn STRB low for switch/LED access                             *
*****************************************************************************
SW_MODE         EQU   *
                LDX   #REGBS
                PSHA              save A
                LDAA  #%00010101  full-input handshake, STRB low
                STAA  PIOC,X      write to ctrl reg
                PULA
                RTS
*****************************************************************************
* PWR_UP_INIT - Initialize control registers, I/O and RAM.                  *
*****************************************************************************
PWR_UP_INIT     EQU   *
                LDX   #REGBS
                CLR   PORTB,X                     led's off, lcd disabled
                CLR   DDRC,X                      port c as input
                LDAA  #%01000000                    set pulse accu. ctrl reg
                STAA  PACTL,X
                CLR   PACNT,X                       clear pulse counter
                JSR   INIT_LCD                      init lcd
                LDAA  #$20                          set space character
                STAA  CHAR_CODE                     for LCD test mode
                CLI
                RTS
                PAGE
*****************************************************************************
* INIT_LCD - Initialise LCD                                                 *
*          - Refer to Samsung KS0066 LCD controller data sheet              *
*****************************************************************************
INIT_LCD        EQU   *
                JSR   LCD_MODE                      turn STRB high for LCD
                LDX   #50                           wait for 50 ms
                JSR   DELAY_IN_MS
                CLC                                 select instruction reg
                LDAA  #%00111000                    set 8-bit function
                JSR   WRT_TO_LCD

                LDX   #5                            wait for 5 ms
                JSR   DELAY_IN_MS
                CLC                                 select instruction reg
                LDAA  #%001110000                   set 8-bit function
                JSR   WRT_TO_LCD

                LDX   #1                            wait for 1 ms
                JSR   DELAY_IN_MS
                CLC                                 select instruction reg
                LDAA  #%00111000                    set 8-bit function
                JSR   WRT_TO_LCD                    above sequence recommended
                                                    for init by supplier
*
                CLC                                 select instruction reg
                LDAA  #%00111000                    set 8-bit interface
                JSR   WRT_TO_LCD                    2 line LCD, 5x7 dots

                CLC                                 select instruction reg
                LDAA  #%00001000                    display off
                JSR   WRT_TO_LCD

                CLC                                 select instruction reg
                LDAA  #1                            display clear
                JSR   WRT_TO_LCD

                CLC                                 select instruction reg
                LDAA  #%00000110                    set entry mode, cursor ->,
                JSR   WRT_TO_LCD                    display not shifted
                CLC                           select instruction reg
                LDAA  #%00001100              display on, cursor off,
                JSR   WRT_TO_LCD              blink off

                CLC                           select instruction reg
                LDAA  #%10000000              set display data RAM addr
                JSR   WRT_TO_LCD              to 0
                RTS
                PAGE

*****************************************************************************
* DELAY_IN_MS - On entry, X=Delay duration in ms                            *
*****************************************************************************
DELAY_IN_MS     EQU   *

* Loop1 delay = 286x7x0.5 us = 1 ms
LOOP1           LDY   #287
LOOP2           DEY                           4 cycles
                BNE   LOOP2                   3 cycles
                DEX
                BNE   LOOP1
                RTS
                PAGE
*****************************************************************************
* ADC_TEST   - Test ADC                                                     *
*            - Read ADC and display hex value in LCD                        *
*****************************************************************************
ADC_TEST        EQU   *
                LDX   #MSG_ADC                get ROM message ptr
                LDY   #MSG_BUF                get RAM message buffer ptr
                JSR   COPY_MSG                copy
                LDY   #MSG_BUF                point to message buffer
                STY   MSG_PTR
DO_ADC          LDX   #REGBS
                LDAA  #%10000000              ADPU=1 for ADC
                STAA  OPTION,X
                LDAA  #%00110000              continuous adc
                STAA  ADCTL,X                 set ADC ctrl reg
TST_EOC         BRSET ADCTL,X bit7 DPLY_ADC   finished conversion
                JSR   CHK_STRA                check if STRA is pressed?
                BCS   XADC_TEST               yes, get out
                BRA   TST_EOC                 wait for end-of-conversion
DPLY_ADC        LDX   #REGBS
                CLRA                          clear high byte first
                LDAB  ADR4,X                  read adc result
                ASLB                          x2 to get ~5V full scale
                BCC   NO_C                    no carry
                LDAA  #1                      otherwise, set high byte
NO_C            JSR   BIN_BCD                 convert to BCD
```

```
                LDX    #MSG_BUF
                LDAA   BCD_BUF+2                    load ls digit
                ANDA   #$0F                         mask high nibble
                ORA    #$30                         convert to ASCII
                STAA   8,X                          put it at the right place
                LDAA   BCD_BUF+2                    load ls digit
                LSRA                                put it at the right place
                LSRA
                LSRA
                LSRA
                ORA    #$30                         convert to ASCII
                STAA   7,X                          put it at the right place
                LDAA   BCD_BUF+1                    load ls digit
                ANDA   #$0F                         mask high nibble
                ORA    #$30                         convert to ASCII
                STAA   5,X                          put it at the right place

                LDY    #MSG_BUF                     point to message buffer
                STY    MSG_PTR
                JSR    DPLY_MSG                     display it
                BRA    DO_ADC
XADC_TEST       RTS
                PAGE

*****************************************************************************
* COPY_MSG - Copy message from ROM to RAM buffer                            *
*          - X=source, Y=destination                                        *
*            terminated by ETX in string                                    *
*            input string cannot have ETX as text                           *
*****************************************************************************
COPY_MSG        EQU *
NEXT_BYTE       LDAA   0,X                          transfer loop starts
                INX                                 copy MSG_ADC to MSG_BUF
                STAA   0,Y
                INY
                CMPA   #ETX                         last byte
                BNE    NEXT_BYTE                    transfer loop ends
                RTS

*****************************************************************************
* BIN_BCD  - Binary to BCD conversion                                       *
*          - Input value in D, conversion in BCD_BUF, BCD_BUF+1, BCD_BUF+2  *
*****************************************************************************
BIN_BCD         EQU    *
                CLR    BCD_BUF                      clear BCD buffers
                CLR    BCD_BUF+1
                CLR    BCD_BUF+2
                STAA   A_REG                        save high byte
TST_D           LDAA   A_REG                        recover high byte
```

```
                SUBD  #0                            dummy to set Z flag
                BEQ   XBIN_BCD                      0, get out
                SUBD  #1
                STAA  A_REG                         save A
                LDAA  BCD_BUF+2                     increment lsb
                ADDA  #1
                DAA
                STAA  BCD_BUF+2
                BCC   TST_D
                LDAA  BCD_BUF+1                     increment next byte
                ADDA  #1
                DAA
                STAA  BCD_BUF+1
                BCC   TST_D
                LDAA  BCD_BUF                       increment msb
                ADDA  #1
                DAA
                STAA  BCD_BUF
                BCC   TST_D
XBIN_BCD        RTS
                PAGE

*******************************************************************************
* PA_TEST - Test Pulse Accumulator                                            *
*         - LCD shows number of times B1 (PAI) is pressed                     *
*******************************************************************************
PA_TEST         EQU   *
                LDX   #MSG_PA                       get ROM message pointer
                LDY   #MSG_BUF                      get RAM message buffer ptr
                JSR   COPY_MSG
                LDX   #REGBS
                CLRA
                LDAB  PACNT,X                       get count
                JSR   BIN_BCD                       convert to BCD
                LDX   #MSG_BUF
                LDAA  BCD_BUF+2                     load ls digit
                ANDA  #$0F                          mask high nibble
                ORA   #$30                          convert to ASCII
                STAA  15,X                          put it at the right place
                LDAA  BCD_BUF+2                     load ls digit
                LSRA                                put it at the right place
                LSRA
                LSRA
                LSRA
                ORA   #$30                          convert to ASCII
                STAA  14,X                          put it at the right place
                LDAA  BCD_BUF+1                     load ls digit
                ANDA  #$0F                          mask high nibble
                ORA   #$30                          convert to ASCII
                STAA  13,X                         put it at the right place
                LDY   #MSG_BUF                     point to message buffer
                STY   MSG_PTR
                JSR   DPLY_MSG                     display it
                JSR   CHK_STRA                     change mode?
                BCC   PA_TEST
                RTS
                PAGE

************************************************************************
*                                                                      *
* DIP_SW  - Test DIP switches                                          *
*         - Port C reads a 8-waw DIP switches and port B drives a LED array *
************************************************************************
DIP_SW          EQU   *                            set switches/led mode
                JSR   SW_MODE
                LDX   #REGBS                       read DIP sw status & disp
                LDAA  PORTC,X                      on LED array: 1=on, 0=off
                COMA
                STAA  PORTB,X
                JSR   CHK_STRA
                BCC   DIP_SW
                RTS
                PAGE

************************************************************************
*                                                                      *
* LED_TEST - Test LED at port B by cycling 1 bit                       *
************************************************************************
LED_TEST        EQU   *                            turn STRB low for switches
                JSR   SW_MODE
                CLC                                set bit 7 to 1
                LDAA  #%10000000
ROTATE          LDX   #REGBS                       display in port B LED
                STAA  PORTB,X                      save A register
                STAA  A_REG
                TPA                                save CC register
                STAA  CC_REG
                JSR   CHK_STRA
                BCS   XLED_TEST                    restore CC register
                LDAA  CC_REG
                TAP                                restore A register
                LDAA  A_REG                        delay 100 ms
                LDX   #100
                JSR   DELAY_IN_MS                  rotate bit pattern left
                RORA
                BRA   ROTATE
XLED_TEST       LDX   #REGBS                       clear all LEDs
                CLR   PORTB,X
                RTS
                PAGE
```

```
*************************************************************************
* DPLY_MSG - Diplay message on LCD
*          - Port C drives DB0-DB7 of LCD array and PB0-PB2 drive the control*
*            lines of LCD display.                                         *
*************************************************************************
DPLY_MSG       EQU   *
               JSR   LCD_MODE                  select LCD mode by STRB=1
               CLC                             select instruction reg
               LDAA  #%10000000                set display data addr
               JSR   WRT_TO_LCD                write to LCD
               CLC                             select instruction reg
               LDAA  #%00001100                turn display on
               JSR   WRT_TO_LCD                write to LCD

               LDY   #LCD_DD_RAM_ADR
               STY   LCD_PTR                   get pointer to display data
                                               RAM
NX_CHAR        LDY   MSG_PTR
               LDAA  0,Y                       get pointer to message
               INY                             get 1 byte of message
               STY   MSG_PTR                   move pointer to next byte
               CMPA  #ETX
               BEQ   END_OF_MSG                get out if ETX is met

               PSHA
                                               save it for later
               LDY   LCD_PTR
               LDAA  0,Y                       get DD RAM addr, which is
               INY                             disjoint between the 1st 8
               STY   LCD_PTR                   and the last 8

               ORAA  #%10000000
               CLC                             form display data addr
               JSR   WRT_TO_LCD                select instruction reg
                                               set display data addr
               SEC
               PULA                            select data reg
               JSR   WRT_TO_LCD                get the byte to write
               BRA   NX_CHAR                   write it to LCD
END_OF_MSG     RTS                             process next byte

WAIT_LCD_RDY   EQU   *
*Wait until LCD status indicates ready

               CLC
               JSR   READ_LCD
               TSTA
               BMI   WAIT_LCD_RDY
```

```
                RTS

LCD_DD_RAM_ADR FCB   0,1,2,3,4,5,6,7
               FCB   $40,$41,$42,$43,$44,$45,$46,$47
               PAGE

***************************************************************************
* LCD_TEST - Perform character set test on LCD                            *
*          - All characters are shown, one at a time.                     *
***************************************************************************
LCD_TEST        EQU   *
START_LINE      LDAA  CHAR_CODE                     fetch character code
                LDAB  #16                           16 characters to write
                LDX   #MSG_BUF                      points to message buffer
NEXT_FILL       STAA  0,X                           put it in buffer
                INX                                 prepare for next byte
                DECB                                count down
                BNE   NEXT_FILL                     next character
                LDAA  #ETX                          terminator
                STAA  0,X
                LDY   #MSG_BUF                      point to message buffer
                STY   MSG_PTR
                JSR   DPLY_MSG                      display it
                LDX   #500                          .5 second per pattern
                JSR   DELAY_IN_MS
                JSR   CHK_STRA                      change mode?
                BCS   XLCD_TEST                     get out if yes
                LDAA  CHAR_CODE                     recover code
                INCA                                next pattern
                CMPA  #$80                          skip blanks ($80-$9F)
                BEQ   SKIP_80
                CMPA  #0                            skip blanks ($00-$1F)
                BEQ   SKIP_20
                BRA   CONT_LINE
SKIP_20         LDAA  #$20
                BRA   CONT_LINE
SKIP_80         LDAA  #$A0
CONT_LINE       STAA  CHAR_CODE                     save code
                BRA   START_LINE                    repeat first character
XLCD_TEST       RTS

***************************************************************************
* WRT_TO_LCD - Write a byte to LCD panel                                  *
*            - A=data to write to LCD register                            *
*              Carry(C)=LCD register select (RS)                          *
*                     C=1=RS selects data register                        *
*                     C=0=RS selects instruction register                 *
***************************************************************************
WRT_TO_LCD      EQU   *
```

```
                LDX   #REGBS
*Set RS
                BCC   SET_RS_LOW1
                BSET  PORTB,X rs
                BRA   SET_RW1
SET_RS_LOW1     BCLR  PORTB,X rs
SET_RW1         BCLR  PORTB,X rw                   set write mode
                BSET  PORTB,X e                    enable LCD
*Set data
                LDAB  #$FF
                STAB  DDRC,X                       set port c as output
                STAA  PORTC,X                      write byte to LCD
                BCLR  PORTB,X e                    disable LCD
                BSET  PORTB,X rw                   set back to read mode
                CLR   DDRC,X                       set port c as input again
                LDX   #2                           delay 2 ms
                JSR   DELAY_IN_MS
                RTS

READ_LCD        EQU   *
* On entry, Carry=LCD register select (RS)
* On exit, A=data read from LCD register

                LDX   #REGBS
*Set RS.
                BCC   SET_RS_LOW2
                BSET  PORTB,X rs
                BRA   SET_RW2
SET_RS_LOW2     BCLR  PORTB,X rs
SET_RW2         BSET  PORTB,X rw                   set read mode
                BSET  PORTB,X e                    enable LCD
                LDAA  PORTC,X                      read LCD register
                BCLR  PORTB,X e                    disable LCD
                LDS   #2                           delay 2 ms
                JSR   DELAY_IN_MS
                RTS
                PAGE

*****************************************************************************
* CHK_STRA      - Check for STRA transition                                 *
*               - On exit, C-flag=0, if no active transition of STRA        *
*                                =1, if there is active transition          *
*               - On exit, STAF flag is cleared                             *
*****************************************************************************
CHK_STRA        EQU   *
                LDX   #REGBS
                CLC                                assume no transition
                BRCLR PIOC,X bit7 EXIT_STRA        exit if not set
```

```
DEBOUNCE        JSR CLR_STAF                      to clear STAF in PIOC
                LDX   #20                         debounce key
                JSR   DELAY_IN_MS
                LDX   #REGBS
                BRSET PIOC,X bit7 DEBOUNCE        exit if not set
                SEC                               set transition flag
EXIT_STRA       RTS

****************************************************************************
* CLR_STAF      - Clear STAF or STRA flag                                  *
*               - Also used to clear previous unintended setting           *
****************************************************************************
CLR_STAF        EQU   *
                LDX   #REGBS
                LDAA PIOC,X                       to clear STAF in PIOC
                LDAA PORTCL,X                     need this as well
                RTS
                PAGE

****************************************************************************
* JRTI - Return from interrupt.                                            *
****************************************************************************
JRTI            RTI
                PAGE
***************
*   VECTORS   *
***************
                ORG   EEPROMBS+$07D6
VSCI            FDB   JRTI
VSPI            FDB   JRTI
VPAIE           FDB   JRTI
VPAO            FDB   JRTI
VTOF            FDB   JRTI
VTOC5           FDB   JRTI
VTOC4           FDB   JRTI
VTOC3           FDB   JRTI
VTOC2           FDB   JRTI
VTOC1           FDB   JRTI
VTIC3           FDB   JRTI
VTIC2           FDB   JRTI
VTIC1           FDB   JRTI
VRTI            FDB   JRTI
VIRQ            FDB   JRTI
VXIRQ           FDB   JRTI
VSWI            FDB   JRTI
VILLOP          FDB   BOOTSTRAP
VCOP            FDB   BOOTSTRAP
VCLM            FDB   BOOTSTRAP
VRST            FDB   BOOTSTRAP
```

Review of College Instrumentation

Sixth College on Microprocessor-based Real-time Systems in Physics

Abdus Salam ICTP, Trieste, Italy. October 9 — November 3, 2000

A.J. Wetherilt
Artesis A.S.
Tuzla
Istanbul
Turkey

email:jim@arcelik.com.tr

Abstract

Hardware developed for the Colleges on Microprocessor—based Real—time systems in Physics is reviewed. An embedded system based around an **MC 6809** microprocessor is introduced together with a real-time, multi-threading kernel developed to run on the board. The kernel is designed to implement a small memory manager, a task scheduler,software system calls and installable device drivers. On top of the system, several layers of software are implemented, that provide full high level language library support including a version of the Posix 1003.1c (PThreads) standard. Examples are provided that illustrate the use of these libraries together with methods for compilation and debugging of C code.

1 Introduction

Since the start of the series of Colleges on Real–Time Systems in Physics, several pieces of small but useful hardware items have been developed by members of the instruction staff with the aim of furthering the effectiveness of the material presented in the course lectures. Several such items are discussed in these notes from both their hardware, and where appropriate, their software aspects. It is important to realise that although developed primarily for teaching the principles of real time systems using personal computers and embedded systems, several pieces of the hardware can be used for a much wider class of applications than found in the teaching laboratory. Cards similar in design to the MC6809 board described here have been used by the author for many data acquisition applications such as temperature control, transient digitisers and intelligent signal averagers. When equipped with the IEEE 488 instrumentation interface, the *de facto* standard for small laboratories, such instrumentation can perform significantly better than many commercially obtainable pieces of equipment and at prices at least an order of magnitude lower.

2 Hardware

2.1 The GPI board

The General Purpose Interface card (GPI) was designed by Manuel Gonçalves in 1994 to provide a means of interfacing digital signals to the then recently introduced PC systems running the Linux operating system. It is based around a single Intel 8255 I/O chip with only three other chips to provide address and I/O decoding and hence provides an extremely simple example of the principles of PC interfacing. A schematic of the board is shown in Figure 1, page 224. The 24 input/output lines from the IC can be programmed in two groups of 8 and two groups of 4 as either input or output. Two of these lines (PC0 and PC3) can provide interrupt capability when the jumpers JP1 or JP2 together with either JP6 or JP7 are selected. The interrupts selected in this case are either IRQ5 or IRQ7 which are often free in many systems.

The data lines are buffered by a 74LS245 tri-state driver to reduce loading of the PC bus. This is generally necessary in PC interface designs as the bus can drive a maximum of around 2 LSTTL loads per card. Address decoding on address lines A3-A9 is achieved via a 74LS682 digital comparator with internal pull-up resistors and an 8 way DIP switch tied to ground. The AEN line of the PC bus is also checked by the comparator to be low in order to

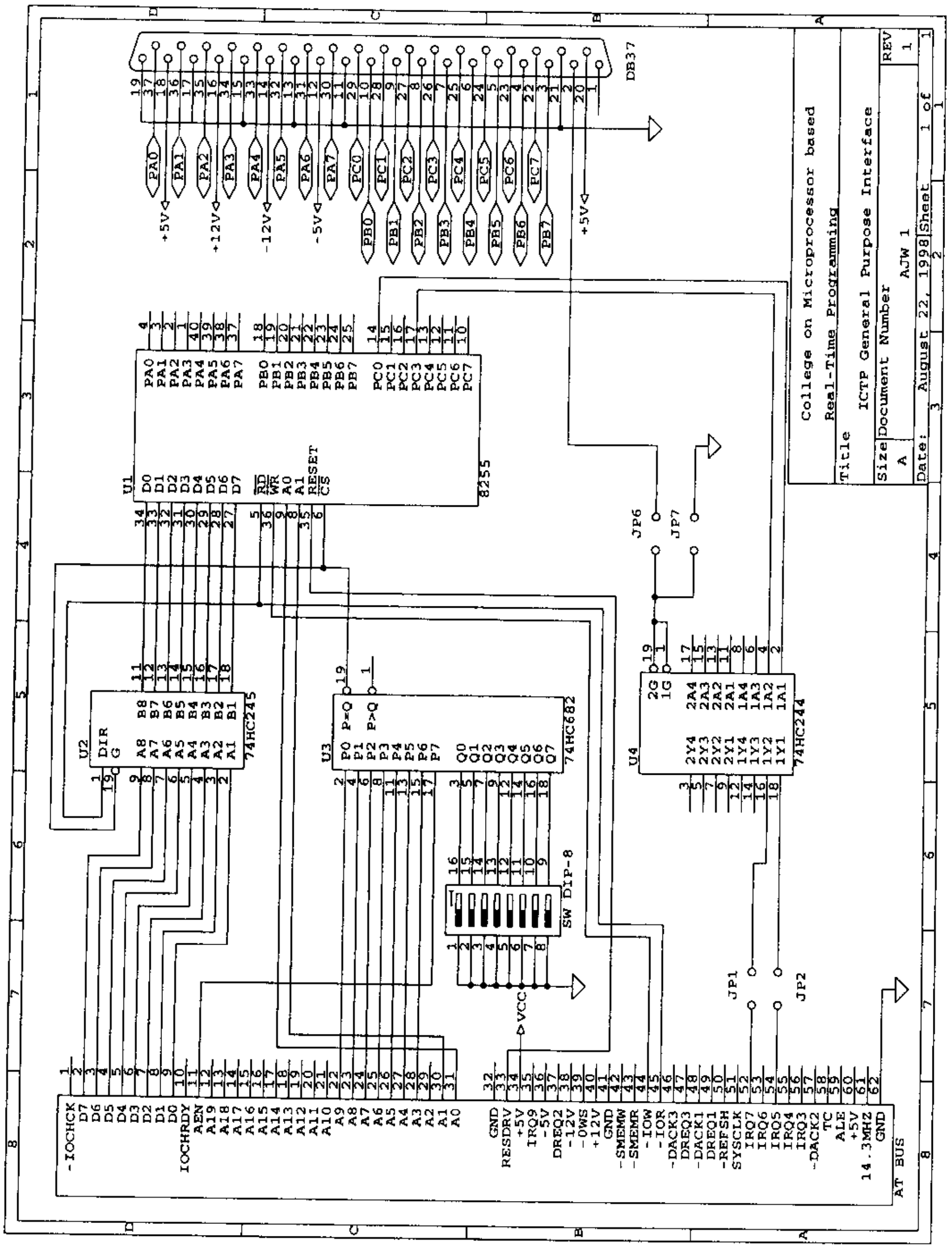


Figure 1: Schematic Drawing of the GPI board

prevent spurious access during DMA cycles (when AEN is high). As the card is to sit in the IO address space of the PC, the IOR and IOW lines are decoded directly by the 8255 programmable parallel interface. The default address is 0x320 corresponding to the locations originally assigned by IBM to the prototype card.

2.2 The Colombo board

The Colombo board is actually a complete system with provisions made for a 6809 microprocessor, a 6821 programmable interface adapter, and RAM and ROM situated on one half of the board. The remainder of the board comprises a 4 digit, 8 segment LED display together with various switches and devices that can be interfaced via a 26 pin connector (J2) to either the on-board microprocessor or an external host machine (see Figure 2, page 226). It is this latter feature that has been used predominantly during the various colleges.

The 26 pin connector definitions are shown in Figure 4 (page 228) and are to be considered the standard connections for College instrumentation. Two sets of data lines can be seen which reflect the characteristics of the 6821 PIA around which the board was designed. The set of A lines (PA0 - PA7) are connected to the latch/drivers of the 4 LED displays and data latched in the following manner (see Figure 2 on the page 226): The hexadecimal digit to be written on a given LED is placed on lines PA4–PA7. The data is latched by first setting the E pin of the specified digit low and then resetting it back high again. As each digit has a separate line attached to it, digits that share the same data can be latched individually. A clock that produces pulses at a selectable rate can be attached to line CA1. When connected to a suitable input on the host device an interrupt can be raised by these pulses. On some cards a buzzer is connected to line CA2, on others the buzzer has been replaced by a LED. In either case, setting CA2 high causes the attached device to function.

Connections to the B side are entirely inputs (Figure 4 on page 228): On lines PB4-PB7, a 16 position rotary switch is attached; on PB3 and PB2, are two toggle switches; and on PB0 and PB1 are two push button switches, connected via a 74279 for debouncing. These push buttons can also be jumpered to line CB2 which, when connected as an input on the host machine, can raise interrupts. Finally, a voltage to frequency converter is attached to line CB1. This device converts an analogue voltage signal into a sequence of pulses at a frequency determined by the magnitude of the signal. If the number of pulses arriving per unit time is counted, the magnitude of the signal can be determined.

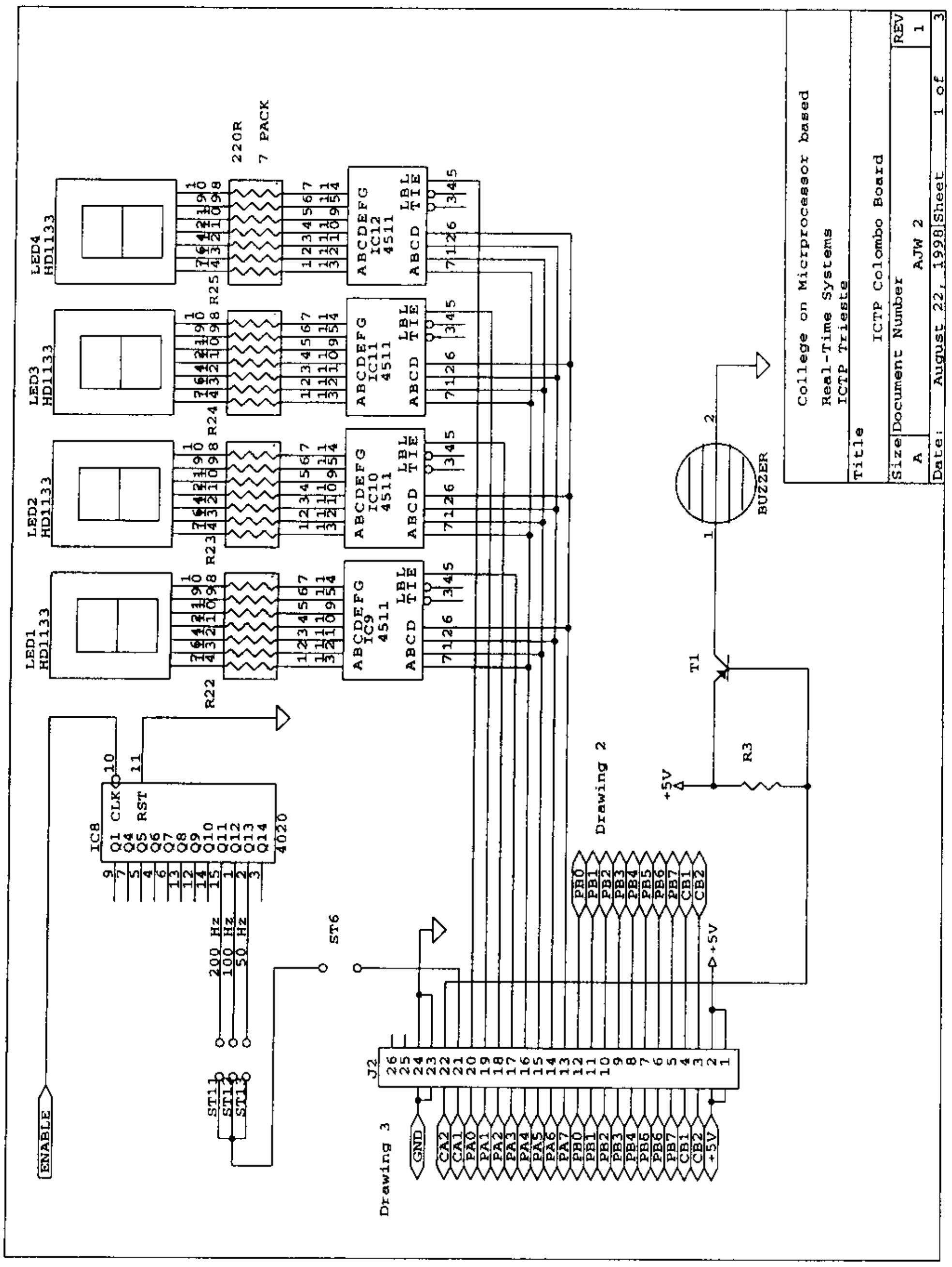


Figure 2: Schematic Drawing of the Colombo Board

+5V	**● 1**	**2 ●**	**+5V**
CB2	**● 3**	**4 ●**	**CB1**
PB7	**● 5**	**6 ●**	**PB6**
PB5	**● 7**	**8 ●**	**PB4**
PB3	**● 9**	**10 ●**	**PB2**
PB1	**● 11**	**12 ●**	**PB0**
PA7	**● 13**	**14 ●**	**PA6**
PA5	**● 15**	**16 ●**	**PA4**
PA3	**● 17**	**18 ●**	**PA2**
PA1	**● 19**	**20 ●**	**PA0**
CA1	**● 21**	**22 ●**	**CA2**
Ground	**● 23**	**24 ●**	**Ground**
Timer 3 gate	**● 25**	**26 ●**	**Timer 3 output**

Figure 3: Pin Definition of the Standard ICTP Connector

2.3 The LCD display board

Designed, by C.S. Ang, as a more up to date and modern replacement for the Colombo board described previously, this card features a 16 digit ASCII LCD display panel in addition to two push button switches, an 8 way DIP switch and an 8 LED strip (Figure 5, page 229). A number of different connectors allows several possibilities for the host machine. The first of these, is a 40 pin strip connector for direct interfacing to the 6811 card described next. The standard ICTP 26 pin connector is also found on the card allowing connections to be made to either the GPI card or the 6809 card described later. Since the latter connector has fewer pins than the former, the card functionality is also somewhat reduced when this connector is used. However, it still allows a significantly better display capability than the Colombo board. As the details of the card with reference to the 6811 interface are more than adequately covered in the notes of C.S.Ang, only those aspects relevant to the standard ICTP interface will be discussed here.

The LCD display is an Agena AA16102 module capable of displaying up to 16, 5x7 pixel alphanumeric characters. With CA2 held high, data are placed on lines PB0-PB7. The line PA1 is set low for a write operation and PA2 is set according to whether the operation is a write data (high) or a write instruction (low). The line PA0 is then strobed from low to high to low for the data to be latched. For full details of all possible operations, please refer to the manufacturer's data sheet.

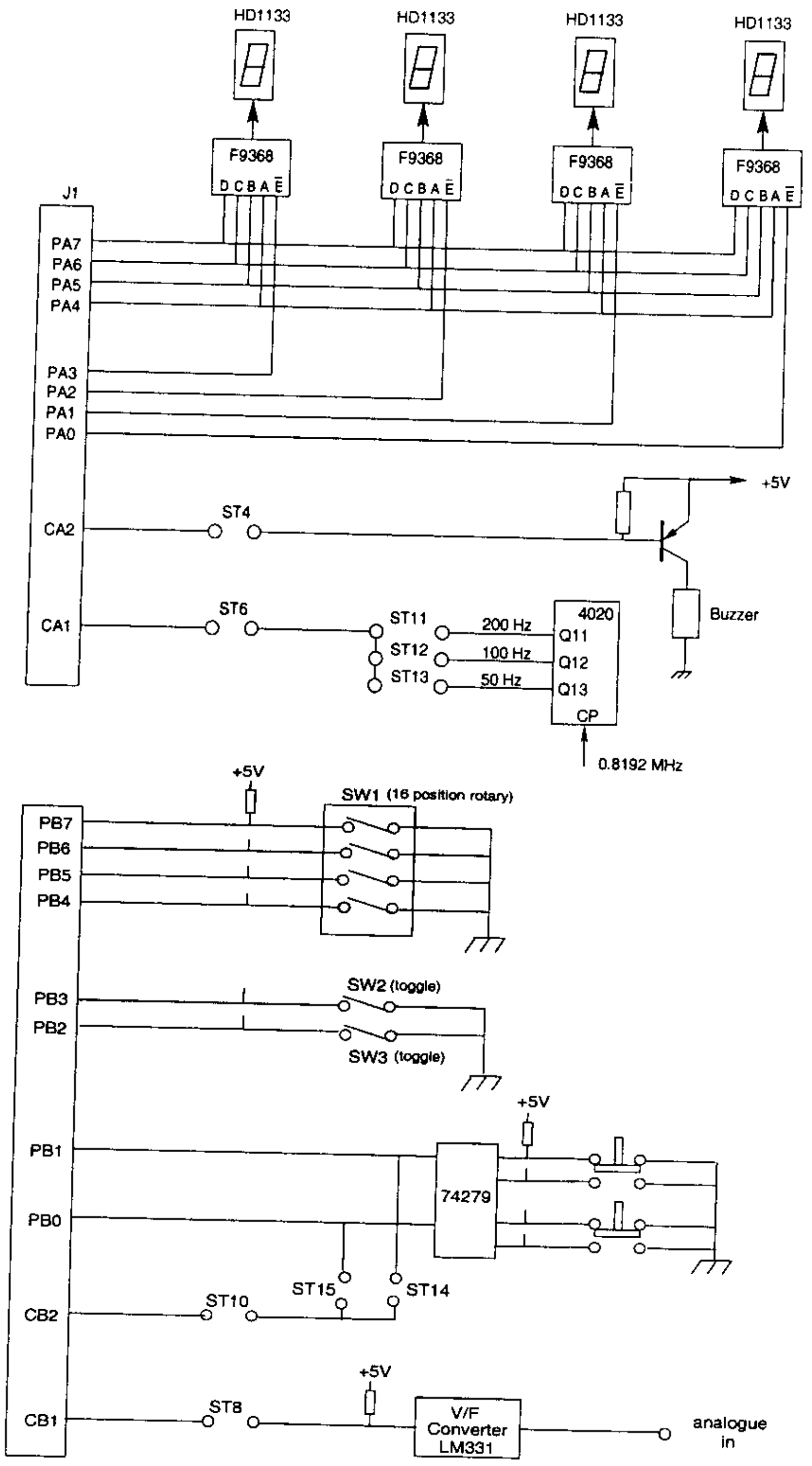


Figure 4: Simplified details of the Colombo Board showing A and B sides

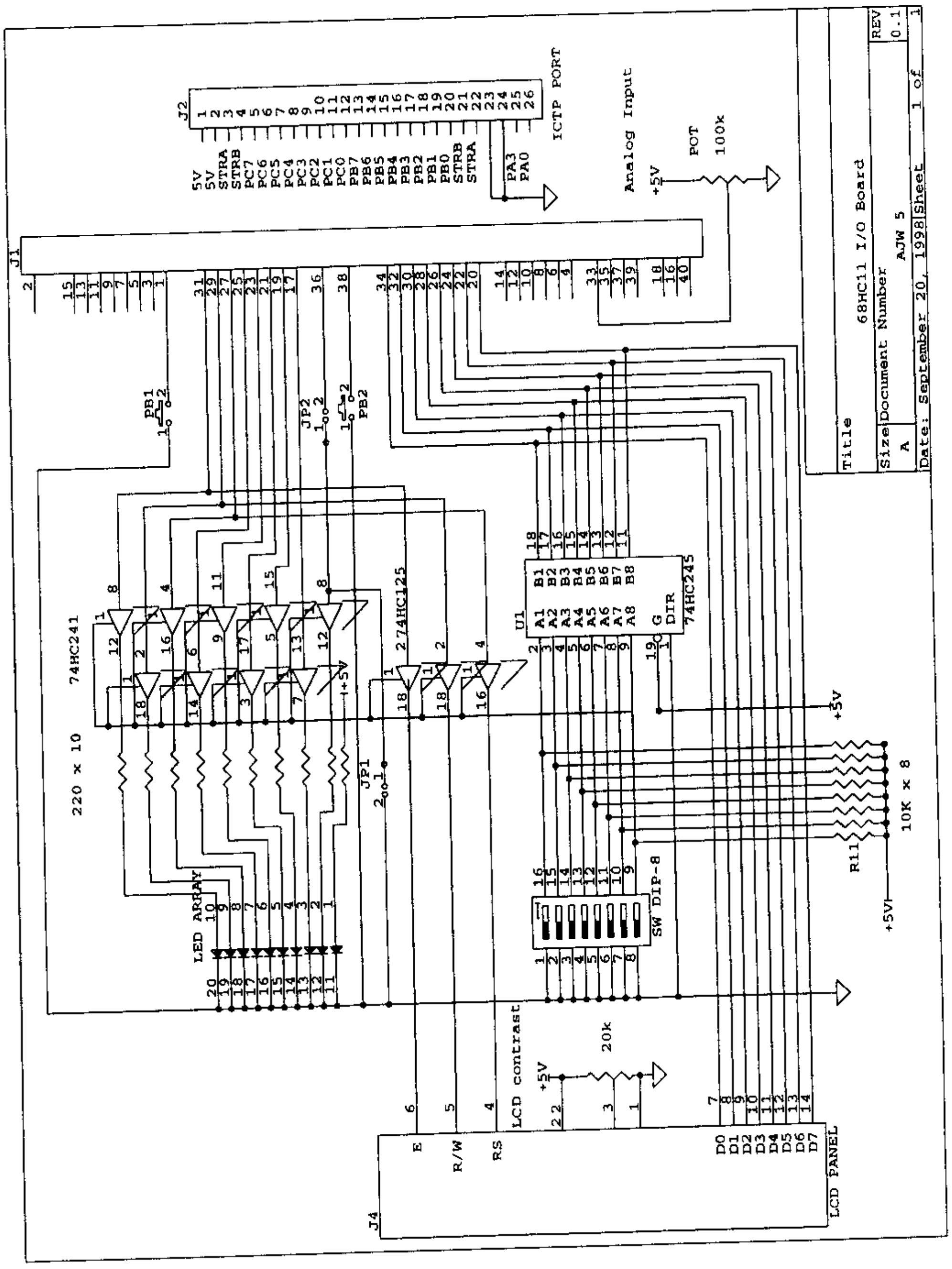


Figure 5: Schematic Drawing of the LCD Display Board

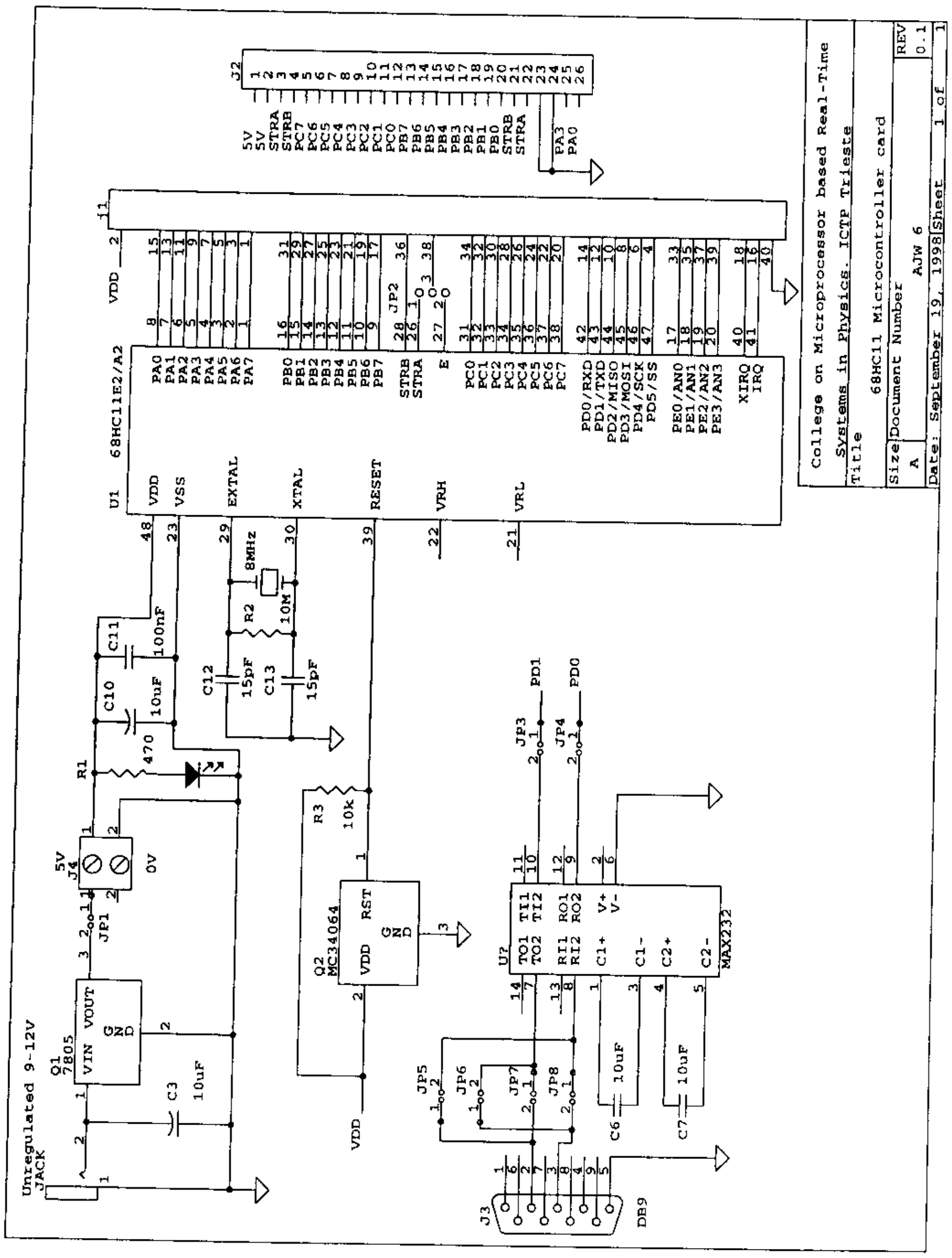


Figure 6: Schematic Drawing of the M6811 board

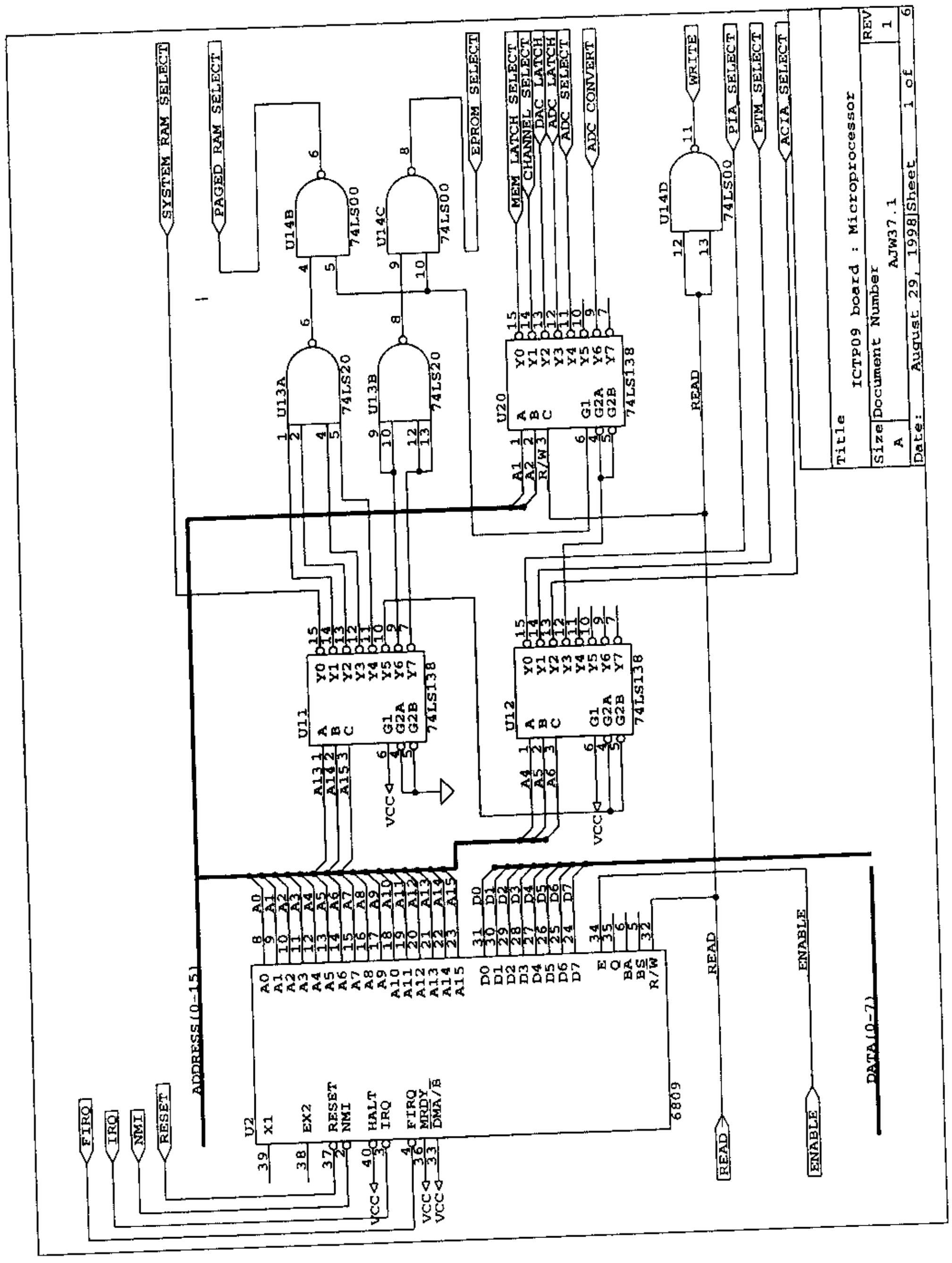


Figure 7: Schematic Drawing of the M6809 board

)

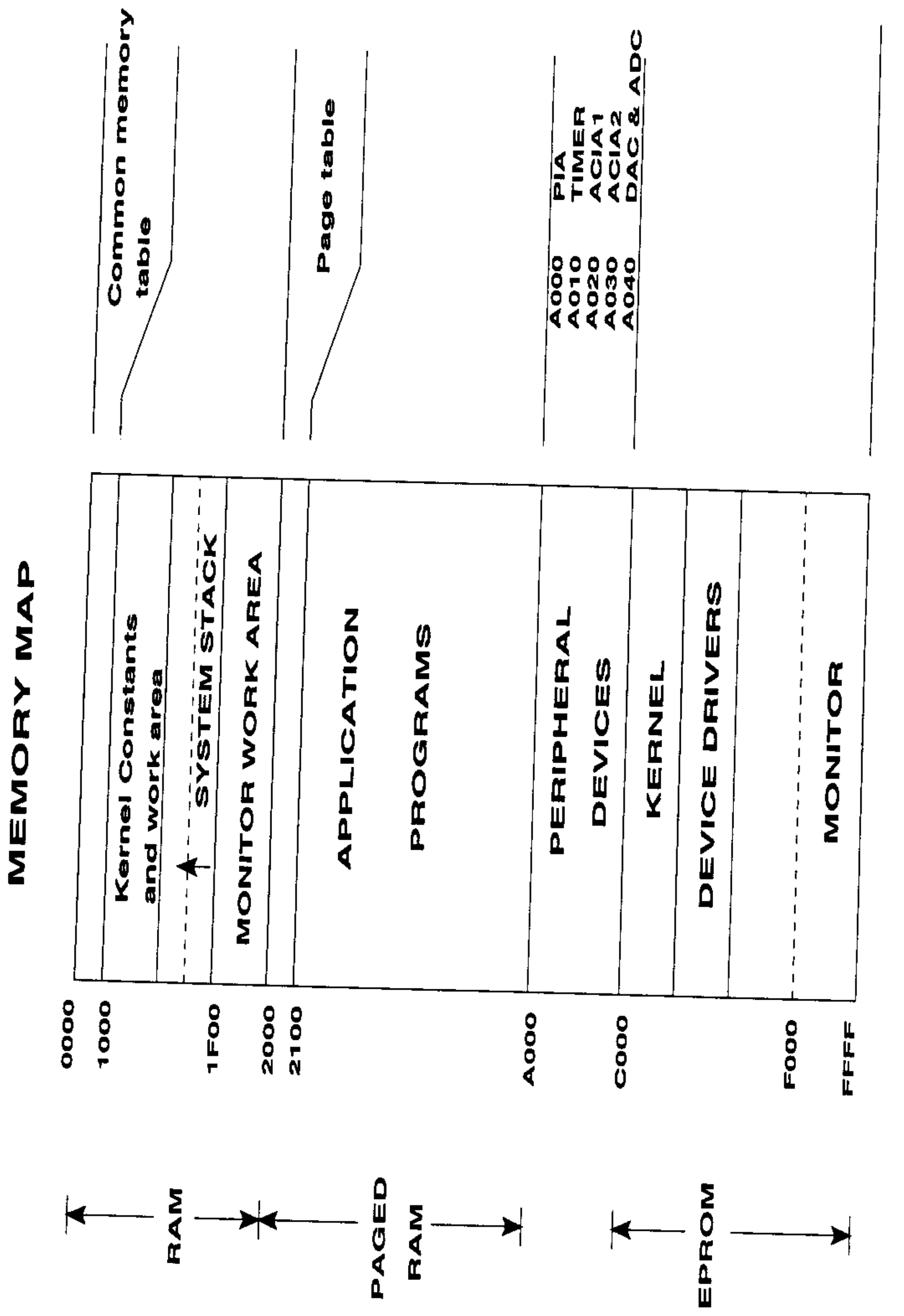


Figure 8: Memory Map of the M6809 under RInOS

system calls, the user is unaware of the details of the kernel and has only to supply a means of using the system calls from the programming language of his choice. For C running under LINUX, the GCC compiler modified to produce absolute code for the 6809 can make use of libraries encapsulating the assembler commands. Simple memory management is provided so that a process can be allocated memory as and when it is needed and return the memory to the heap when finished. Processes can be loaded and relocated by the memory manager. Alternatively, absolute code can be used as long as certain well defined steps are followed.

2.5.2 Hardware description

The board is based around a MC6809 processor running at a clock speed of 1 MHz. Although the 6809 is now an old microprocessor, its use in a piece of hardware intended mainly for teaching purposes can be justified on the grounds of its superior instruction set and clarity of use. The 6809 arguably, still has the best instruction set of any 8 bit microprocessor or micro controller and is ideally suited for the current purpose. Development tools are widely and freely available at many sites on the Internet which is a great advantage for any device.

Throughout the design stage, stress has always been laid on those areas that will allow the various aspects of microprocessor teaching to be emphasised. For this reason two identical serial communications ports have been provided. These allow communications drivers to be debugged easily using one port connected via the monitor to the host machine and the second to the hardware application. For both ports, the baud rate can be set by changing jumper JP2. If faster rates are required, the ACIAs at 0xA020 and 0xA030 (Figure 9, page 235) must be configured so that the clock is divided by 1 rather than 16 and the jumpers adjusted accordingly. Communication uses only the TxD, RxD and ground return lines of the 9 pins of the RS 232 ports. For interconnection between the board and a host PC, null modem cables must be used.

The 6840 PTM provides 3 timer channels. The first is attached to the NMI line and is used by the monitor for tracing through code, and the second is used for the system clock by the kernel. It issues a clock interrupt on the IRQ line at 10 ms intervals. The third clock is available to a user and has both gate and output on the onboard standard ICTP 26 pin strip connector. To ensure these and other interrupt signals are processed, the jumpers must be set correctly on jumper JP1. Under RInOS, all interrupts except the MON signal from timer channel 1 which is jumpered to the NMI line, must be jumpered to the IRQ line. Jumpering to the FIRQ line without special

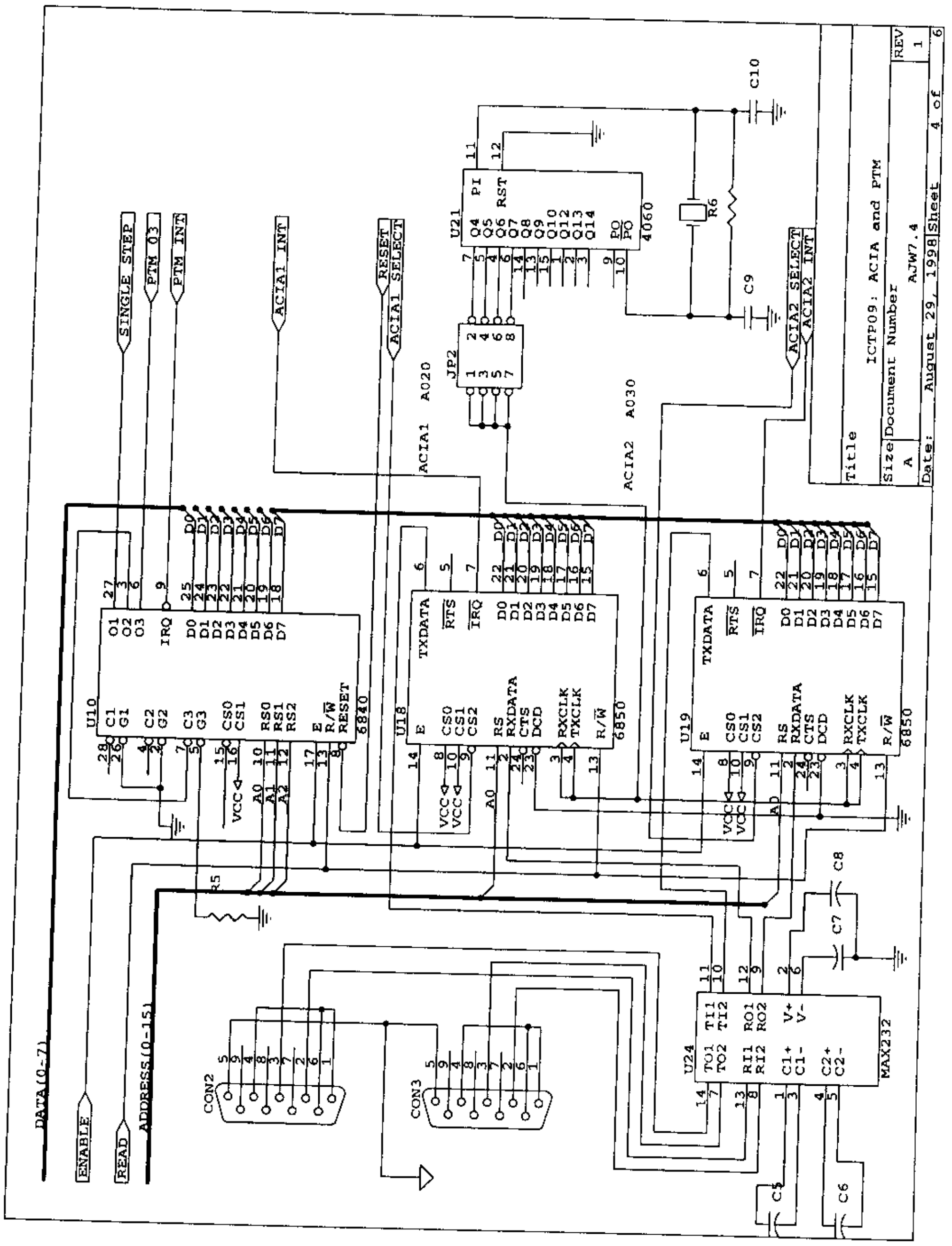


Figure 9: ACIA and PTM on the M6809 board

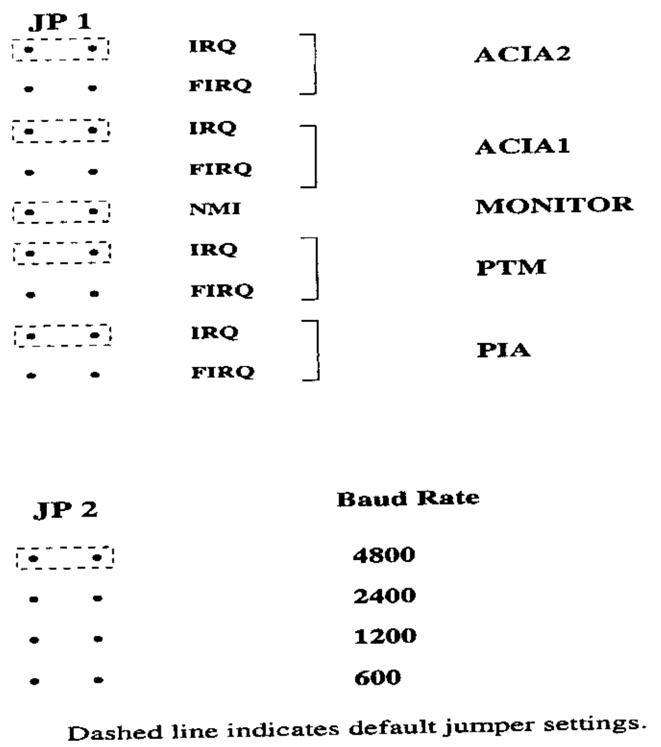


Figure 10: Jumpers Setting for the M6809 Board

provision will cause unpredictable results and generally will hang the system. Refer to Figure 10 on page 236 for a description of the jumper settings.

Random access memory is used to provide (i) a common area for system and application programme use and (ii) an area in which large processes can be loaded. These are supplied by a 2764 equivalent 8k RAM at 0x0000–0x1FFF and a 581000 128k RAM at 0x2000–0x9FFF (see Figure 11, page 237). Since the entire address space of the 6809 is only 64k, the 128k of the 581000 is divided into 4 pages each of 32k in size by decoding the upper two address lines of the 581000 with an address latch. Writing the values 0-3 to the latch will cause the appropriate page to be set. It is advised that application processes do not interfere with this register when the kernel is running.

Two channels each of ADC and DAC are provided. No interrupt capability is provided for the ADC channels as at a clock rate of one MHz, conversion takes less than approximately 25 μs, which is only barely more than the time required to handle a straight forward interrupt request. For times longer than this, timer channel 3 can be used.

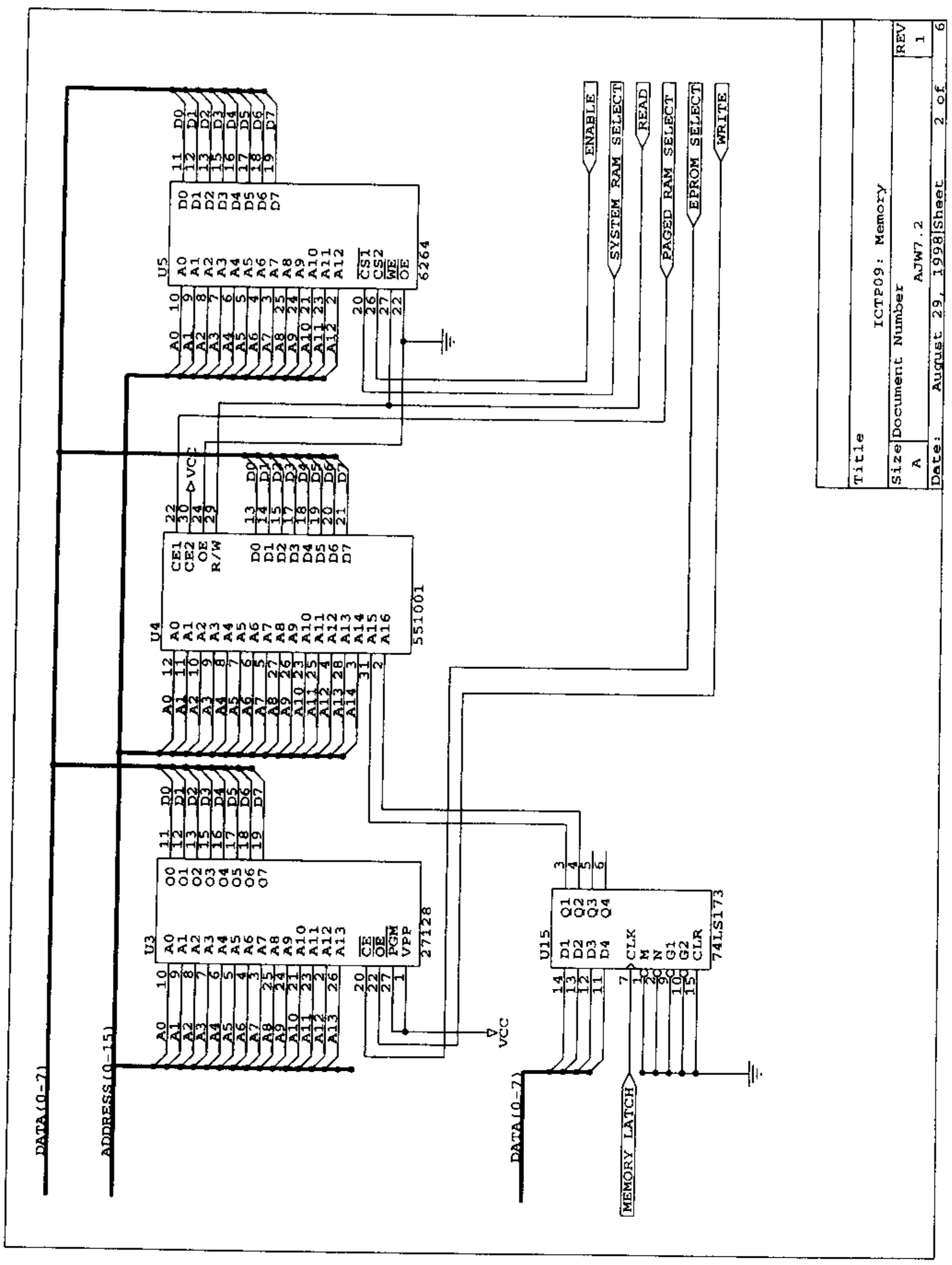


Figure 11: Memory on the M6809 Board

3 The RInOS kernel

This first version of the kernel is written entirely in 6809 assembler rather than a high level language such as C. The design criteria were that the system should :

(i) Use software interrupt system calls to an EPROM based kernel to interface to an applications programme rather than be linked in with it at compile time and subsequently downloaded to RAM.

(ii) Allow a variable number of applications to be downloaded to RAM where they could be run concurrently when the kernel was started.

(iii) Provide a set of functions that would allow the efficient coexistence of, and communications between, a number of processes.

(iv) Provide a means of installing device drivers that could be changed after the start of the kernel and without having to re-assemble the system code.

Several real-time kernels were examined for their suitability for use in an embedded system such as the 6809 board. Initially, the μ/COS real time kernel was considered as a possible candidate but was rejected because criterion (i) above was difficult to fulfil. Another alternative was to modify the MCX11 Real Time Executive provided for the 6811 by Motorola. The dispatcher of this kernel was particularly suitable for use with the 6809 and was used as a model for the RInOS scheduler. The remainder of the code was, however, written completely afresh to fulfil the various criteria.

3.1 Context switching under RInOS

At the heart of any kernel is the process scheduler or dispatcher. This function determines which task will run and for how long. RInOS uses a simple technique to determine whether the current task is to continue running or will yield to another, by assigning to each task a priority level between 1 and 255. The priority level 0 is reserved for the null task which always is in a runnable state but runs only when no other task is available. The priority level is used to determine the position where the task can be inserted into a linked list of tasks starting with the highest priority task and ending with the null task. The system variable **prioptr** at address 0x108 always points at this list. A task can change its priority by unlinking and inserting itself into its new level. During a context switch the linked list of tasks is searched until one is found that is in a runnable state. This and all other information needed by the system to describe each task is found in a structure called a task control block or TCB which is given in Table 1 on page 240. The STATUS field of this structure indicates whether or not a task is runnable or is asleep. When a runnable task is found, the following sequence of events occurs:

(i) All interrupts are switched off

(ii) The system variable **taskptr** is set to point at the new task. This always indicates the address of the current task

(iii) The system variable **intlvl** is decremented. This variable indicates the level of interrupt nesting. If after decrementing, it is zero, the system was not interrupted and it is safe to return to the application task that was either running at the time the interrupt was issued in the case of a hardware interrupt or issued the system call in the case of a software interrupt. Non zero values indicate that the system itself was interrupted and it is not safe to perform a context switch at this time. In the latter case, a simple return from interrupt is issued and control flows to the point of interruption. In the former case the sequence continues with the value saved in the STACK_POINTER field of the TCB being transferred to the stack pointer register of the processor.

(iv) A return from interrupt is now issued with the new stack pointer. This causes the machine registers to be filled with the values found on the stack. The final register to be pulled from the stack is the programme counter which causes a jump to the new value just pulled from the stack and hence a transfer of code execution to a new task.

This sequence is illustrated in Figure 12, page 241. During the issuing of a system call, the reverse process occurs:

(i) The register set is pushed onto the stack and interrupts are switched off

(ii) The variable **intlvl** is incremented

(iii) If **intlvl** was zero before being incremented, the value of the stack is saved in the STACK_POINTER field of the TCB pointed at by **taskptr** and the value of the system stack is placed in the stack pointer. Otherwise the system stack is already in use and is not therefore reset.

Table 1: Important Task Control Block structure fields

void *	PRIORITY_POINTER	Link to next highest priority task
int	PID	Unique thread handle
char	PRIORITY	Priority level
char	STATUS	Current status of the task. Possible values are: READY WAITING SLEEPING SUSPENDED NO_TASK
void*	CODE_ENTRY	Initial entry point of task
void*	CODE_START	The start of the code in memory. This is written by the memory manager after loading
void*	STACK_SEGMENT	The start of the stack reserved for the thread
void*	STACK_POINTER	Used to save the current context of a task during a system call or hardware interrupt
int	STACK_LEN	The size of the stack allocated to the thread
char	PAGE	The page of memory allocated to the task
int	PARENT	The handle of the parent thread
char	EXIT_STATUS	Termination status of the thread
char	EXIT_CODE	Return code of the thread
void*	EXITFUNC	Pointer to an optional exit function
void*	EXITFARG	Pointer to an optional parameter to be passed to the exit function
void*	MAILBOX	Pointer to the task mailbox
void*	SEMAPHORE_LIST	A link to the next task waiting on a semaphore
int	TIMER_COUNT	Used to indicate how many clock ticks a sleeping task has to wait before being woken
void*	TIMER_LIST	A pointer to the list of threads waiting on the timer
void*	ARGPTR	A pointer to the optional argument list passed to the thread
char	ERROR_STATUS	The status of the last error encountered by the thread
char	ATTRIBUTE	The thread attribute bit fields

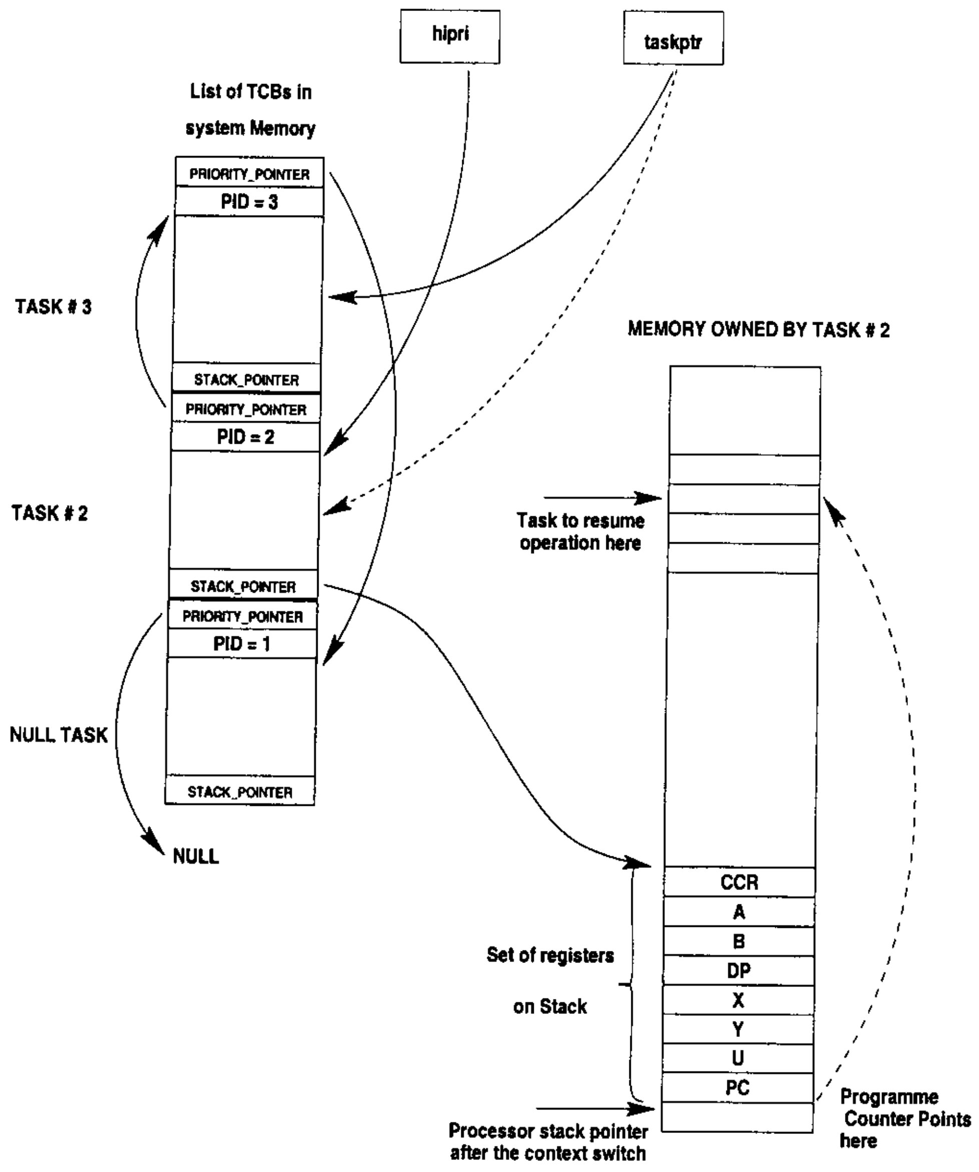


Figure 12: The steps involved in a context switch. Task #3 was running when an event occurred that made task #2 runnable. Task # 3 is thus preempted and **taskptr** is changed to point at task #2. The STACK_POINTER field of the TCB belonging the task #2 points at the set of registers pushed onto the stack when task #2 was last interrupted or preempted. These registers are pulled from the stack individually ending with the Programme Counter. When this register is finally pulled, the context switch is complete and task # 2 resumes running.

(iv) The function call number is examined and a jump is made to the particular call requested

(v) Interrupts are switched back on again. In general, interrupts are off only during sections where interruption would cause problems and switched back on again as soon as possible. Application programmes should not change the interrupt status as this could interfere with the functioning of the kernel.

When a task is first created, it is given a new TCB and a unique integer handle, stored in the PID field of the TCB. The area reserved for the task's stack is placed into the STACK_POINTER field and the various register values are initialised on the stack. The value of the CODE_ENTRY field is placed on the stack so that it will be pulled off into the programme counter during a context switch. In order to start multitasking, the kernel will simply find the highest priority task in the linked list and perform a context switch to that task.

3.2 Hardware interrupt handling and device drivers

In contrast to the handling of system calls via software interrupts which occur in an orderly and predictable manner, hardware interrupts by their very nature are asynchronous and can occur at any time. On the 6809 and many other processors, a hardware interrupt is handled by reading a location specially reserved for the interrupt and jumping to the address found in that location. The actual mechanisms may vary from processor to processor, but in general the actions are similar. On the 6809 board, the interrupt vectors are found in the EPROM at the addresses between 0xFFF0 to 0xFFFF. This means that the interrupt vectors themselves can not be changed and must always point to the same handler. To circumvent this problem, the monitor maintains a second set of interrupt vectors in RAM which can be altered at will and, during installation, the kernel writes the value of its own set of interrupt handlers to these vectors. The interrupt sequence then becomes:

(i) The processor stacks the register set in the same order as for a software interrupt;

(ii) A jump is made to the location found in the IRQ vector at address 0xFFF8;

(iii) A second indirect jump is made to the location in the monitor vector table;

Table 2: Device table structure fields

void*	INTERRUPT_SERVICE	Address of the interrupt handler for the device
void*	DEVICE_DRIVER	Address of the device driver
void*	HARDWARE	Address of the hardware
void*	DATA_AREA	Address of an area reserved for use by the device driver in which it can store information it needs
char	INSTALLED	A flag to indicate whether or not the particular device is installed. 0 indicates it is not

(iii) The kernel interrupt handler processes the interrupt.

This sequence adds about 9 cycles to the time required to service the interrupt and may be undesirable in a very time critical application. However, when demonstrating the principles of real time techniques, the versatility gained by being able to replace the interrupt handler has a number of advantages.

The RInOS kernel uses a system of device drivers to form an interface between applications programmes and the system hardware. An application programme should never manipulate the hardware directly in a multitasking environment as this could interfere with the operation of another task which also requires the use of the hardware. The kernel maintains a list of the six devices available on the board, namely: ACIA1, ACIA2, PIA, ADC, DAC and TIMER3. The system clock is treated separately. Each entry in the device table contains the structure shown in Table 2 on page 243.

The table is completed by the kernel during system initialisation with values for the default device drivers. TIMER3 at present has no default driver, therefore one must be supplied if this device is to be used by an application programme. Since the table is in RAM, it is possible to replace an entry with the parameters of a new driver. The system call **OSInstallDriver** should be used for this purpose.

Note that the particular structure of the device table allows ACIA1 and ACIA2 to share the same device driver software but to have different DATA_AREA and HARDWARE fields.

During a hardware interrupt, the device table is examined to find the device causing the interrupt. Each device capable of raising an interrupt has a status register that indicates whether or not it requires service. If a device is found to require service the service routine at the address found in the INTERRUPT_SERVICE field is called. Otherwise the next device in the list is examined. If no other devices are found to have issued an interrupt, the system clock on PTM channel 2 is examined. and appropriate action taken. If no device is found to have requested service, a serious system fault could occur if the spurious interrupt does not clear itself, as on return to the point of interruption, an uncleared interrupt will immediately reassert itself and cause an loop of interrupts that will effectively hang the system.

3.3 Memory management

Memory management under RInOS distinguishes between two types of available memory. The first 8k of memory starting at 0x0000 is available to both processes and system alike. The paged memory is only available, however, to processes on the same page; processes on separate pages cannot share data in this area but must use the common area starting at 0x0000. Separate memory allocation and deallocation system calls exist, therefore, for these two distinct regions of memory. In each case, however, RInOs uses similar constructs to handle their management. The first 256 bytes of each area (at 0x0000-0x00FF and at 0x2000-0x20FF on each page) contain a page table for the relevant block of memory. Memory is allocated in blocks of 32 bytes each for the common memory and in blocks of 128 bytes each for paged memory. After system initialisation, each unused block is marked by 0xFF and each used block contains the pid number of the task owning it or zero if the system owns it. If a block of memory is requested, the relevant table is scanned to find a vacant area of sufficient size. The first such area found is marked as belonging to the requesting task and a pointer to the start of the area returned to the caller. If paged memory is requested and no space can be found on the first page, successive pages are searched until a block is found. No hardware protection is provided to prevent one process from using the memory of another: it is expected that such antagonistic actions can be guarded against by the application programme designer.

3.4 Semaphores

Perhaps the single most important programming construct of real time programming is the semaphore. A semaphore is basically a lock that permits a given number of users to access a system resource of some description.

Table 3: Semaphore structure fields

char	SEMA_TYPE	The type of the semaphore 0 = Mutex 1 = Counting 4 = Event
char	SEMA_VALUE	The value of the semaphore
void *	SEMA_POINTER	A pointer to the first task in a linked list of tasks waiting on the semaphore

When a task claims the semaphore by performing a DOWN operation, it effectively locks out other users from the resource until the task again releases the semaphore. Alternatively, a semaphore can be used to signal that an event which one or more tasks have been waiting for has occurred. RInOS uses three types of semaphore: the binary semaphore or mutex; the counting semaphore; and the event semaphore.

The structure of the semaphore is given in Table 3 on page 245.

A mutex can have but two values: 0 or 1. A task can claim the mutex if its value is 1, which will immediately cause the value to change to 0. Any other task trying to claim the mutex will be blocked at this stage. On receiving such a request, the kernel rather than decrementing the value of the semaphore in the SEMA_VALUE field, links the task into a list of tasks already waiting on the semaphore. The first task in the list is pointed to by the field SEMA_POINTER. If this field is null, then no tasks are waiting on the semaphore yet. A mutex places the task in the list according to the priority of the task and sets the SEMAPHORE_LIST field of the TCB to point at the next task in the list. When an UP operation is performed on the mutex, it first attempts to wake the highest priority task i.e. the first task in the linked list, and to remove the task from the list. If this fails because no task is waiting, it increments the semaphore value to 1 (see Figure 13 on page 246).

Counting semaphores are implemented in a similar manner, but can take any positive value up to 255. They are typically used in situations where there is a limited number of resources such as slots in a buffer which can be allocated to users. In this case they would be initialised to the number of usable resources. A DOWN operation would decrement the semaphore value until 0 is reached at which stage the task issuing the DOWN would be

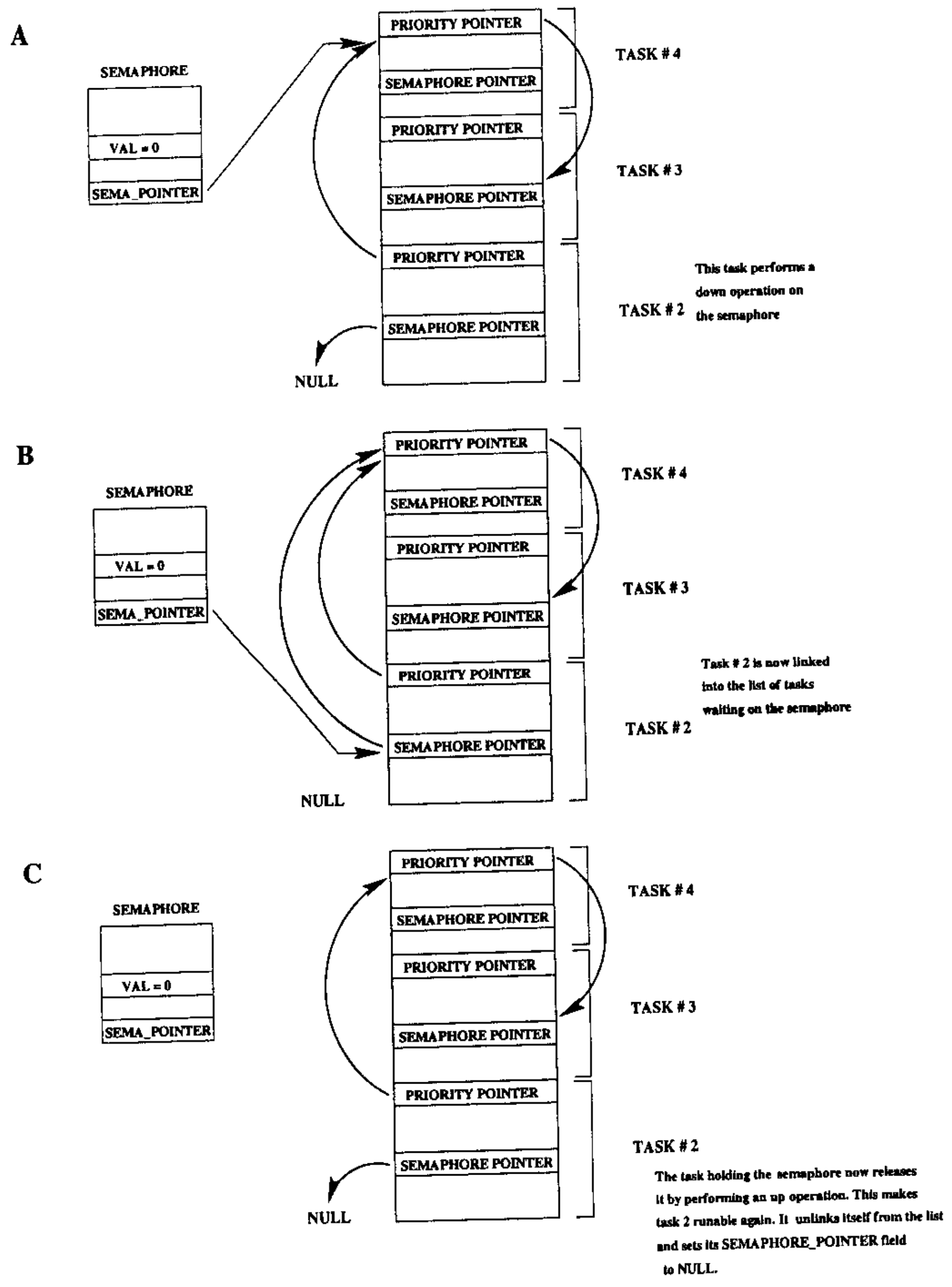

A
SEMAPHORE
VAL = 0
SEMA_POINTER
PRIORITY POINTER
SEMAPHORE POINTER
TASK # 4
TASK # 3
TASK # 2
This task performs a down operation on the semaphore
NULL
B
SEMAPHORE
VAL = 0
SEMA_POINTER
PRIORITY POINTER
SEMAPHORE POINTER
TASK # 4
TASK # 3
TASK # 2
Task # 2 is now linked into the list of tasks waiting on the semaphore
NULL
C
SEMAPHORE
VAL = 0
SEMA_POINTER
PRIORITY POINTER
SEMAPHORE POINTER
TASK # 4
TASK # 3
TASK # 2
The task holding the semaphore now releases it by performing an up operation. This makes task 2 runable again. It unlinks itself from the list and sets its SEMAPHORE_POINTER field to NULL.
NULL

Figure 13: Steps involved as a task waits on a semaphore. (A) task #2 performs a down operation and is blocked. (B) Task #2 is linked into the list of tasks waiting on the semaphore. (C) The semaphore is released allowing task #2 become unblocked and to resume running.

attached to the linked list. A difference between the implementation of mutex and counting semaphores is that whereas the mutex links in order of priority so that the highest priority task is woken first, the counting semaphore links in the strict order of arrival.

The third type of semaphores are used to synchronise events. The semaphore is always initialised to 0 and any task issuing a DOWN on it automatically is put into the linked list to sleep. When the event that the tasks have been waiting for arrives (by performing the UP operation on the semaphore) ALL tasks waiting on the semaphore are woken. This does not mean that all tasks run at the same time, but that all are put into a state where they can run when given the opportunity.

RInOS offers a number of functions to create, free, perform UP and perform DOWN on the three types of semaphore. Semaphores are implemented in two different ways. During kernel initialisation, 128 semaphores are made ready for system usage in shared memory and are thus available globally, irrespective of the page. Alternatively, since a semaphore is only a suitable initialised structure in memory, any such memory block can be used as a semaphore. User semaphores are generally created in paged memory and are thus local to a single page, this makes them suitable for a set of threads created on a single page but not for inter-process (ie between pages) usage.

3.5 Interprocess communication

When loaded, each process has absolutely no information concerning the details such as pid, semaphore numbers etc. of any other thread. A thread can examine its own details by issuing the **OSTaskInfo** system call and getting a pointer to its own TCB, but has no access to information belonging to any other thread. To circumvent this problem, a thread with priority lower than any other except the null task can, as one of its first actions, reserve a block of shared memory and place relevant information in this memory. The format of this information and its meaning must be agreed between the various processes but otherwise is arbitrary. Care must be taken with this and any other shared resource to ensure that access is regulated by guarding with a mutex or counting semaphore. RInOS implements pipes, signals and message passing as basic interprocess communication functions.

Messages are sent between threads using the **OSSend** and **OSReceive** pair of system calls. These functions send messages to and examine the contents of a task's mailbox. A mailbox is basically a linked list attached to the MAILBOX field of the task's TCB. A task sending a message sends a pointer to a block that can be used as memory shared between both the sender and the receiver. The receiver can examine its mailbox at any time and act on the contents of the message. Alternatively, if a task chooses not to look at its mailbox, the messages will go unprocessed. An optional semaphore can be set using the **OSReply** system call that puts the sender to sleep until a reply is received. This should obviously be used with a certain amount of caution.

Signals under RInOS do not invoke a signal handler and are similar to messages except that messages are sent to individual threads and signals to any thread that wishes to receive notification of an event. A thread wishing to receive a particular numbered signal would issue the **OSWaitSignal** system call specifying the desired signal number as an argument. The thread will then block until the desired signal is issued. Two types of signal are provided: The first type is persistent in that after being issued, it is always in effect until cancelled. The second type signals only those threads waiting for the signal at the time of issue. A thread starting a wait after the signal is issued will miss it unless the signal is resent. The user must decide which signal type is most appropriate for the application being designed.

A pipe is basically a queue or fifo for holding data with suitable protection in the form of counting semaphores and mutexes. Under RInOS, functions exist for creating, opening, writing to, reading from, and closing a pipe. When being opened, the user specifies the width of data (in bytes) to be sent down the pipe, ie 1 for *char*, 2 for *int* etc. A pipe will block if it is full when writing or empty when reading.

3.6 The system clock

When the PTM is jumpered to the IRQ line, the system clock provides a periodic interrupt every 10 ms. Tasks wishing to sleep for an integral number of clock ticks can use the **OSSleep** system call to perform this operation. On receipt on this call, RInOS attaches the TCB of the calling task to a linked list of tasks waiting on the timer starting with the system variable **clktsk** and continuing with the SEMA_POINTER field of the TCB. The number of clock periods to sleep is entered in the TIMER_COUNT field of the TCB. Finally the calling task is put to sleep to await expiry of its timer. At each clock interrupt, all tasks in the linked list have their TIMER_COUNT fields decremented and any reaching zero are woken and removed from the list. A call to **OSSleep** with an argument of zero results in the task not being placed in the linked list but nonetheless put to sleep. This means that the task will never wake.

3.7 The loader

The RInOS loader resides in the monitor and was designed to accept position independent code that could be loaded at any suitable address found vacant by the memory manager. Under this scheme, the code is compiled or assembled using an origin of zero (the default in many assemblers) to a Motorola S19 format file and sent to the board via a terminal emulator over a serial line to ACIA1. The size including all code, data and stack requirements is specified on the command line together with an optional priority and argument list. The memory manager reserves the required memory at a suitable location and loads the file into this area. It then calls **OSCreate** to create a new TCB for the process, sets the STACK_POINTER field in the TCB to 12 bytes before the end of the reserved area and fills in the remaining 12 bytes with the default values of the registers in readiness for running the process when the kernel starts multitasking. The final location on the stack, from where the programme counter will be pulled is loaded with the value in the CODE_ENTRY field of the TCB which in turn is obtained from the S9 record of the downloaded file.To initiate a download, the following command must be sent to the board (via a terminal emulator).

```
l <codesize> [argument list]+
```

followed by a carriage return.

A default stack of 0x100 bytes is reserved for the thread, but this can (and should for C programmes) be changed using the ss command:

```
ss <stacksize>
```

4 Compilation tools

4.1 Introduction

A complete set of compilation and debugging tools have been provided mainly through the efforts of Rinus Verkerk. The GNU C compiler was adapted to produce position independent M6809 assembler code and a suitable assembler and linker found. The combination produces relocatable code in S19 format and links together with the standard C libraries, a startup module **crt0**, that performs initialisation by pointing to command line arguments etc. A brief introduction to their usage is presented here.

4.2 The cross compilation and loading chain

4.2.1 Downloading to the ICTP09 board

Code to be downloaded to the ICTP09 board is developed using any text editor and saved to disk. Since GCC can take a very large number of arguments in order to compile even the simplest of programmes, a steering script has been provided that performs a great deal of the work for you. For a full description of the compiler, assembler and linker options, please refer to the manual, "Software for the 6809 Microprocessor board". The script is invoked by:

```
cc09 -treal [-Wall] [-v] prog.c
```

Here, the terms in square brackets are optional but recommended. The `-treal` flag is essential for code to be downloaded to the board, but as we shall see later, must be changed to `-tdb09` when code for the db09 simulator/debugger is intended to be produced. Other options can be include if desired. Following the compilation stage, the script will assemble the compiler output and finally link in the various libraries and startup code. Any errors in the compilation will be reported to the screen, and the user should scrutinise the output to check that compilation completed without mishap. The code must now be downloaded to the board using a terminal emulator. Under Linux, the recommended emulator is Seyon which should be invoked as follows:

```
seyon -modems /dev/modem -- -sb -sl 500 &
```

As an alternative to this long command, if installed, the seyon icon can be clicked. In either case, a board should be connected to a serial port and powered on. When Seyon starts, the reset button on the board should be pressed. This should casuse the RInOS propmt to be displayed. The actual prompt varies with builds, but is currently

```
RInOS version 0.9.1
```

Several methods exist to cause the downloading to start. The easiest of these is to click the **Transfer** button on the Seyon Command Center window that should have been displayed when Seyon was initiated. When the name of the file to be downloaded is supplied, Seyon will issue to the board the sequence of commands necessary to perform the downloading and notify the user when the process is complete. Alternatively, the **Misc** button can be clicked on the Seyon Command Center followed by **Divert** and the name of the file to be downloaded. Prior to this, the command

```
l size
```

must be written to the ICTP09 board, where size is the total size of the programme, its data and any stack.

In either case, when the prompt

```
Task # 02 loaded at address 00:2200+
```

is received (assuming this to be the first file downloaded), the file is ready to be started. This is done by issuing the command:

```
x
```

If debugging is to be done on the board, breakpoints must be set prior to starting with the **x** command as once started, the only way to restart without breakpoints is by reloading the programme.

4.2.2 Debugging on the db09 simulator

It is not too easy to debug satisfactorily on the real hardware, for several reasons. Firstly, a number of inconvenient and annoying bugs in the monitor, make life more difficult than need be. For example, once a breakpoint has been set, it is necessary to remove it prior to restarting. The timer jumper must also be removed if tracing is to be performed. Secondly, breakpoints cannot be set in the kernel itself or in code in any part of the eprom. This effectively restricts debugging to client programmes, which can be inconvenient. On the other hand, there is no substitute for the real hardware when debugging device drivers or any code using the on board hardware.

As an alternative to the hardware debugger, debugging using the db09 simulator can remove many of the aforementioned problems with the hardware. Code destined for use with db09 must be compiled with the **-tdb09 cc09** option. The commands used by db09 are different to those employed by the real hardware and the user is recommended to refer to the complete list in the ICTP09/RInOs user manual. An additional feature recently implemented in db09 is the ability to debug at source code level. This makes the debugging process considerably easier than at the assembler level. Details of this feature will be provided as an addendum to the user manual.

4.3 GUM

As a final point in this section, note is made of attempts to bring all the loose strands of the previous sections together in a unified debugging environment tentatively known as GUM. Under GUM, the user sees only db09: all communication, breakpoints, memory dumps etc are handled by db09 over the serial link. Thus the user has to handle but a single interface and a single set of commands. Again as this is still under development (as of mid July 1999) the user is requested to refer for further details in a separate addendum.

5 Libraries available under RInOS

Libraries are available for provision of a number of functions under RInOS. First, support for the C language is made available via the **libc.a** library. This library contains the standard functions such as printf, putchar, malloc etc demanded by the ANSI C standard. The library is automatically linked in as part of the compilation chain and the functions can be referenced by use of the standard headers **<stdio.h>**, **<stdlib.h>**, **<string.h>**, etc.

The second group of library routines generally consists of wrapper functions for RInOS system calls and allow the user basic level access to the kernel and device driver services. These routines are found in **libcreal.a** and are automatically linked in as required. The routines are accessed by inclusion of the **<syscalls.h>** header file which also includes all basic information on structures and types defined and used by functions making system calls. This header is also automatically called by the standard header files (stdio.h) etc and only if none of these headers are included is it necessary explicitly to declare **<syscalls.h>** in a file. A second library, **libIOreal.a** performs a similar function by bridging the low level I/O services of the kernel to C level functions. These functions are in turn used by the standard I/O functions of the C library. If low level output is required, the header file <ICTP_IO.h> should be included.

The third group of library functions are used by the compiler during code generation and are **libgcc.a** and **libmath09.a**. **libgcc.a** is used by the compiler to perform integer arithmetic operations and to convert between the various integer types. **libmath09** performs a similar function with floating point functions. Neither library should be called explicitly in a user defined function.

Finally, an implementation of the POSIX 1003.1c (pthreads) library is made available to simplify the mechanics of preparing multi-threaded programmes. The implementation is reasonably complete within the confines of the 8 bit microprocessor platform and its use is encouraged as the functions for thread creation and mutex usage in particular offer greatly simplified functionality over the native RInOS functions. As extensions to this library, the following functions allow simple manipulation of (counting) semaphores and events

```
int event.init(event.t *event, const eventattr.t *attr);
int event.destroy(event.t *event);
int event.signal(event.t *event);
int event.wait(event.t  *event);
int semaphore.down(semaphore.t *sema);
int semaphore.init(semaphore.t *sema,
                   const semaphoreattr.t *attr);
int semaphore.up(semaphore.t *sema);
int semaphore.destroy(semaphore *sema);
int semaphore.init(semaphore *sema,
                   const semaphoreattr.t *attr);
int semaphore.destroy(semaphore.t *sema);
```

All the definitions and types used in these function definitions can be found in the header file <pthread.h> which should be included when any reference is made to any function member of the library.

6 Programming examples

6.1 Introduction

Although programming in C under RInOS is quite straightforward, there are several points that should be noted. Code is presented that illustrate some of these points and demonstrate the use of several of the available libraries.

6.2 Creating threads and mutexes

This example uses the pthreads library to create a child thread and a single mutex. First a thread attribute is created and the desired priority of 20 is set using a variable of type struct sched param. Before the child is created, a static mutex is claimed using the down user **sem()**function. This is actually a RInOS wrapper function rather than the pthreads equivalent showing that the two libraries can be mixed at will. The child thread is then created using the **pthread_create()** function. Finally the mutex is released and the parent exits, at which point the child gains the mutex and is able to run.

Note the use of the static initialiser for the mutex. Static initialisation is a convenient method for all types of semaphore creation. In this example the thread owning the mutex will experience a priority boost if its priority is lower than any thread waiting on the mutex. Please refer to the sycalls.h and pthread.h header files for further semaphore types.

```
#include <pthread.h>

/* Child  prototype */
void* child.thread(void *arg);  /* Static mutex construction */
struct semaphore mutex = {MUTEX | SMBOOST, 1, 0, 0, 0};

int main()
{
  pthread.t child ; pthread.attr.t attr;
  /* Child will have high priority, default is 10 */

  struct sched.param priority = {20};

  pthread.attr.init(&attr);
  /*initialise  thread  attribute */
  /*  Set priority of thread */

  pthread_attr_setschedparam(&attr,&priority);

  /*  Get semaphore  before  anyone else  can */
  down_user_sem(&mutex);
  pthread.create(&child, &attr, child.thread, NULL);

  /* Finally release mutex */
  up_user_sem(&mutex);
  return NULL;
}

void*  child.thread(void   *arg)
{
 /* Try to get mutex */
  down_user_sem(&mutex);
  return NULL;
}
```

6.3 Mutex, semaphore and event handling

The previous example showed how a mutex can be created using a static initialiser. The base type for all three semaphore types is the *semaphore* structure defined in syscalls.h which is redefined in <pthread.h> as *pthread_mutex_t*, *semaphore_t* and *event_t* for mutexes, counting semaphores and events respectively. Under pthreads, a mutex would be defined as:

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

This would initialise the mutex to a value of 1, and allow priority boosting by default. The priority boosting uses the priority ceiling protocol and _POSIX_THREAD_PRIO_PROTECT is defined by the implementation. The functions

```
int0 pthread_mutex_lock(pthread_mutex_t *mutex)
int pthread_mutex_unlock(pthread_mutex_t *mutex)
```

are defined as operations on mutexes under pthreads which perform the down and up operations respectively.

Events and semaphores do not form part of the standard pthreads implementation. Extension functions have been added to allow use of the objects in a manner consistent with mutexes. Static initialisation of all semaphore types is similar and follows the outline of the following fragment in which a semaphore and several types of event are defined:

```
#include <pthread.h> /* For function prototypes */

#define INITIAL.VALUE 10
semaphore_t count = {COUNT,INITIAL_VALUE,0,0,0}

event_t pevent = {EVENT,0};  /* Persistent event */
event.t revent = {REVENT,0}; /* Resetable event  */
event.t sevent = {SEVENT,0}; /* Single event,
                                freed after signal */

main()
{
.... /* Create a child */
event_wait(&revent); /* Wait for an event to occur */
}
```

```
void* child(void* artg) {
  /* Child thread created by main */
        ....

  /* Signal any threads waiting for the event */
  event_signal(&revent) }
```

The same fragment using basic RInOS functions would be:

```
#include <syscalls.h> /* For function prototypes */
#define INITIAL_VALUE 10
semaphore count = {COUNT,INITIAL.VALUE,0,0,0}

semaphore pevent = {EVENT,0}; /* Persistent event */
semaphore revent = {REVENT,0}; /* Resetable event */
semaphore sevent = {SEVENT,0}; /* Single event, freed after signal*/

main()
{
.... /* Create a child */
  down_user_sem(&revent); /* Wait for an event to occur */
}

void* child(void* artg)
{
   /* Child thread created by main */
  ....

   /* Signal any threads waiting for the event */
  up_user_sem(&revent)
}
```

Note again the use of an initial value for the counting semaphore. As an alternative to static initialisation, all semaphore types can be created dynamically and initialised separately. Under RInOS however, this is wasteful of memory and is not recommended.

6.4 Memory allocation under RInOS

As the following examples shows, memory allocation follows standard ANSI C practise using *malloc()* and *free()*. In this example 5 blocks of (paged) memory are allocated and then freed.

```
#include <stdio.h>
#define NBLOCKS 5

void main(void)
{
 void* memptr[NBLOCKS];
 int index;

 for (index = 0; index ! NBLOCKS; index++){
   memptr[index] = malloc(256);  }

 for (index = 0; index < NBLOCKS; index++){
 free(memptr[index]); }
}
```

Global or shared memory can similarly be accessed using the pair of non standard functions:

```
void* globalalloc(int size)
void globalfree(void *p)
```

6.5 Accessing system variables

When downloading a programme to RInOS, an optional, run-time argument list can be specified. The startup module, crt0 makes this list available to the programme via the standard argument passing mechanism of argc and argv. As usual, argc is the number of arguments passed to the programme and argv is an array of strings containing the individual arguments. The startup file crt0 is also responsible for making available several other global resources. In earlier versions of RInOS, input and output functions acted on file descriptors rather than file pointers. In the current release it is expected to use the file pointers in the following list:

File	Filename Description	Type
stdin	Standard input	read only
stdout	Standard output	write only
stderr	Standard error	write only
fpia	''lcd''LCD file	read/write
facia1	''com1''ACIA1 file	read/write
facia2	''com2''ACIA2 file	read/write
fadc	''adc''ADC file	read only
fdac	''dac''DAC file	write only

With the exception of stdin, stdout and stderr, all the preceding files should be opened prior to use and closed when finished in the standard manner. Thus the following code fragment will open and later close the the first serial port for writing:

```
#include <stdio.h>

 FILE* f; /* File pointer definition */
 ...

 if ((f = fopen(''com1'',''w'')) != NULL) {
 fprintf(f,...);
}
 ...
 ...
fclose(f);
```

Finally, crt0 provides access to the pushbutton on the LCD board. During kernel initialisation, a resetable event semaphore is reserved and initialised.
On each press of the pushbutton, any threads waiting for this event will be woken. The startup module stores a pointer to the event semaphore in the global variable **pshbttn** and all any thread wishing to wait on this event has to do, is to perform a down using one other of the two functions:

```
        down_user_sem(pshbttn);
        event_wait(pshbttn);
```

The second of these is illustrated in the following code fragment:

```
/* Push button test */

#include <pthread.h>

extern semaphore.t* pshbttn; /* pushbutton semaphore pointer */
void main(void)
{
  event_wait(pshbttn); /* wait for pushbutton to be pressed */
  .... /* Got event, now do something ... */
}
```

7 Bibliography

1. Software for the 6809 Microprocessor board. C. Verkerk and A.J. Wetherilt
2. Modern Operating Systems. Andrew Tannenbaum
3. The MCX11 Real Time Executive. Motorola
4. The ASSIST09 Monitor, Motorola
5. The μ/COS Real Time Kernel, Jean Lebrosse
6. Specifications for the AA16107-LY, Agena Industries Co.
7. PThreads Programming Nichols, Buttlar & Farrell, O'Reilly.