UNITED NATIONS EDUCATIONAL, SCIENTIFIC AND CULTURAL ORGANIZATION
INTERNATIONAL ATOMIC ENERGY AGENCY

INTERNATIONAL CENTRE FOR THEORETICAL PHYSICS

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

Trieste, 7 October - 1 November 1996

LECTURE NOTES

Volume II

MIRAMARE - TRIESTE

October 1996

The United Nations University

Editors:
Abhaya S. Induruwa
Catharinus Verkerk

Presented by: Prof. Indurain - Dec. '96 -

Conclusion

The completion, towards the end of the course, of Volume II of these Lecture Notes gives us the occasion to acknowledge the various contributions to the success of the Fourth College on Microprocessor-based Real-time Systems in Physics made by many individuals.

We sincerely thank the Director of ICTP, Professor Miguel A. Virasoro for his interest in the course and the laboratory work, and for his support.

Special thanks are due to Professor Ines Wesley-Tanaskovic, Dr. Caterina Casullo and Professor Luciano Bertocchi for their encouragement and continued support. We are grateful to the United Nations University and to the International Centre for Theoretical Physics for their respective financial contributions.

Several members of the ICTP staff gave us valuable assistance both before and during the collegee. Our thanks go in particular to Italo Birri, Mohammed Iqbal, Stanka Tanaskovic and Marco Zorzini. Their help has been greatly appreciated.

We wish to acknowledge the work of the lecturers and of the instructors. All of them did a great job in preparing and presenting their lectures, in keeping the laboratory running and giving useful and friendly assistance to the participants. Our sincere thanks go to Imtiaz Ahmed, Chu Suan Ang, Paul Bartholdi, Razaq Ijaduola, Anita Kane, Ravindra Karnad, Carlos Kavka, Ulrich Raich, Pablo Santamarina, Abdellatif Tchantchane, Alexei Tikhomirov, Jim Wetherilt and Wu Geng Feng. A number of them made particularly important contributions in preparing enhanced hardware and software for this course.

The hard work and dedication of the participants made the interaction with them an enriching experience for the teaching staff. We hope that they will all have the opportunity to apply their newly acquired knowledge on return to their home Institutes. We wish them success and full satisfaction in their professional life.

Abhaya S. Induruwa.
Catharinus Verkerk.
Directors of the College.
Trieste. October 1996.

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

Trieste, 7 Oct-1 Nov 1996

Table of Contents

Volume II

Embedded Systems

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

Trieste, 7 Oct–1 Nov 1996

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 ENTER and 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 ENTER key or the DELETE key.
- If ENTER is pressed, the display stops blinking.
- If 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.

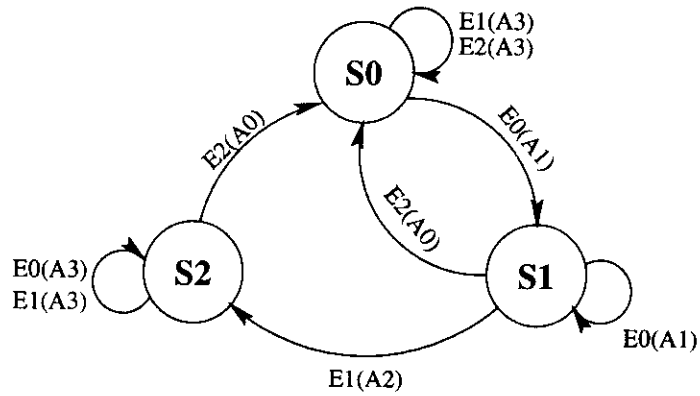
There are 3 types of events:

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.



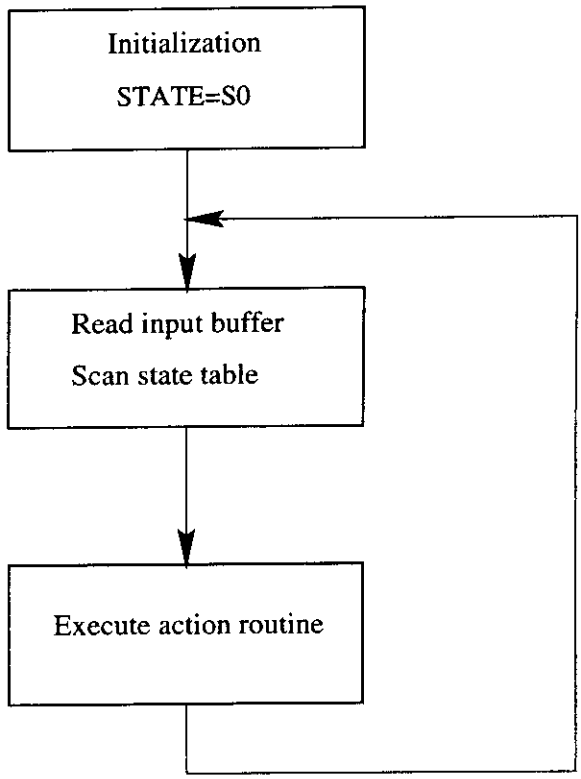
The above state diagram can be easily transformed into a state table representation.

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

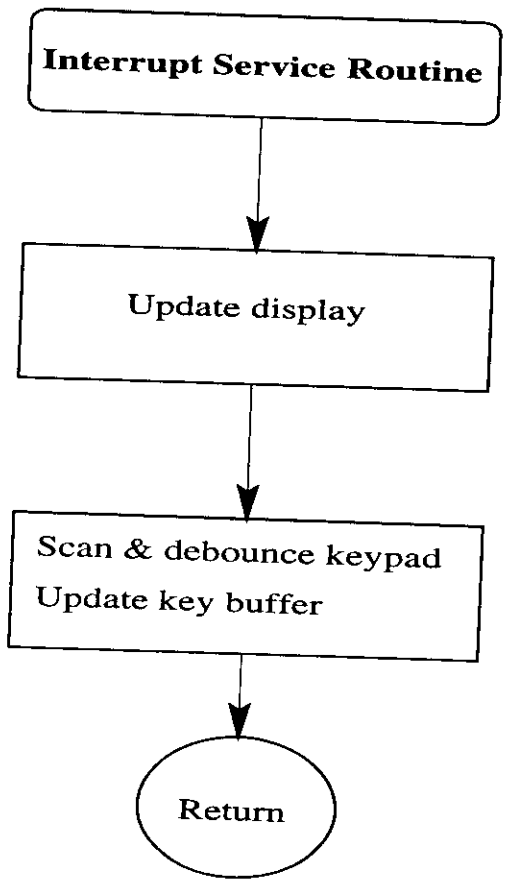
The complexity of the system has thus been broken down into:

- 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

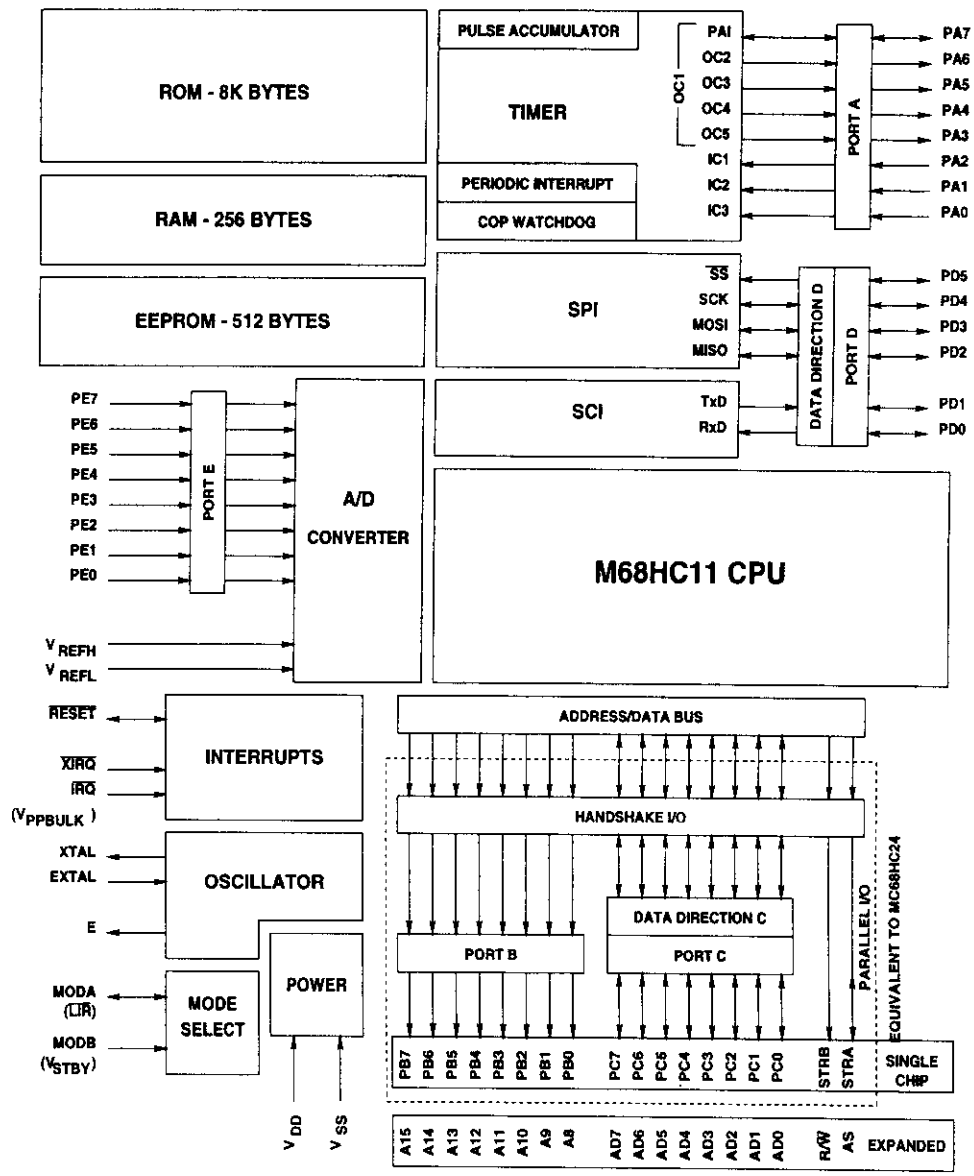
We have mentioned earlier that there are now many microcontrollers available in the market. We shall look at one of them, the 68HC11, in this section. It is a family of microcontrollers with members providing different I/O and memory facilities. They can be used in *single-chip* or *expanded mode.*

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.

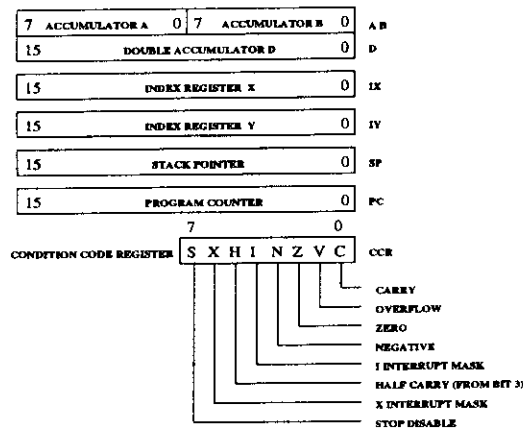
6.1 Architecture of the 68HC11

A simplified diagram of the architecture of the 68HC11 is shown in the figure ??. 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.



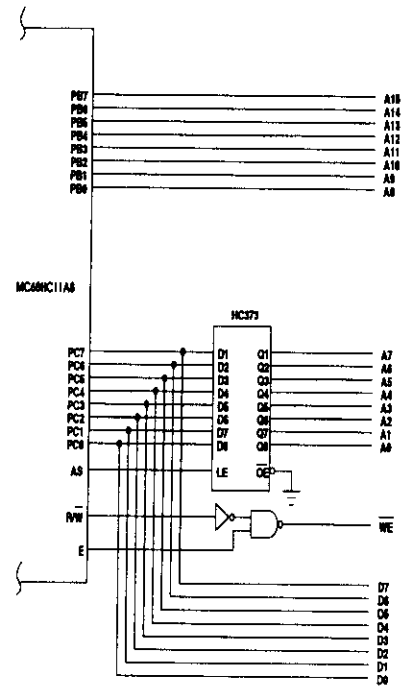
6.3 Modes of Operation

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.

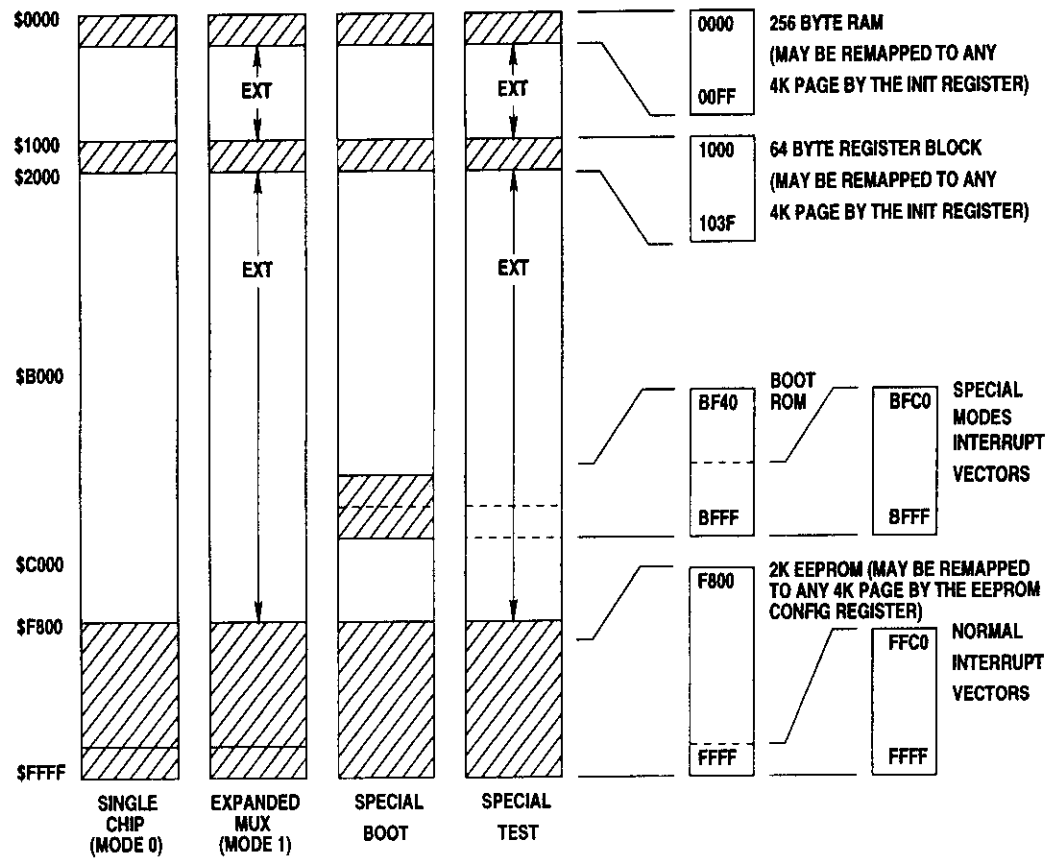
- **Expanded-multiplexed mode**. The chip can access a 64KB address space. The total address space includes the on-chip memory addresses. The expansion is made up of port B and port C, and control signals AS and R/W.



- **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.



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

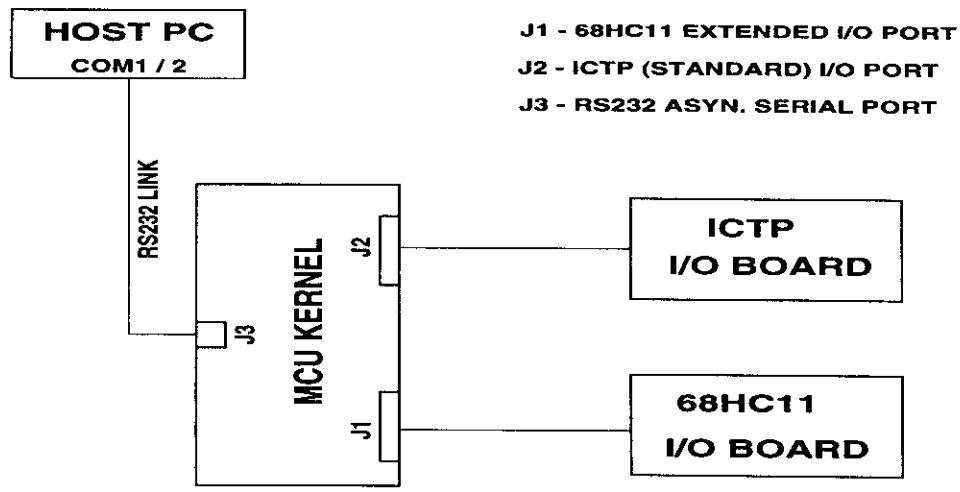
7.1 System Overview

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 microcontroller, 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.



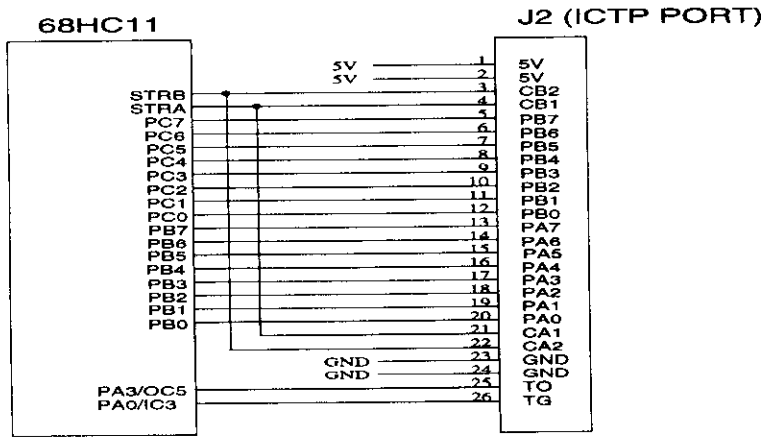
7.1.1 Host PC

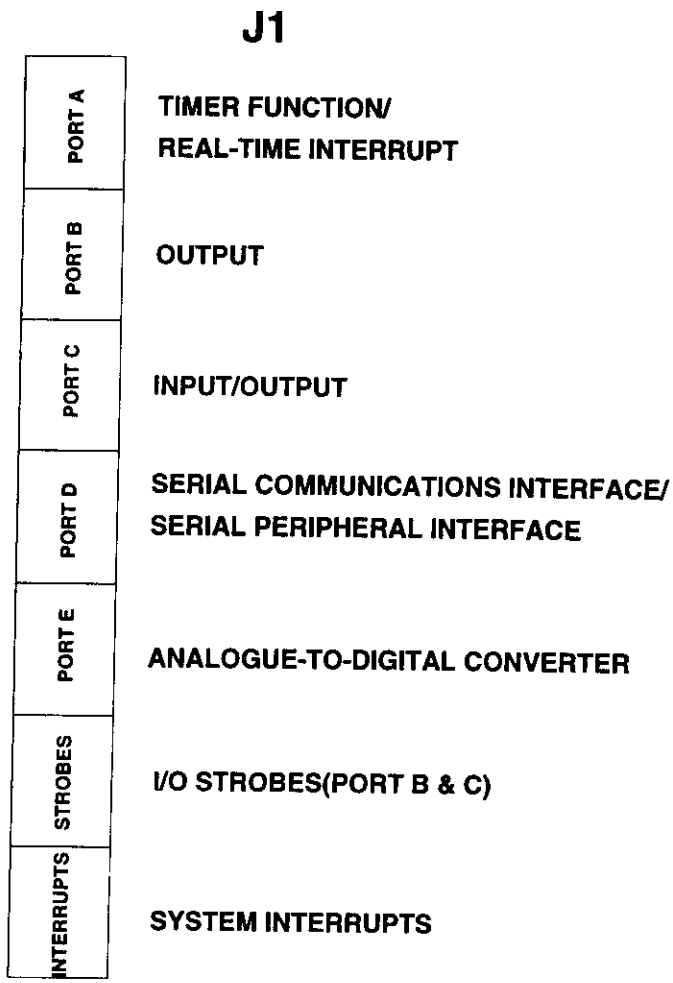
The external host is typically a PC running Linux or other operating system with suitable cross development software for the 68HC11. A COM port on the PC is used to communicate with the 68HC11 target system. This link serves as a code downloading or uploading channel during the development stage. During 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

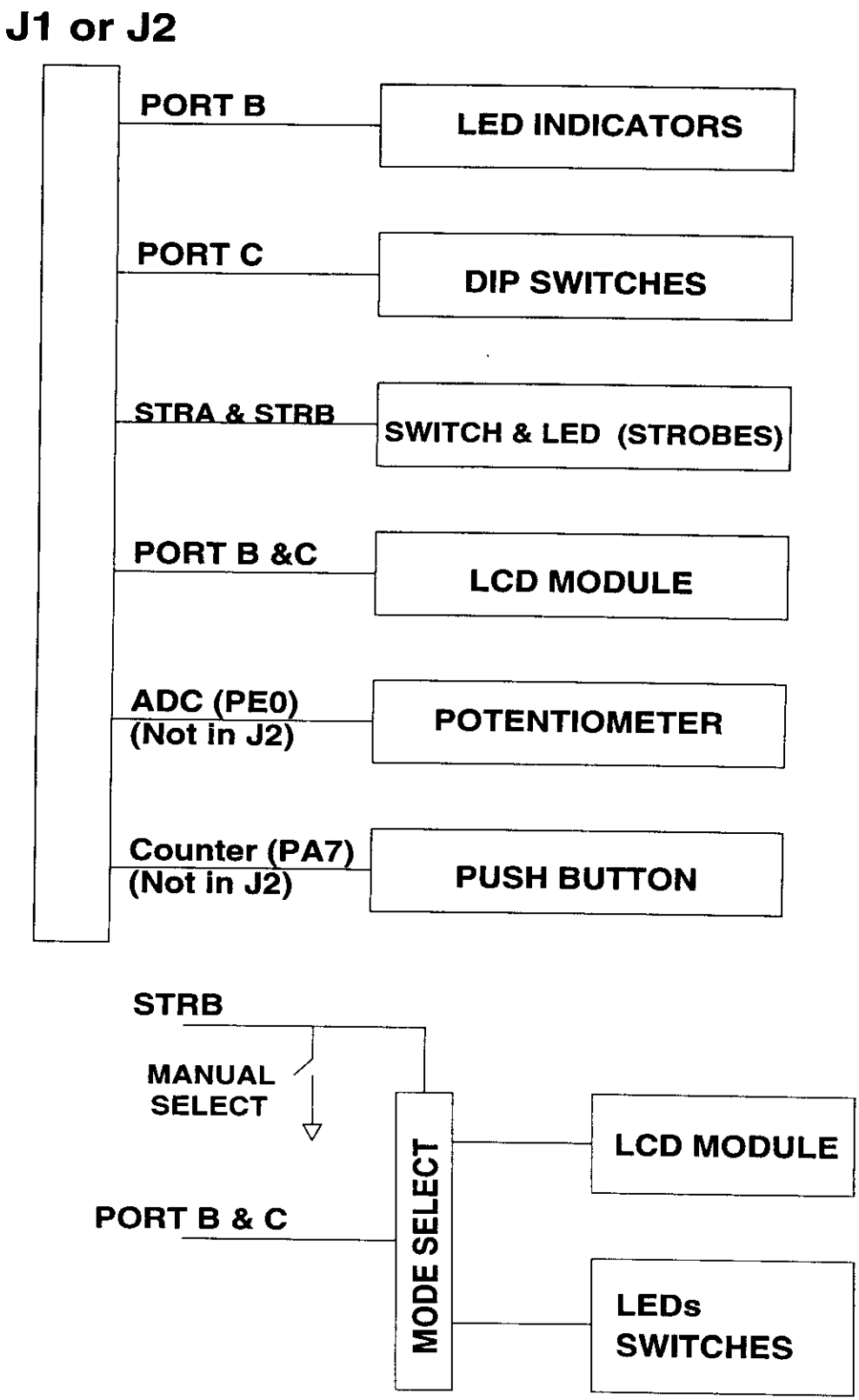
of displaying simple text messages. It is a rather useful *user interface* in a standalone system.

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

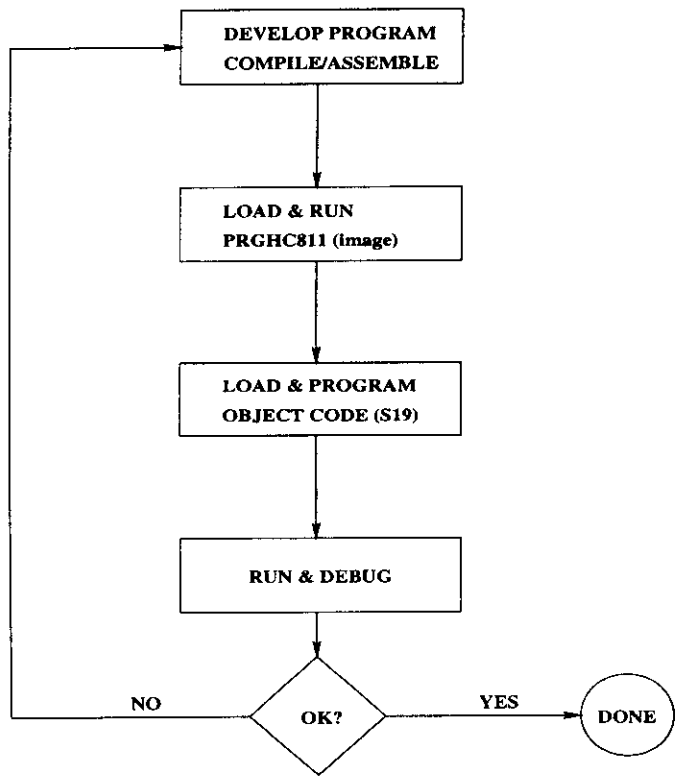
This itself is an embedded sub-system consisting of a twisted nematic mode reflective liquid crystal dot matrix display and an embedded controller and driver in bare chip form directly attached on the PCB. The display appears as a 16-character by 1-line alphanumeric display while internally, as far as 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

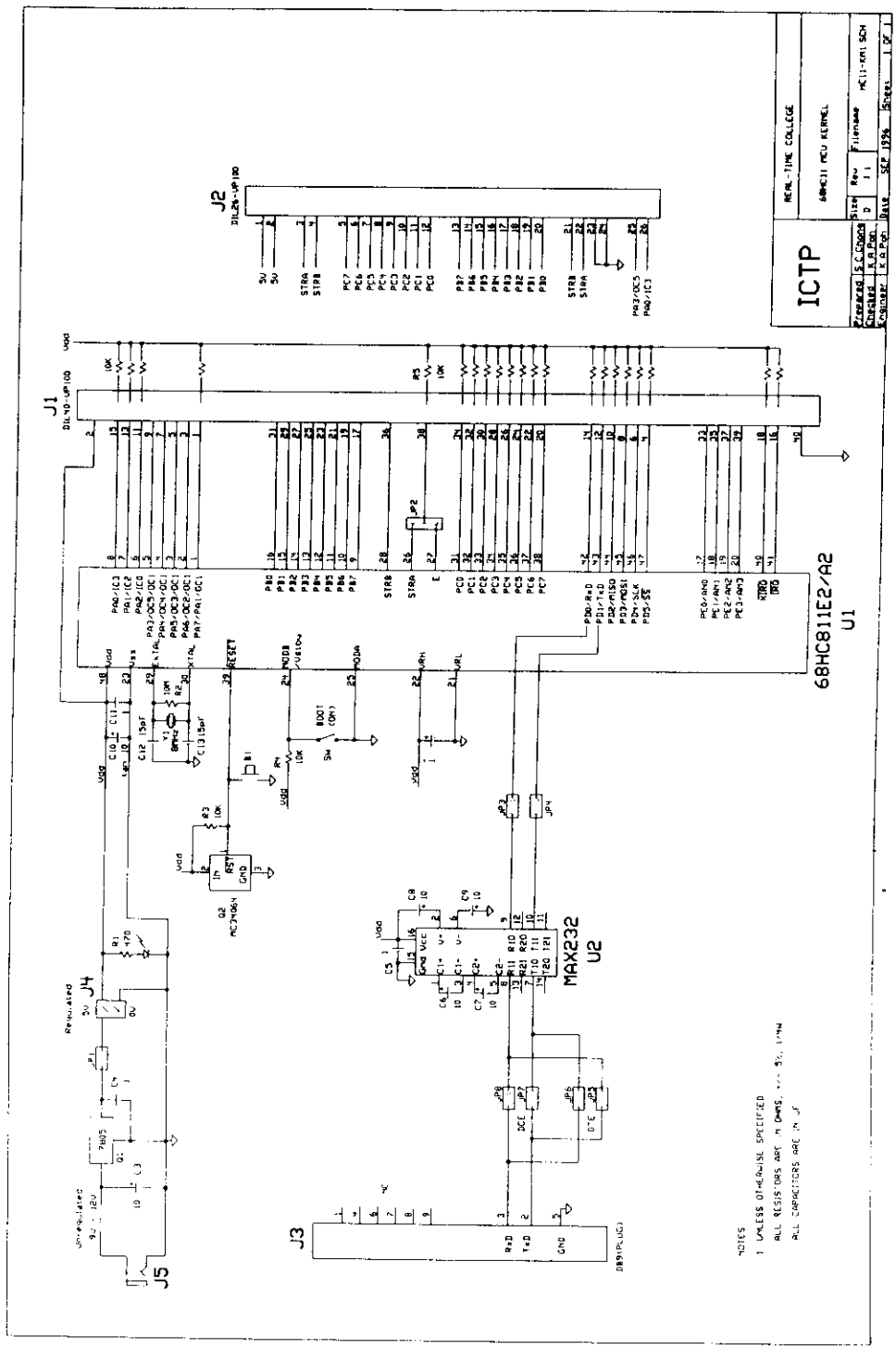
7.4 Program Development

Program development for the 68HC11 Embedded System consists of the following steps:

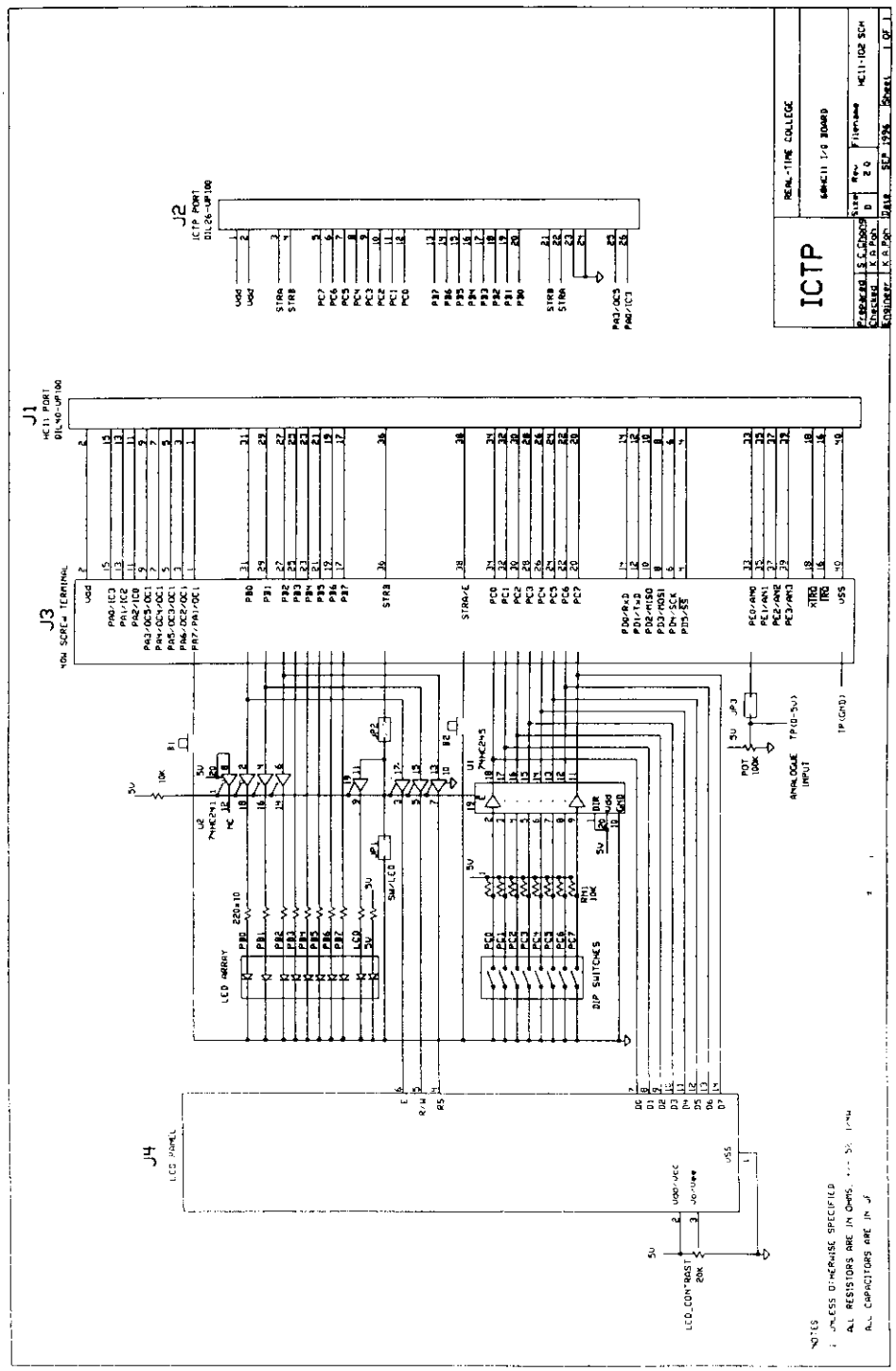
- Develop source program in host environment either in high level language or in HC11 assembly language.
- Compile or assemble source into HC11 object code in S19 format.
- Download and run HC811 programmer (PRGHC811) memory image code to the 68HC11 Embedded System RAM using the bootstrap mode.
- Download 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

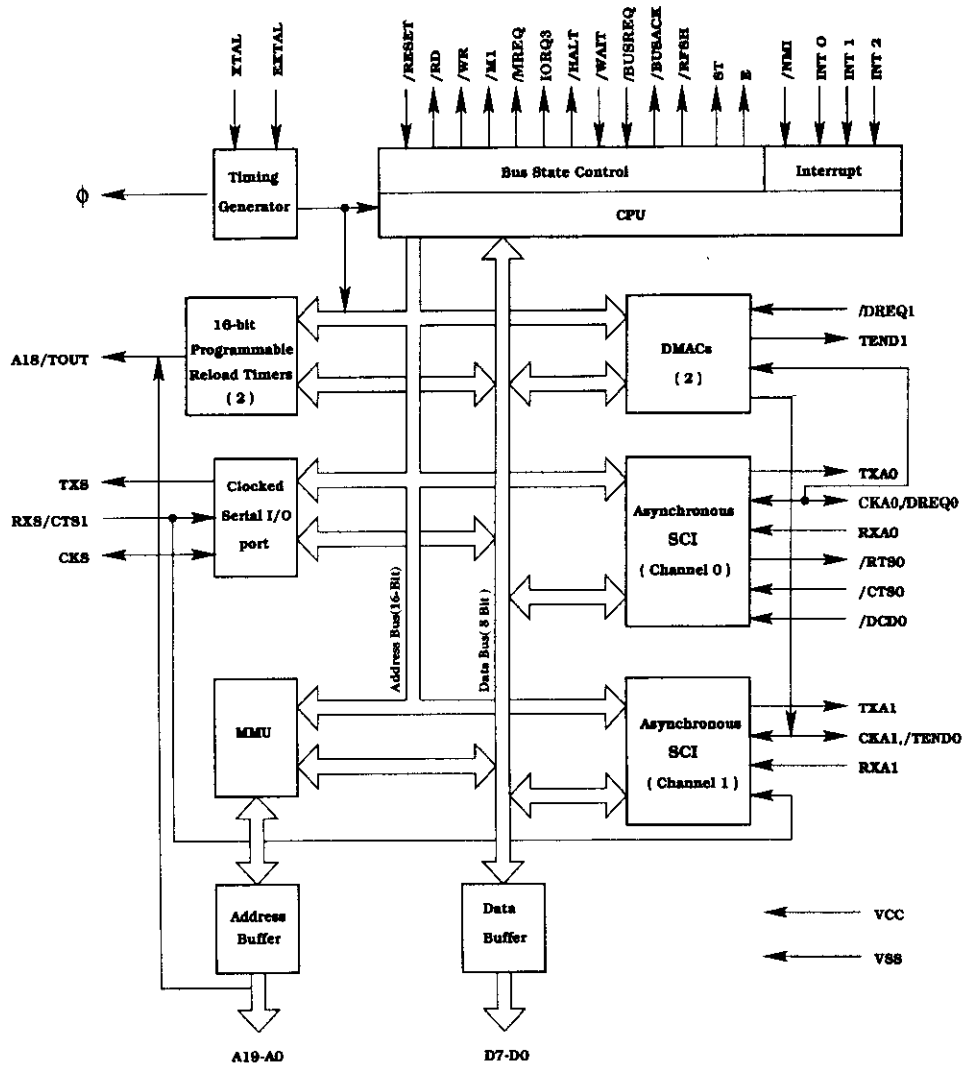
a microcoded execution unit in CMOS, this chip offers rather high performance and maintains compatibility with a large amount of existing Z80 programs.

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.

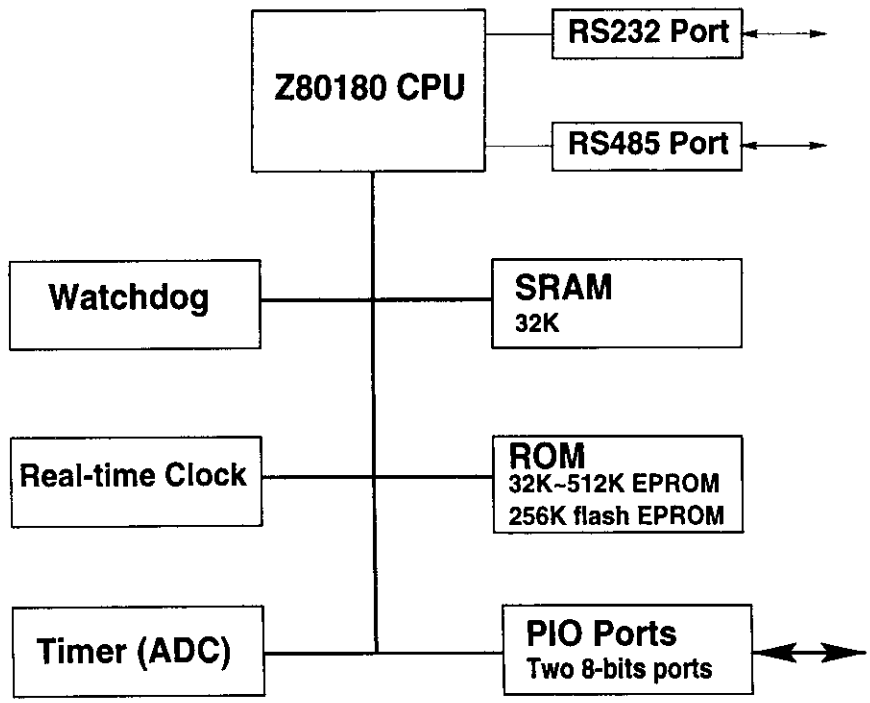
9 A Z80180 Embedded System

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:

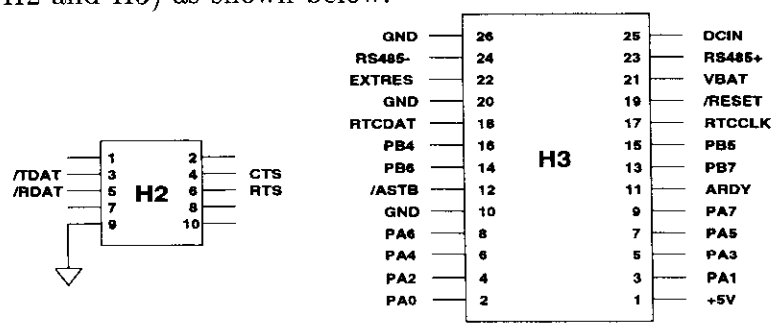
- **Use target system with EPROM.** Use the target system with a Dynamic C BIOS EPROM and connect the RS232 port directly to the PC. The RAM provides up to 32K of code and data space.

- **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.

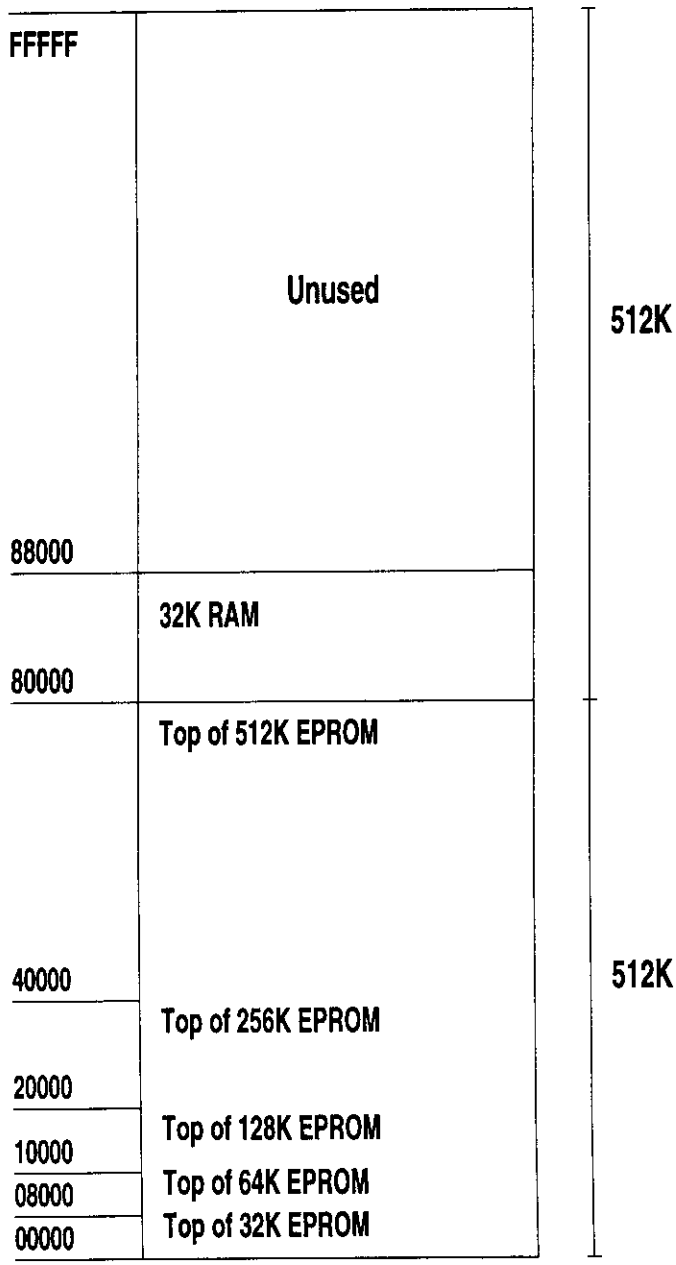
- **Real–time clock** A real-time clock (DS1302) provides time and date function, plus 31 bytes of scratchpad RAM. An external battery (connected to VBAT) is used to retain data when power is down. Data are clocked using $\overline{\text{RTCCLK}}$ and $\overline{\text{RTCDAT}}$. $\overline{\text{RTCRST}}$ 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, μ *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:

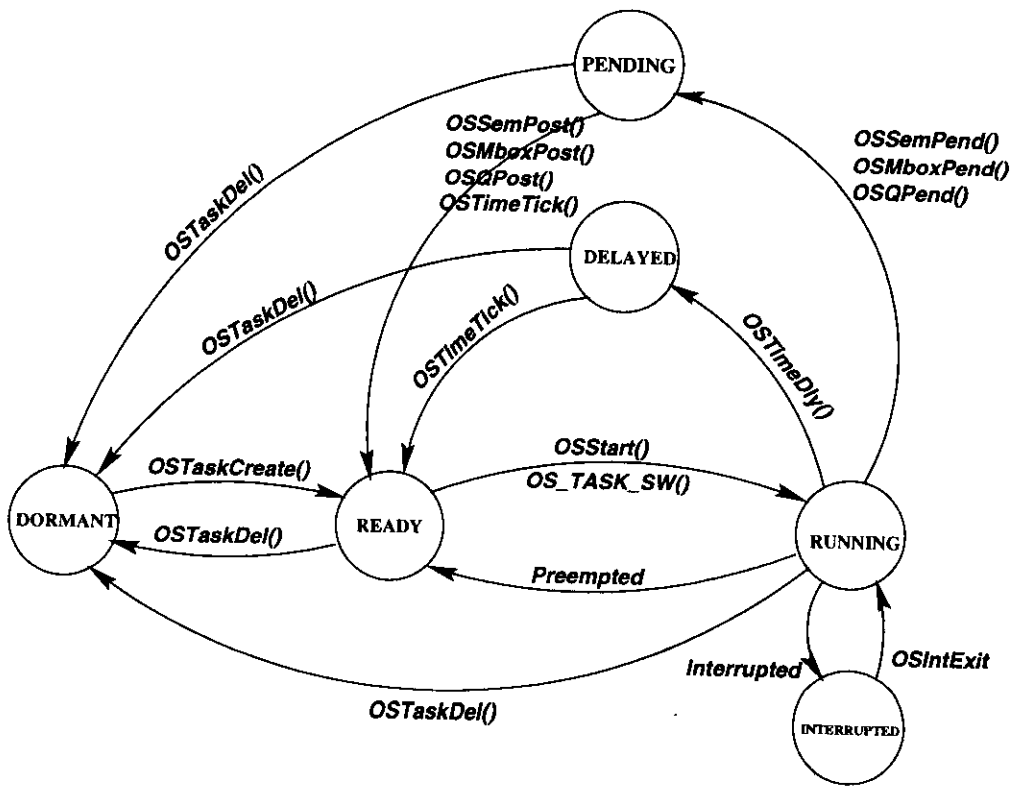
- DORMANT -The state when a task has not been made available to μC/OS.
- READY - When a task is created by calling **OSTaskCreate()**, it is in the READY state. Tasks may be created before multitasking starts or dynamically by a running task. If the created task has a higher priority than its creator, 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**. **OSTimeTick()** 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)

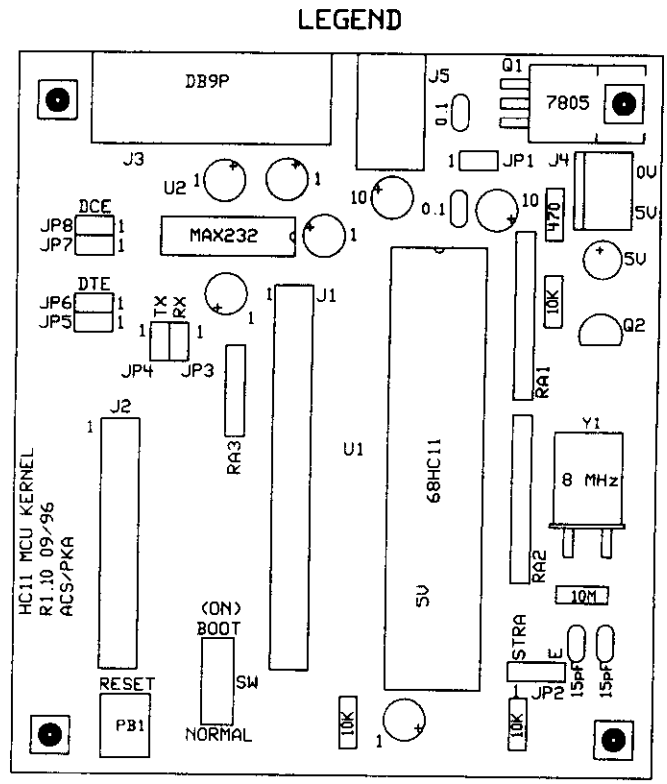
10.15 Kernel Services

The kernel services are given in the following table:

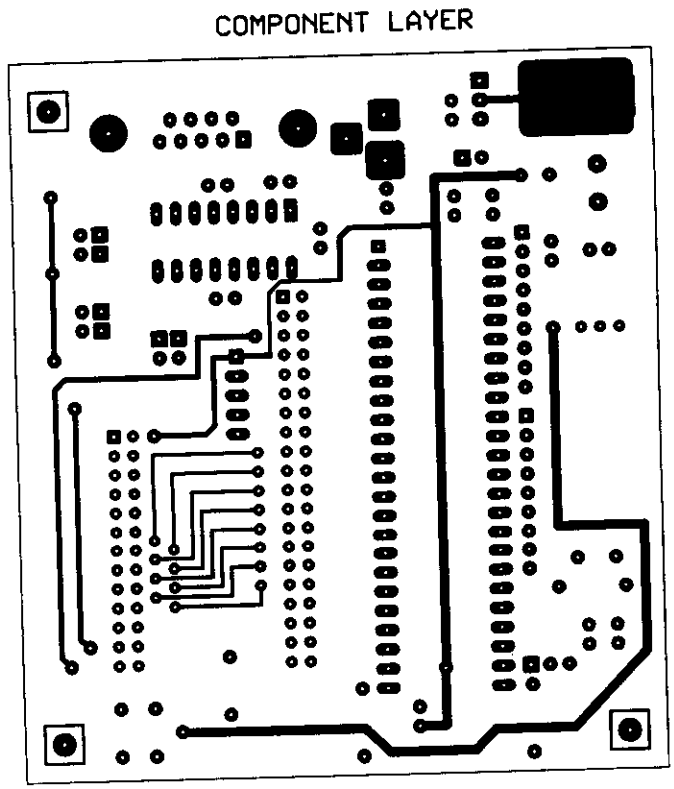
#	SERVICE	DESCRIPTION
1	**OSInit()**	Initialise µC/OS
2	**OSIntEnter()**	Signal ISR entry
3	**OSIntExit()**	Signal ISR exit
4	**OSMboxCreate()**	Create a mailbox
5	**OSMboxPend()**	Pend for mrssage from mailbox
6	**OSMboxPost()**	post a message to mailbox
7	**OSQCreate()**	Create a queue
8	**OSQpend()**	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

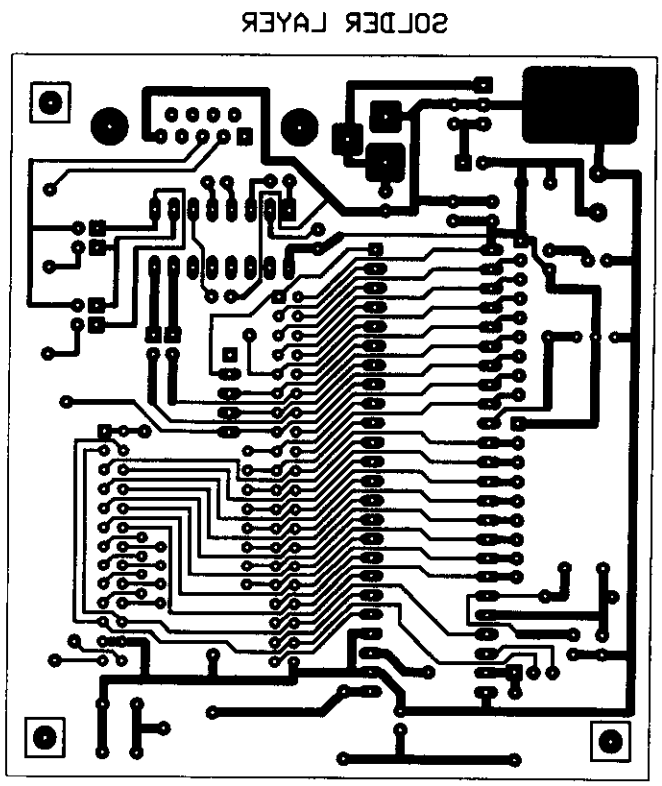
A.1 HC11 MCU Kernel Board



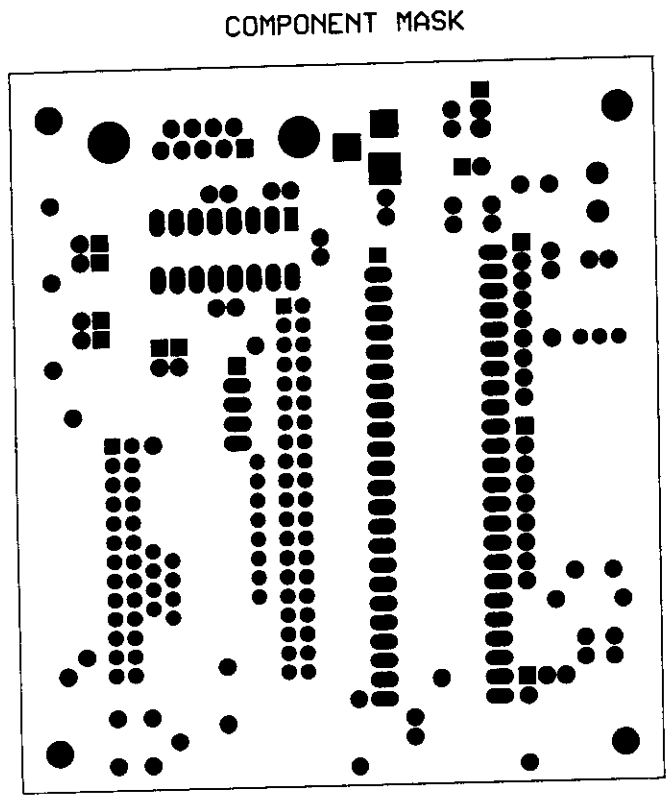
A.2 HC11 MCU Kernel Board



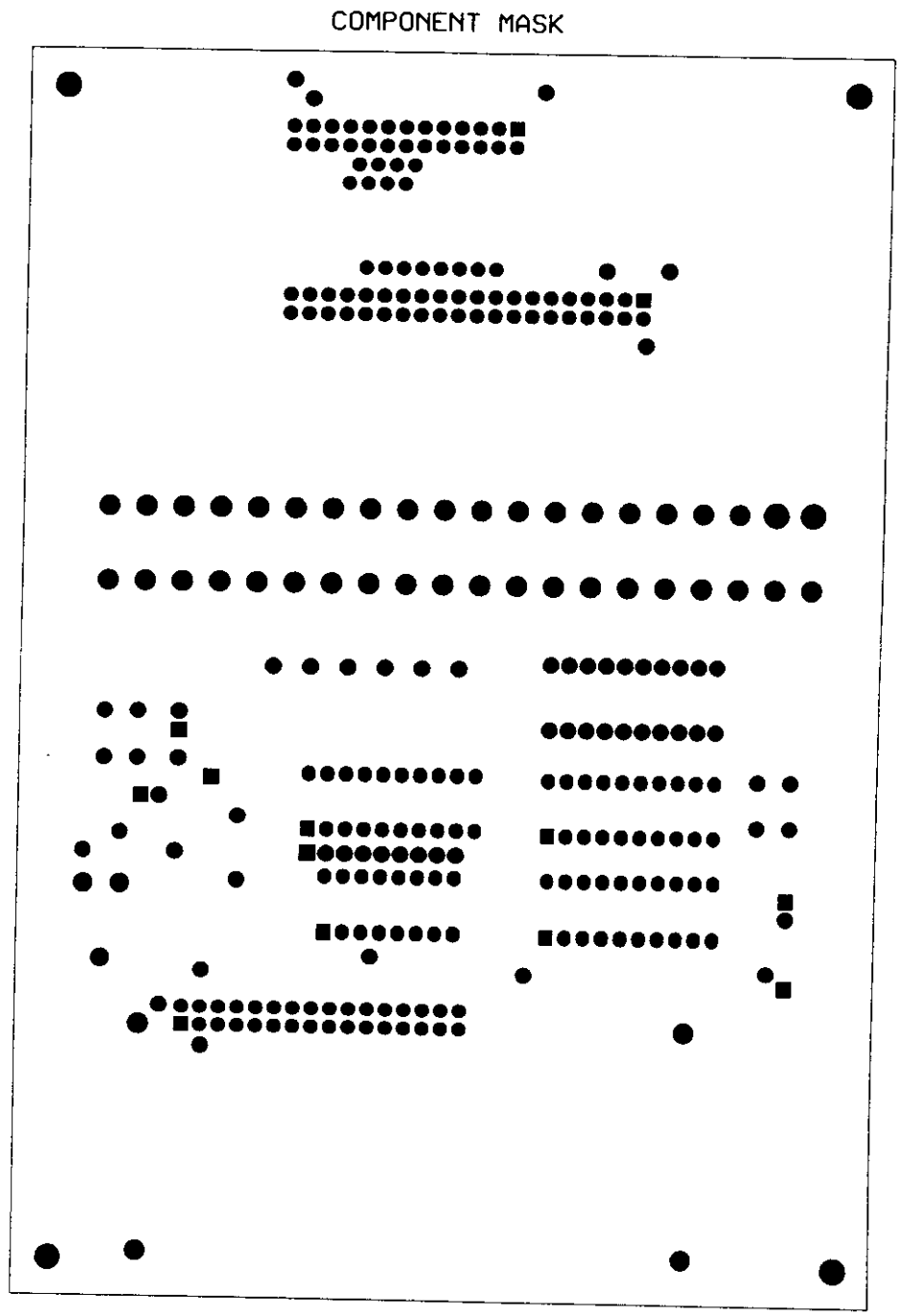
A.3 HC11 MCU Kernel Board



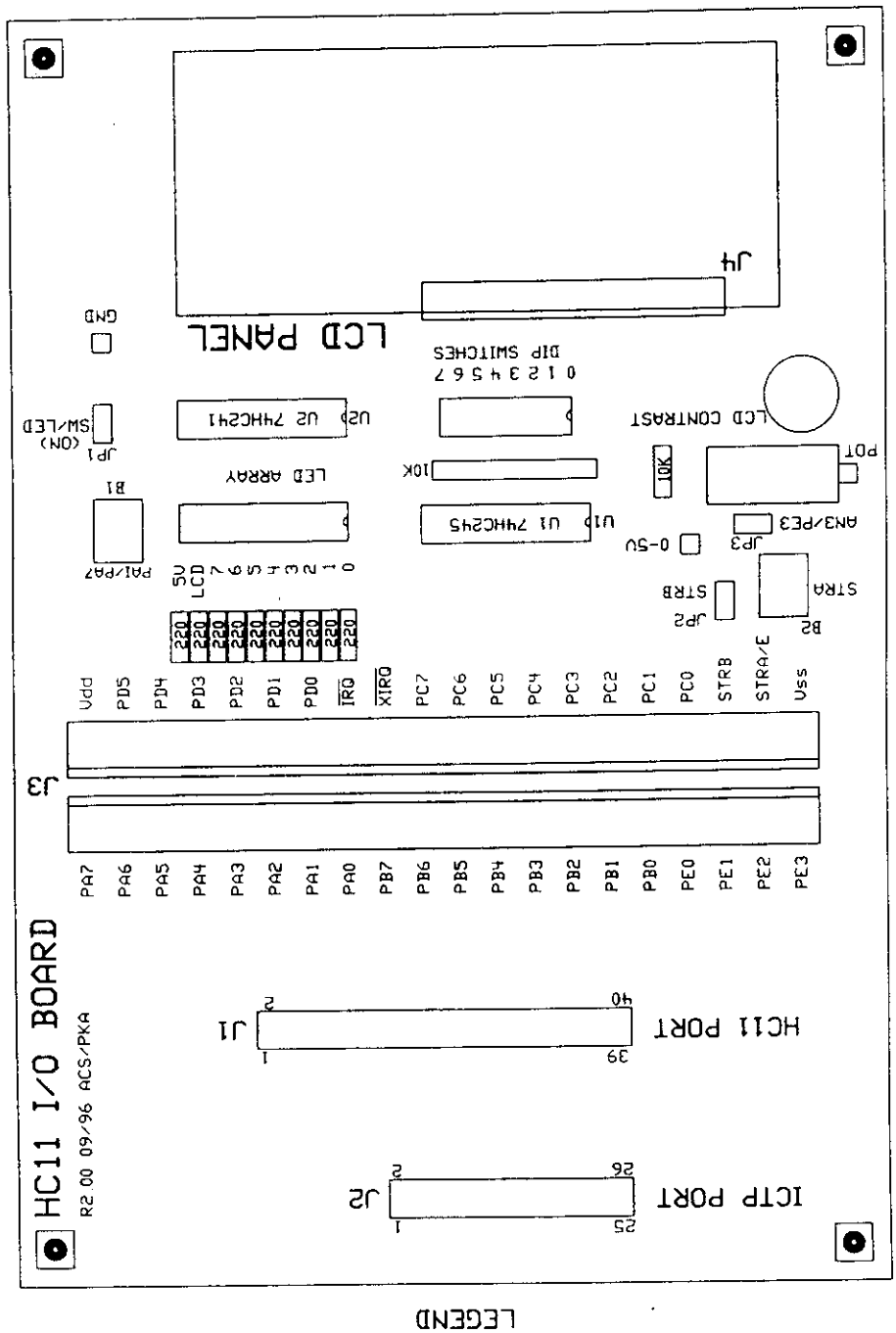
A.4 HC11 MCU Kernel Board



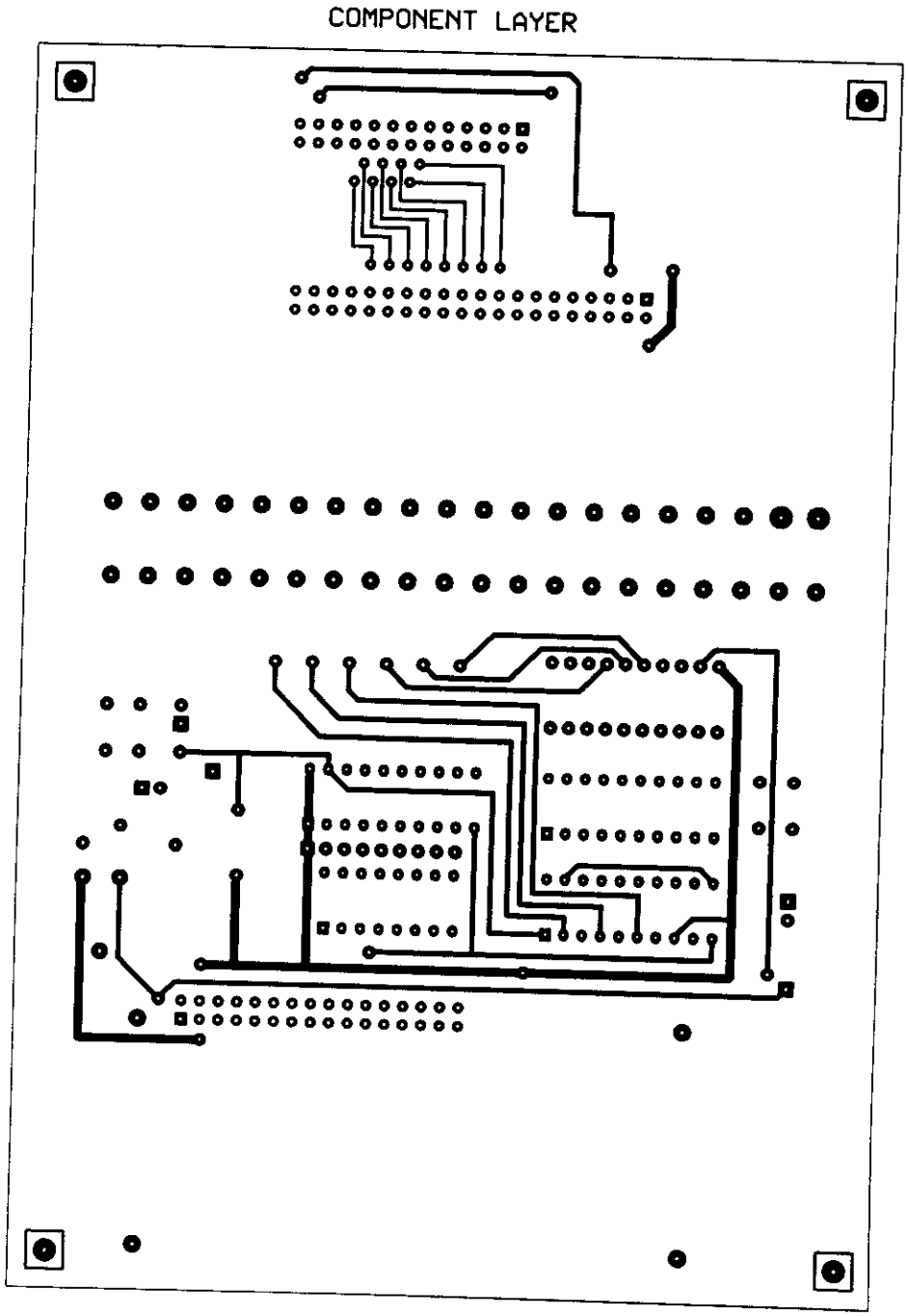
A.5 HC11 I/O Board



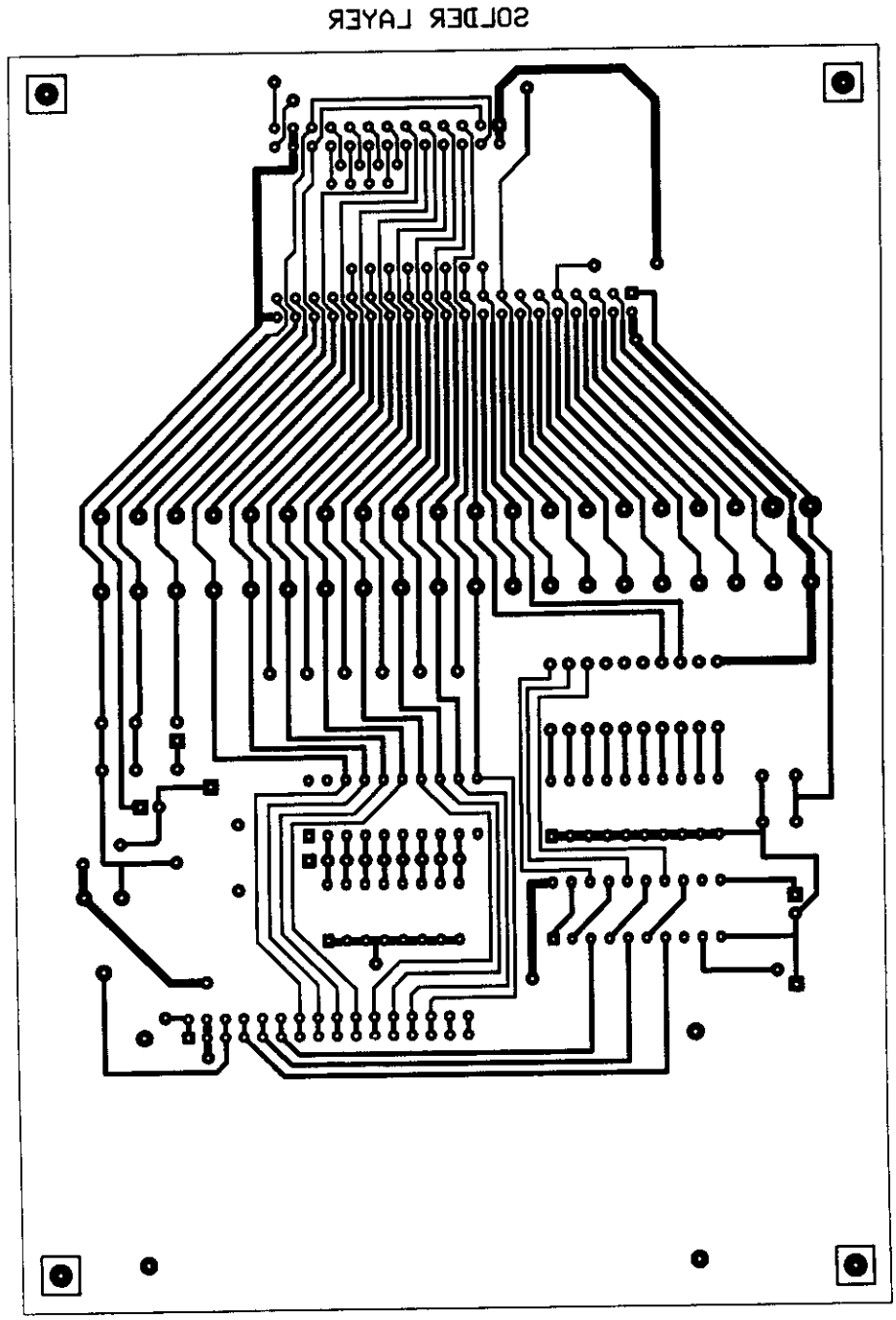
A.6 HC11 I/O Board



A.7 HC11 I/O Board



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   *
                JSR   SW_MODE                    set switches/led mode
                LDX   #REGBS
                LDAA  PORTC,X                    read DIP sw status & disp
                COMA                             on LED array: 1=on, 0=off
                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   *
                JSR   SW_MODE                    turn STRB low for switches
                CLC
                LDAA  #%10000000                 set bit 7 to 1
ROTATE          LDX   #REGBS
                STAA  PORTB,X                    display in port B LED
                STAA  A_REG                      save A register
                TPA
                STAA  CC_REG                     save CC register
                JSR   CHK_STRA
                BCS   XLED_TEST
                LDAA  CC_REG                     restore CC register
                TAP
                LDAA  A_REG                      restore A register
                LDX   #100                       delay 100 ms
                JSR   DELAY_IN_MS
                RORA                             rotate bit pattern left
                BRA   ROTATE
XLED_TEST       LDX   #REGBS
                CLR   PORTB,X                    clear all LEDs
                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           get pointer to display data
                STY   LCD_PTR                   RAM

NX_CHAR         LDY   MSG_PTR                   get pointer to message
                LDAA  0,Y                       get 1 byte of message
                INY                             move pointer to next byte
                STY   MSG_PTR
                CMPA  #ETX                      get out if ETX is met
                BEQ   END_OF_MSG

                PSHA                            save it for later

                LDY   LCD_PTR                   get DD RAM addr, which is
                LDAA  0,Y                       disjoint between the 1st 8
                INY                             and the last 8
                STY   LCD_PTR

                ORAA  #%10000000                form display data addr
                CLC                             select instruction reg
                JSR   WRT_TO_LCD                set display data addr

                SEC                             select data reg
                PULA                            get the byte to write
                JSR   WRT_TO_LCD                write it to LCD
                BRA   NX_CHAR                   process next byte
END_OF_MSG      RTS

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

Fourth College on Microprocessor–based Real–time Systems in Physics

Trieste, 7 Oct–1 Nov 1996

A.J. Wetherilt
Institute for Energy Systems and Environmental Research
Marmara Research Centre
Gebze
Turkey

email: jim@yunus.mam.tubitak.gov.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 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. The development and use of these features is discussed.

1 Introduction

Since the start of the series of Colleges on Real-Time systems and Control, 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 The GPI card

The General Purpose Interface card (GPI) was designed by Manuel Goncalves 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. 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 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

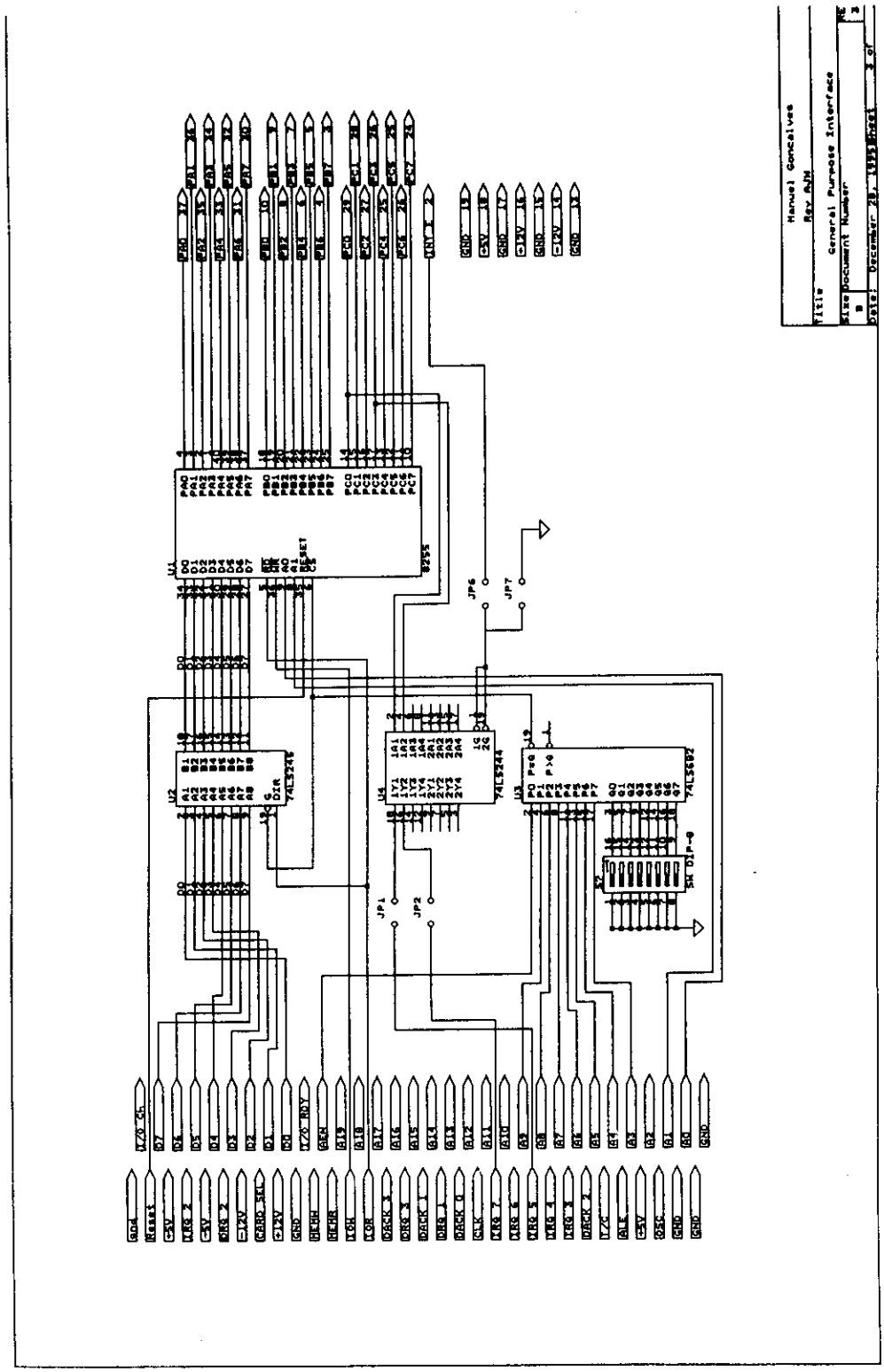


Figure 1: Schematic Drawing of the GPI board

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.

3 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 (see Figure 2). Its other side comprises a 4 digit, 8 segment LED display together with various switches and devices that can be interfaced via a 26 pin connector to either the on-board microprocessor or an external host machine. It is this latter feature that has been used predominantly during the various colleges.

The 26 pin connector definitions are shown in Figure 3 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 (Figure 4): 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 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.

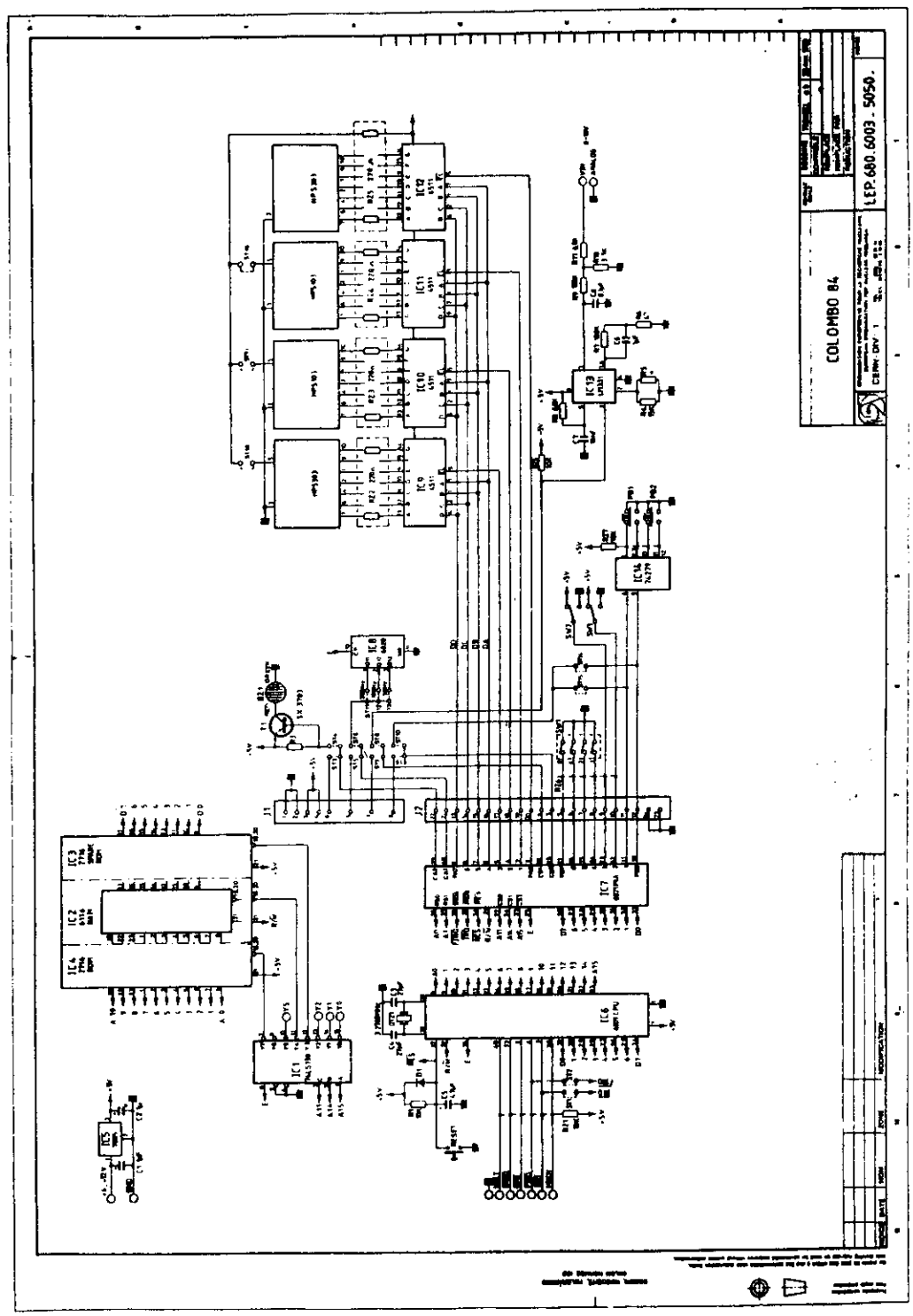


Figure 2: Schematic Drawing of the Colombo Board

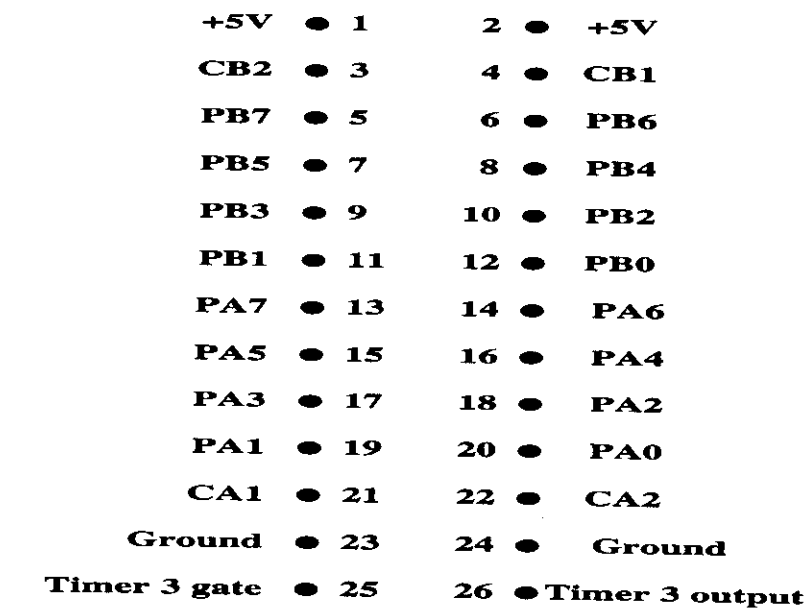


Figure 3: Pin definitions of the standard ICTP, 26 pin strip connector

4 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). 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

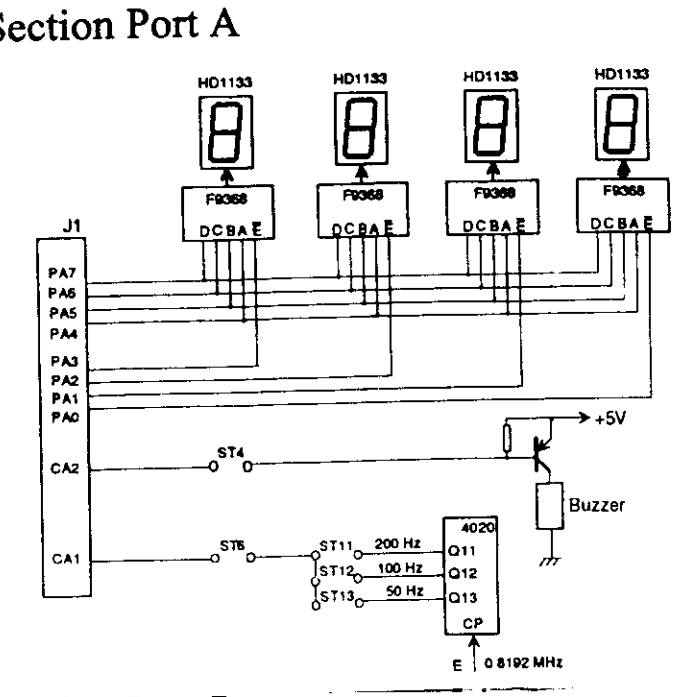


Figure 4: Simplified details of the Colombo Board Showing the A and B Sides

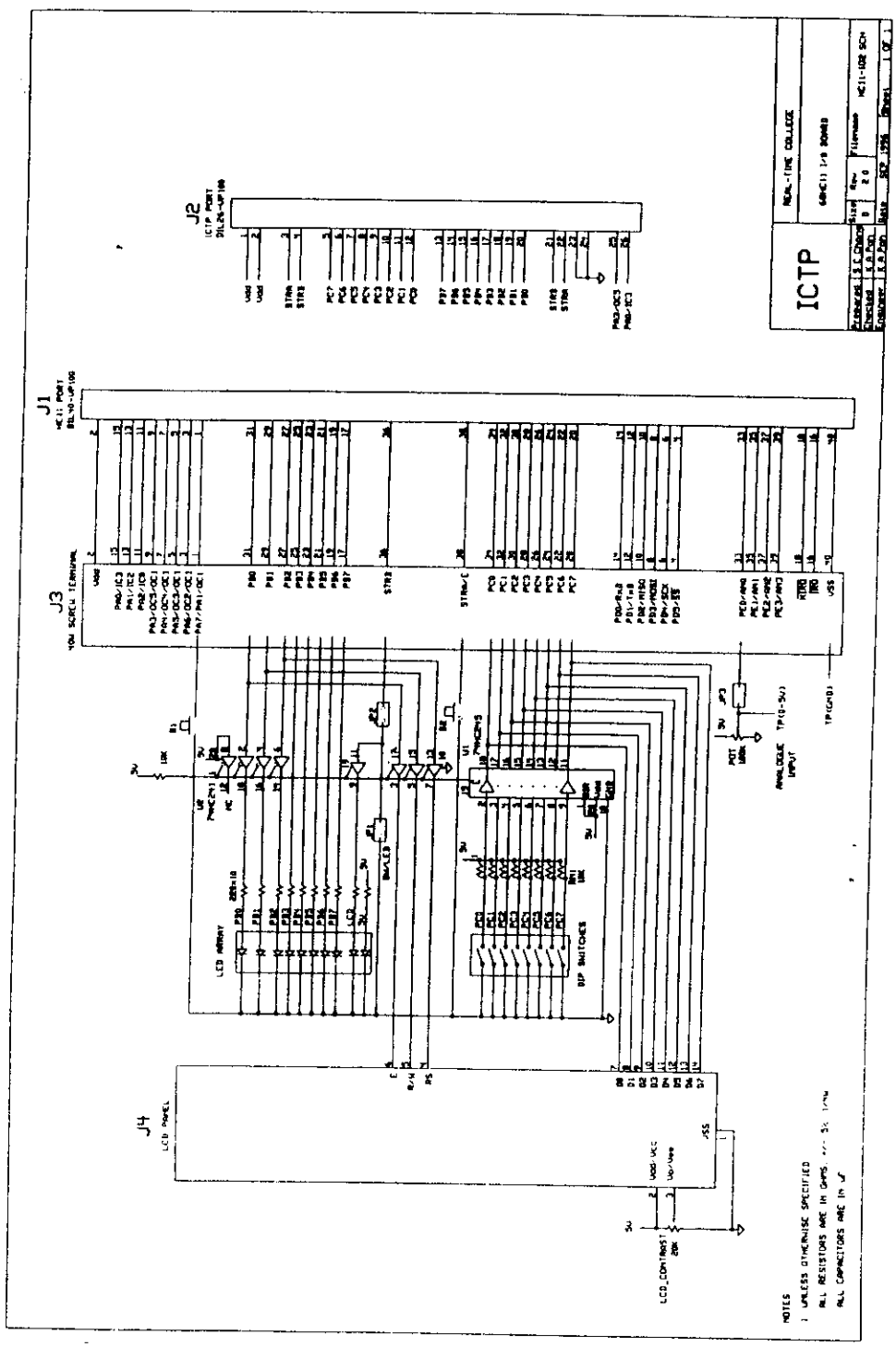


Figure 5: Schematic Drawing of the LCD Display Board

for the data to be latched. For full details of all possible operations, please refer to the manufacturer's data sheet.

If line CA2 is set low, writing to the LCD is disabled and the DIP switch and LED strip become accessible by reading from and writing to, the lines PB0-7 and PA0-7 respectively. In both modes a push button is connected to CA1 to allow interrupt capability. Neither the second push button, nor the analogue voltage are connected when using the ICTP connector.

5 The M68HC11 board

This board was also designed by C.S.Ang for this College and provides a complete but minimal system with but three IC components (Figure 6). The board's designer discusses its details in his notes elsewhere and the reader is referred to these for further information

6 The MC6809 board

6.1 Introduction

This board was also designed by the author for the present College to provide better coverage of embedded systems programming. In contrast to the 6811 board, the 6809 board implements a large number of functions at the price of complexity: Some 24 integrated circuits are used in its construction (see Figure 7). It comprises:

- 2 serial communications ports
- 1 parallel port
- 3 timer channels
- 2 channels of 12 bit ADC input
- 2 channels of 12 bit DAC output
- 16k EPROM
- 8k base RAM
- 128k RAM arranged in 4 pages, each of 32k

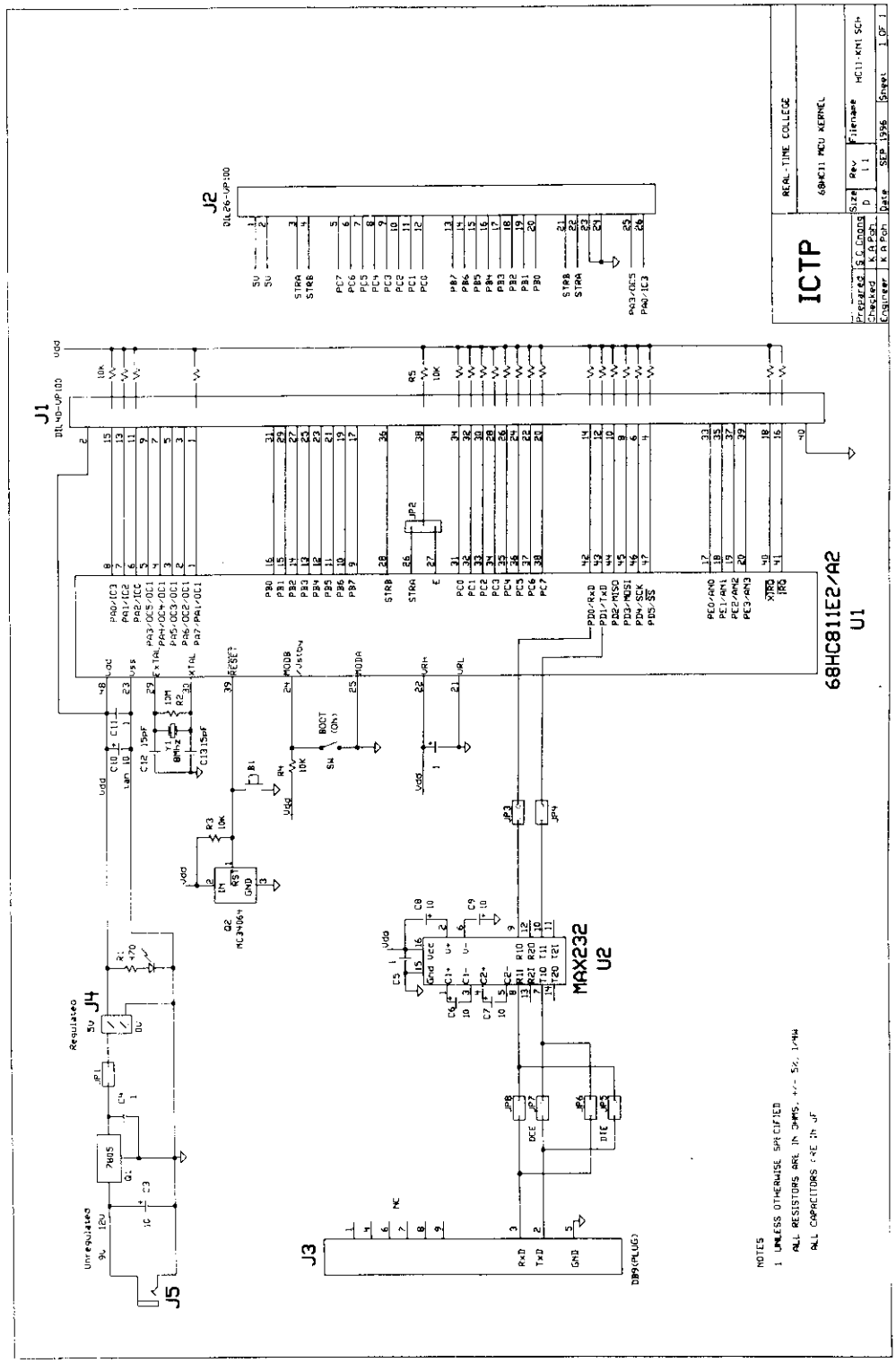


Figure 6: Schematic Drawing of the M6811 board

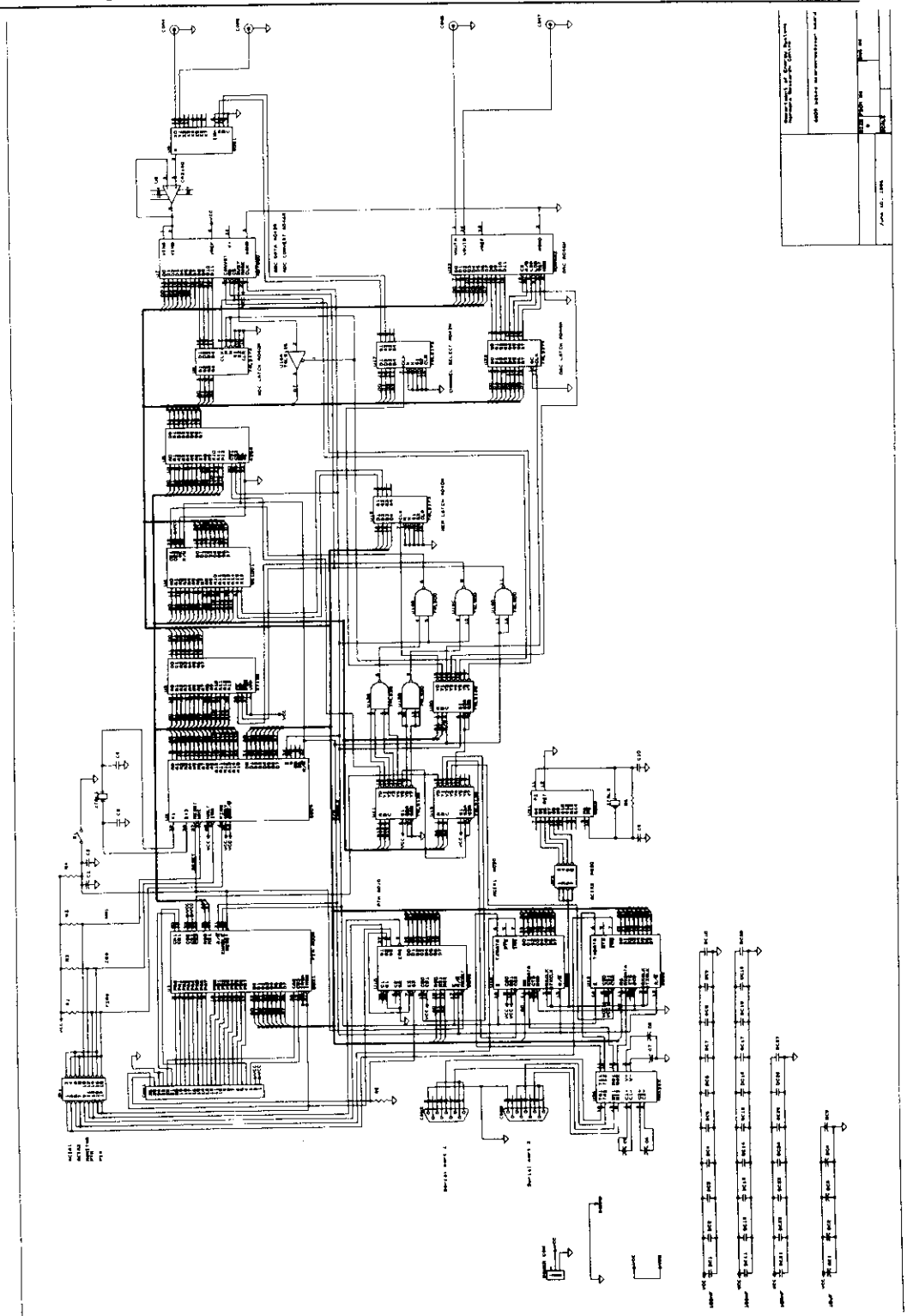


Figure 7: Schematic Drawing of the MC6809 board

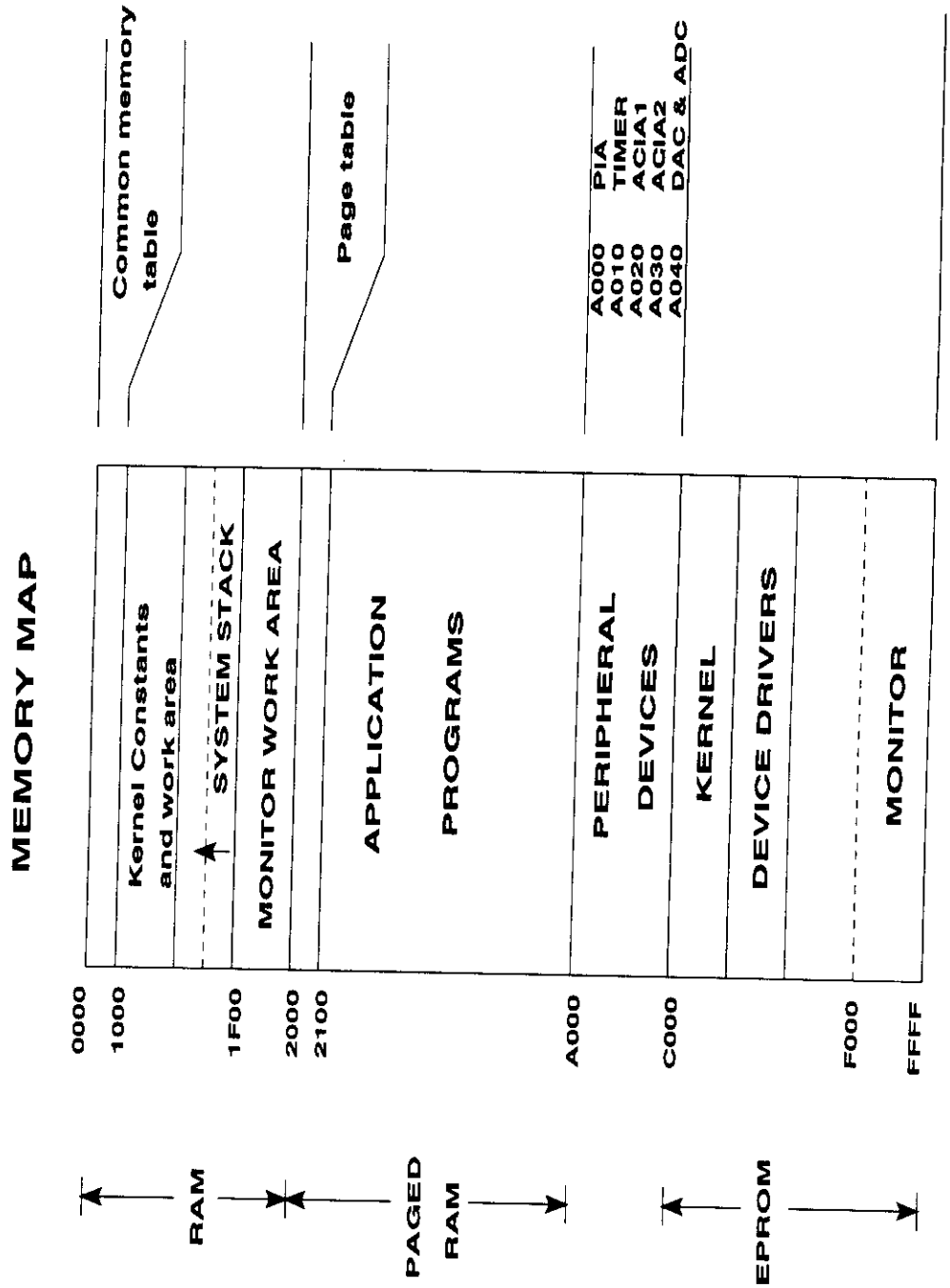


Figure 8: Memory Map of the M6809 under RInOS

The memory map of the system is shown in Figure 8.

In order to facilitate programming, the ASSIST09 monitor provided by Motorola has been modified to reflect the structure of the board and resides in the EPROM at address 0xF000. The monitor allows full debugging and downloading facilities including the setting of breakpoints and tracing through instructions. A real time kernel, RInOS (or Real time INtegrated Operating System) has been written that provides the basic functions of a real time multitasking environment such as context switching, semaphores, message passing between tasks, priority changing, hardware interrupt handling and device drivers etc. As these services are provided in the form of 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.

6.2 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 7) 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 RS 232 9 pin ports. For interconnection between the board and a host PC, null modem cables must be used.

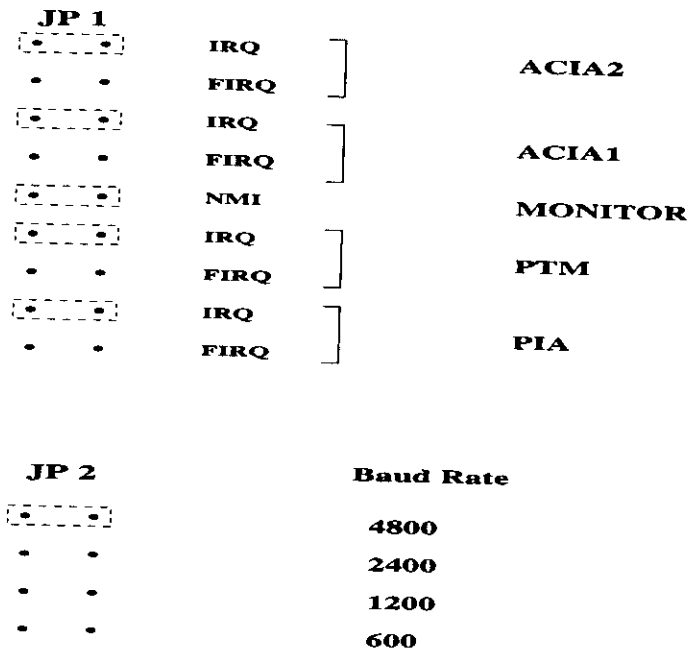


Figure 9: Jumpers settings for the M6809 Board

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 provision will cause unpredictable results and generally will hang the system. Refer to Figure 9 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. 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.

6.3 The ASSIST09 Monitor

The ASSIST09 monitor is made available by Motorola to provide a full range of debugging tools for the 6809. This version has been adapted and extended to fulfil the requirements of the paged memory and kernel. The commands given in Table 1 are supported. All commands are lower case and must end with a carriage return.

Table 1: Commands supported by the ASSIST09 Monitor

l	Load absolute
l *size* [, *priority*] [*arg1 arg2* ...]	Load relocatable module of length *size* priority priority with argument *list arg1, arg2, ...*
g	Go from current address
g [p:]*addr*	Execute from address p:*addr*
x	Start kernel execution.
b	Display breakpoint list
b [p:]*addr*	Add address p:*addr* to breakpoint list
b -[p:]*addr*	Remove address p:*addr* from breakpoint list
.	Trace a single instruction
r	Display/modify registers
d *addr size*	Display *size* bytes of memory starting at *addr*
m *addr*	Modify memory location *addr*
smp *p*	Set memory page to *p*
rmp	Get memory current memory page
pid	Get pid of current task
sp *p pid*	Set priority of task with pid *pid* to *p*
rp [*pid*]	Get priority of task *pid* or current task if no argument
t *n*	Trace *n* instructions
ctrl x	Cancel current instruction

Note

i that segmented memory addresses refer to paged memory. If a page is not specified, it defaults to the current page

ii The go and x instructions only return control to the monitor if a breakpoint is encountered. Otherwise, the monitor effectively is killed as a process

iii Breakpoints are allowed only in RAM.

6.4 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.

6.4.1 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 only runs 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 **hipri** 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 2. 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.

Table 2: Task Control Block structure fields

void *	PRIORITY_POINTER	Link to next highest priority task
char*	PID	Unique task identifier
char	PRIORITY	Priority level
char	STATUS	Current status of the task. Possible values are: READY WAITING 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_POINTER	Used to save the current context of a task during a system call or hardware interrupt
char	PAGE	The page of memory allocated to the task
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*	ENVIRONMENT	A pointer to the optional argument list passed when the task was downloaded

This sequence is illustrated in Figure 10. 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.

(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

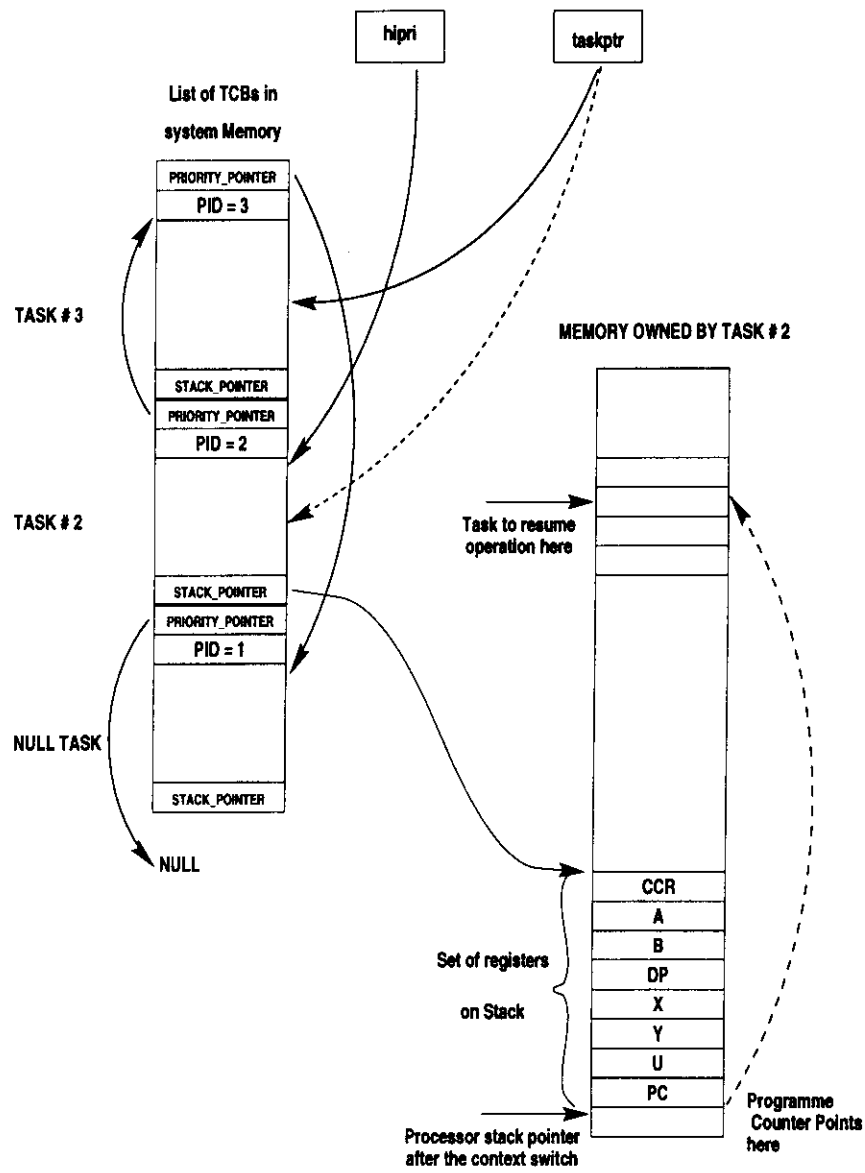


Figure 10: The steps involved in a context switch. Task #3 was running when an event occured that made task #2 runnable. Task # 3 is thus prempted and **taskptr** is changed to point at task #2. The STACK_POINTER field of the TCB belonging the task #2 points at the set of registers pushed onto the stack 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.

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 process identifier or pid 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.

6.4.2 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

(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 3.

Table 3.: 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

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. Appendix 1 gives a full description of this and all other system calls available to an applications programme 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.

A complete list of the functions and modes for the device drivers is given in Appendix 2.

6.4.3 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(see Appendix 1 for full details). 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.

6.4.4 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. 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 4

Table 4: 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

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 11).

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 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 is always initialised to 0 and any task issuing a DOWN on it automatically is

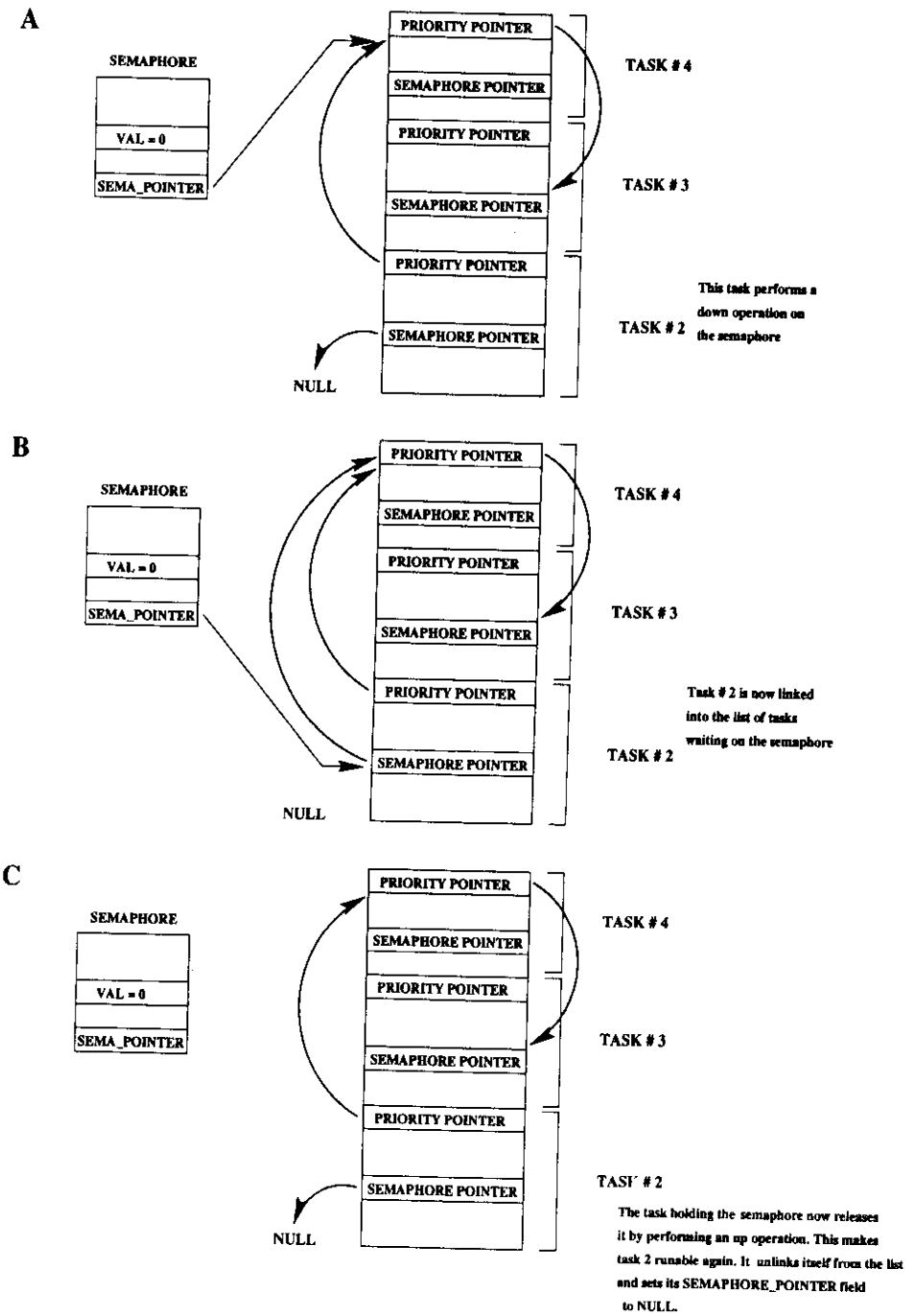


Figure 11: Steps involved as a task waits on a semaphore. (A) task #2 performs a down operation and is blocked. (B) Task #2 is linked into the list of tasks waiting on the semaphore. (C) The semaphore is released alowing task #2 become unblocked and to resume running.

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. Up to 128 semaphores can be created and be in use at the same time.

6.4.5 Loading and running tasks and interprocess communication

The RInOS loader resides in the monitor and allows two types of process to be downloaded to the on-board RAM. RInOs 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 certainly 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 was obtained from the S9 record of the downloaded file. Many compilers, however, do not produce position independent code, but rather produce absolute code that has to reside at a specific address. GCC is currently in this category, and although other compilers exist that produce relocatable code (for instance the MCC compiler) these are not yet sufficiently reliable for current use. RInOS can accept absolute code using the unmodified ASSIST09 loader. However, at present, a file loaded by this scheme neither has memory reserved for it nor does it have a TCB created, and therefore cannot run when the kernel is started. These operations have to be performed manually from within the user process. During the linking process, the module crt0 is included into the final code. This file contains a jump to the **OSInstall** system call which performs all the necessary actions to create the task and to reserve memory for it. Its final action is to start the kernel and wait for control to be transferred back to the address following this jump. By default, all absolute code

will start at address 0x2200 on page 0 of the paged RAM, and in order to run the kernel the user should issue the command

```
g 2200
```

followed by a carriage return.

Each of these two distinct types of code has its own advantages and disadvantages. For relocatable code, as many separate tasks as needed (up to the system limit of 31) can be loaded individually. When loaded, each of these processes has absolutely no information concerning the details such as pid, semaphore numbers etc. of any other task. A task can examine its own details by issuing the **OSTask** system call and getting a pointer to its own TCB, but has no access to information belonging to any other task. To circumvent this problem, a task 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. The process will then issue the **OSSignal** system call and communicate the address of the block of memory to all other processes that require this information. These processes being of higher priority will have already had an opportunity of running before the signal is sent. These tasks should issue the system call **OSWaitSignal** as one of their first actions and sleep pending receipt of the signal. On waking, each process can fill the fields of the block with any of its own information that is required by the other tasks. In this manner shared memory can be established. If necessary, a semaphore can be created to ensure access by only one task at a time, but this is dictated only by the requirements of the applications.

For absolute code, only one file can be downloaded at a given time. Thus, the file should contain all the tasks that will run as separate threads within the process. The starting process should create a child task for each thread it wishes to run prior to going to sleep indefinitely. Since the tasks created under this scheme all come from a common file, they can share data naturally and without problems. Care should however be exercised to ensure that critical data can be accessed only by one task at a time.

In addition to the numbered signals discussed above, RInOS also offers the possibility of sending messages between tasks 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 has the structure shown in Table 5. and 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.

Table 5: Message structure fields

char	SENDER	PID of sending task
void*	NEXT_MESSAGE	Pointer to next message in list
char	MSG_SEMAPHORE	Semaphore number if reply is expected

6.4.6 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.

7 Bibliography

1. Modern Operating Systems. Andrew Tannebaum
2. The MCX11 Real Time Executive. Motorola
3. The ASSIST09 Monitor, Motorola
4. The μ/COS Real Time Kernel, Jean Lebrosse
5. Specifications for the AA16107-LY, Agena Industries Co.

8 Appendix 1. System calls available under RInOS

These system calls are at the level of assembler language. The equivalent C function is also given. All system calls are issued by loading the registers with the specified values and raising a software interrupt. For example:

```
ldx #23  Sleep for 230 ms
swi
fcb OSSleep
```

or in C

```
sleep(23);  /* sleep for 230 ms */
```

Any values returned by the system calls will be found in the specified registers The following are used to denote the 6809 registers:

```
X : X register
Y :  Y register
A : A register
B :  B register
CCR :  Condition Code register
```

Function # 1
OSSleep
Description:

Put a task to sleep for a specified no of clock ticks If the tick count is zero then the task sleeps indefinitely or until woken by another task

Called with:

X = Number of clock ticks to sleep

Returned with:

All registers unchanged

Equivalent C function: `void sleep(int nticks)`

Function # 2
OSWake
Description

Wake up a sleeping task.

Called with:
A = pid of sleeping task
Returned with:
Nothing

Equivalent C function: `void wake(int pid)`

Function # 3
OSSetPriority
Description

Change the priority of a specified task

On entry:
A = new task priority
B = pid of task (0 = current task)
On return:
A contains pid

Equivalent C function: `int set_priority(int priority,int pid)`

Function # 4
OSSemCreate
Description

Create a new semaphore

Called with:
A = Type: MUTEX, COUNTING, EVENT, SINGLE EVENT
B = Initial value of semaphore
Returned with:
A = semaphore number

Equivalent C function: `void sem_create(int sem_type, int init_value)`

Function # 5
OSFreeSem
Description

Release a previously allocated semaphore

Called with:
A = Semaphore number

Returned with:
Nothing

Equivalent C function: `void free_sem(int sem_num)`

Function # 6
OSDownSem
Description

Perform DOWN operation on semaphore. If semaphore blocks add calling task to list and put task to sleep

Called with:
A = semaphore no
Returned with:
A = value of semaphore

Equivalent C function: `int down_sem(int sem_num)`

Function # 7
OSUpSem
Description

Perform UP operation on semaphore. If semaphore is zero and a task or tasks are waiting on it, wake the task:

(a) with highest priority if the semaphore is a MUTEX type

(b) which is first in the linked semaphore list if the semaphore is a COUNTING type

(c) all if the semaphore is an EVENT type An EVENT semaphore will be freed if bit 7 is set in the semaphore type

Called with:
A = semaphore no Returned with:
Nothing

Equivalent C function `int up_sem(int sem_num)`

Function # 8
OSResetSem
Description

Reset an EVENT semaphore. This must be used with extreme caution.

On entry

A = semaphore number

On exit

The carry bit will be set in
the CCR if an error has occurred

Equivalent C function: `int reset_sem(int sem_num)`

Function # 9
OSSend
Description

Send a message to the mailbox of another task

Called with:

A = Task number of the receiver
X = Message address

Returned with:

All registers unchanged

Equivalent C function: `void send(int pid, void * message)`

Function # 10
OSReply
Description

Send a message and wait for a reply. This function is the same as the OSSend. function except that it puts the sender into a WAIT state until the task receiving the message signals the semaphore to indicate that it is finished.

A = Task number of the receiver
X = Message address

Returned with:

All registers are unchanged

Equivalent C function: `void reply(int pid, void * message)`

Function # 11
OSReceive
Description

Receive a message from another task. Either a specific task, or all tasks can be selected.

Called with:

A = pid of the preferred task
or 0 if any message will do

Returned with:

X = Address of the received message
All other registers unchanged

Equivalent C function: `void* receive(int pid)`

Function # 12
OSSignal
Description

Send a numbered signal to all tasks waiting for this signal. Both sender and receiver must agree on meaning of the signal. An optional pointer to any structure can be passed. This is used so that diverse tasks can agree on a set of common parameters

On entry

A = signal number
X = optional pointer to signal parameters

Returned with

Nothing

Equivalent C function: `int signal(int sig_num, int* params)`

Function # 13
OSWaitSignal
Description

Wait for a numbered signal from another task. Again both sender and receiver must agree on the meaning of the signal and the data block

On entry

A = the number of the signal to wait for

Returned with

X = pointer to signal data block

Equivalent C function: `void* wait_signal(int sig_num)`

Function # 14
OSInstallDriver
Description

Install or replace a device driver with a new one

On entry:

A = device # of driver to replace
X = pointer to a structure containing
the new set of driver parameters with the
following structure

void* ISR	Interrupt service handler
void* DRIVER	Device driver address
void* HBASE.	Hardware base address
void* SCRATCH.	Device scratch data area
char INSTALLED.	Specifies whether a device is installed (ff) or not(0)

Returns :

Nothing

Equivalent C function:
`void install_driver(int device_num, struct* params)`

Function # 15
OSStart
Description

Start the kernel. The kernel can be started either from the monitor or by the use of this function call from a user task. The function takes no parameters and does not return.

Equivalent C function: `void start(void)`

Function # 16
OSInstall
Description

Create a task and allocate memory for it. This function is used retrospectively to

(1) Create a tcb

(2) Allocate memory for a block of code that has been loaded at an absolute address and hence has bypassed the usual loading sequence

Called with

X = Pointer to a creation block with the following structure

char	PSEG	Page register
void*	CSEG	Start of module
void*	SSEG	Stack pointer
void*	CSTART	Code entry point
void*	ENVBLK	Address of environment (Should be NULL)
void*	TPRIO	Priority of task
char	TPID	PID of process (will be filled in by call)
int	TMEM	Size of memory required

Returned with

A = PID of new task

Equivalent C function: `int install(struct * params)`

Function # 17
OSCreate
Description

Create a task and put it into a runnable state

On entry A = Task priority
X = Pointer to task creation structure (See OSInstall)

Returned with

X = Pointer to new task TCB

Equivalent C function:

```
void* create(int priority,struct* params)
```

Function # 18
OSCAllocMem
Description

Allocate a block of shared memory. Each block is 32 bytes in size.

On entry
A = PID of task requesting memory
X = Size of memory requested
Returned with
X = Pointer to allocated memory block

Equivalent C function: `void* calloc_mem(int pid, int mem_size)`

Function # 19
OSCFreeMem
Description

Free blocks of shared memory previously allocated using OSCAllocMem

On entry
D = Number of blocks to free
X = Address of first block
Returned with
Nothing

Equivalent C function: `void cfree_mem(int size, void* addr)`

Function # 20
OSPAllocMem
Description

Allocate a block of shared memory. Each block is 32 bytes in size.

On entry
A = PID of task requesting memory
X = Size of memory requested
Returned with
A = Page number of allocated memory
X = Pointer to allocated memory block
Y = Size of memory actually allocated

Equivalent C function:
`void* palloc_mem(int pid, int mem_size, int* page, int* alloc_size)`

Function # 21
OSPFreeMem
Description

Free blocks of paged memory previously allocated using OSPAllocMem

On entry
D = Number of blocks to free
X = Address of first block
Returned with
Nothing

Equivalent C function: `void pfree_mem(int size, void* addr)`

Function # 22
OSTaskInfo
Description

Obtain a pointer to the calling task's TCB

On entry
No registers used
Returned with
X points at the TCB of the task

Equivalent C function: `void* task_info(void)`

9 Appendix 2. Device drivers under RInOS

Default device drivers exist for the following hardware items:

ACIA1	device # 0	Byte Input/Output
ACIA2	device # 1	Byte Input/Output
PIA	device # 2	Byte Input/Output
DAC	device # 3	Word Output
ADC	device # 4	Word Input
TIMER	device # 5	

Driver requests

On error all drivers return the carry bit set in the CCR, otherwise this bit is cleared.

Request # 1
Read Channel 1
Description

Read a single byte or word from channel 1 of the device

Called with
A = 0
Returned with
B = byte input (Byte devices)
D = Word input (Word devices)

Request # 2
Write Channel 1
Description

Write a single byte or word to channel 1 of the device

Called with
A = 1
B = Data to write (Byte devices)
X = Data to write (Word devices)
Returned with
Nothing

Request # 3
Read a string from Channel 1
Description

Read a string from the device. For the ACIA driver, on receipt of a carriage return, a null character is appended to the string.

Called with
A = 2
X = Pointer to location
where string is to be placed
Returned with
Nothing

Request # 4
Write string to Channel 1
Description

Write a null terminated string to the device. For the ACIA, when the null character is encountered in the output stream, carriage return - line feed characters are substituted.

Called with
A = 3
X = pointer to first character
of the string to be sent
Returned with
Nothing

Request # 5
IOCTL
Description

This function allows the registers of the device to be manipulated directly

Called with
A = 4
B = Control byte
Returned with
A = status byte

Request # 6
Initialise device
Description

Before a device can be used it must be initialised. The default drivers for the ACIA2, ADC, DAC and PIA in mode 0 are initialised by the system during start up. The ACIA1 driver must be initialised by an application programme prior to use. If the PIA is to be used in any mode other than mode 0, it also must be initialised to this mode before use. The PIA modes are:

PIA_STD_MODE	Mode # 0	Configure A/B sides of the PIA using a bit mask in X
PIA_HNDSHK_MODE	Mode # $10	Configure for automatic handshaking
PIA_ICTP_DISPLAY_MODE	Mode # $20	Configure for use with the ICTP display card. Read switches and write to LED. Interrupt on push button
	Mode # $21	Configure for use with the ICTP display card. Write to LCD display. Interrupt on push button
PIA_COLOMBO_MODE	Mode # 30	Not yet implemented

Called with
A = 5
B = Mode (for PIA only)
X = Bit mask (for PIA mode 0 only)
Returned with
Nothing

Request # 7
Set Input lock
Description

Lock the input stream by the use of a semaphore. Any task requiring the use of the device should first call this function to check that the device is free, if it is not free the task will sleep pending release of the semaphore. Note that this works only on a voluntary basis as any task not using the semaphore

can make use of the stream and interfere with other tasks co-operating in the use of the semaphore.

Called with
A = 6
Returned with
Nothing

Request # 8
Release Input lock
Description

Release the semaphore to the input stream set by a previous call to Set Input Lock. The highest priority task waiting on the input semaphore will be woken.

Called with
A = 7
Returned with
Nothing

Request # 9 Set Output lock
Description

Set a semaphore lock on the output stream. Refer to Set Input lock request for further details

Called with
A = 8
Returned with
Nothing

Request # 10
Release Output lock
Description

Release a lock set on the output stream by a previous call to Set Output lock

Called with
A = 9
Returned with
Nothing

Request # 11
Read Channel 2
Description

Read a single byte or word from channel 2 of the device

Called with
A = 0
Returned with
B = byte input (Byte devices)
D = Word input (Word devices)

Request # 12
Write Channel 2
Description

Write a single byte or word to channel 1 of the device

Called with
A = 1
B = Data to write (Byte devices)
X = Data to write (Word devices)
Returned with
Nothing

X Windows Programming

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

Trieste, 7 Oct–1 Nov 1996

Ulrich Raich
CERN - European Organisation for Nuclear Research
P.S. Division
CH–1211 Geneva
Switzerland.

email: Ulrich.Raich@cern.ch

Abstract

These lecture notes are intended to give an insight into Graphical User Interface (GUI) Programming using the X-Windows system. It explains the different layers of X11 and gives a short introduction to XLib. Motif, the widget set supplied by the Open Software Foundation (OSF) is used to demonstrate building of more sophisticated GUIs. Even though only very few routines of the Motif libraries are explained these notes are sufficient to build a Motif program driving the ICTP Colombo Board which is proposed as an exercise.

1

Introduction to X-Windows

X started its life in 1984 at the Massachusetts Institute of Technology (MIT) with the project Athena. At MIT several hundreds of workstations were scattered on the campus. They were intended for the use by students and were rather heterogenous (severeal different manufacturers, different operating systems). On the other hand most of them had:

- a powerful 32 bit CPU
- large address space
- a high resolution bitmapped display
- a mouse on some of them
- a network connection.

The idea was therefore to provide a window system allowing to write vendor independent applications, that could run on any of these stations. In addition, it should be possible to access applications on any of the workstations from any other workstation using the network. Of course performance was another keyword in the design.

Since the lecture time for X-Windows programming is fairly limited (there are some 10 books of 700 pages each explaining the X-Windows system!!!) we prepared this little booklet, which should contain an explication of some basic features of the system and also all the calls you will need for the exercises.

In the course of the lectures you will build a little X-Windows application simulating the Colombo board on the screen and interact with it.

2

The Client-Server Model

In order to write device independent applications, the details of device access must be hidden in some sort of driver. In X this is a program called the **X-Server**. It provides all the basic windowing mechanisms by handling connections from X-applications, demultiplexing graphics requests and multiplexing input from keyboard, mouse etc. back to the application. This program is usually provided by the hardware vendor.

An application connects to the X-Server through an interprocess communication (IPC)path either through shared memory or through a network protocol like TCP. Such a IPC path is called the **X-Client**. Since most applications open only a single connection we often call the application itself the X-Client. However: an application having several IPC paths open is considered as several clients.

The communication protocol between an X-Client and an X-Server is called the **X-Protocol.**

One of the main design objectives of this protocol was to minimize the network traffic, because the network must be considered the slowest system component. Therefore an asynchronous protocol has been chosen. In order to bring windows up on the screen the application simply sends off requests without waiting for an acknowledgement. This can be done because of the reliability of the underlying network protocol. The application also does not poll for events like key presses and mouse button presses. It registers interest in certain events with the X-Server, which will then send only relevant events back to the application. Both the output requests and input events are buffered.

The X-Protocol is the fundamental layer onto which other tools can be build. It is user interface policy free. This means that only rectangular windows are handled but no buttons, menus and the like. These so-called widgets are implemented in a toolkit sitting on top of the protocol.

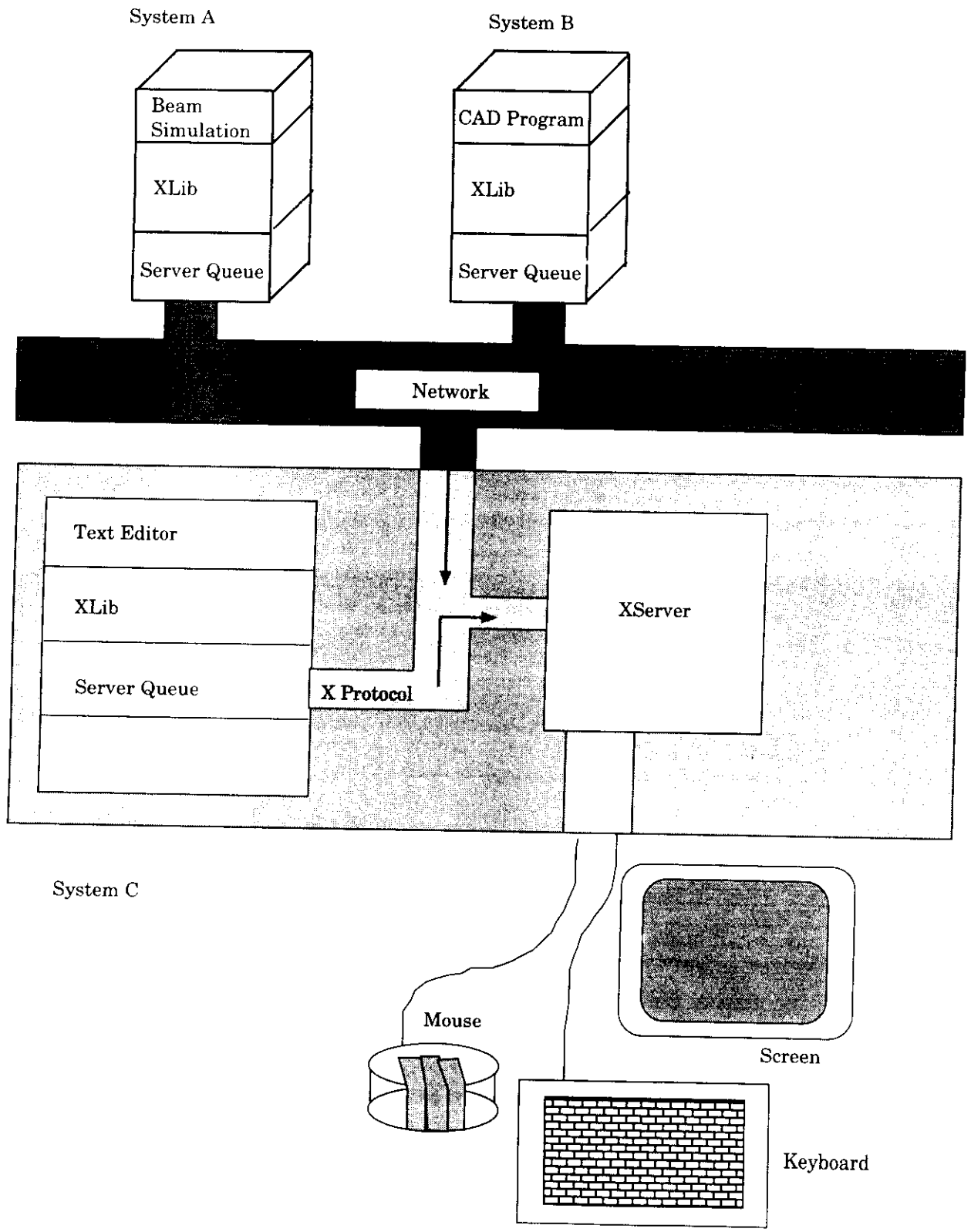
The XLib contains routines that allow access to the X-Protocol. It provides the following functionality:

- **display management** (open, close displays)
- **window management** (create/destroy windows and change their visual aspect)
- **two dimensional graphics** (draw lines, circles, rectangles, text)
- **color management** (color map and its access routines)
- **event management** (registration of interest in events and event reception)

This gives us an overview over the next few chapters.

In fig 2-1 we see 3 X-Clients running on 3 different machines (A,B,C) and communicating with a single X-Server (on system C). For the clients on A and B the X-Protocol runs over the network, while for the text editor a shared memory IPC path is used.

Figure 2-1 The Client-Server Model



3

Display Management

In order to create windows on the screen and to receive events a connection to the **display** must be established. In X terminology the display consists of :

- one or more screens
- a single keyboard
- an (optional) pointing device
- the X-Server.

This connection can be built through the XLib call:

Display **XOpenDisplay**(display_name)

char *display_name

If display_name = NULL the display_name defaults to the value stored in the environment variable DISPLAY. If you want to open the server on your neighbor's workstation, he will first have to allow you access to it (xhost name_of_your_station), then you may define display_name as his_station:server_number.screen_number (usually 0.0).

The return value from this call must be saved, because it will be passed into all subsequent calls. In case of an error a NULL display is returned.

There are several Macros giving information about screen properties like:

- **DisplayHeight**(display, screen_number)
  Display display;
  int screen_number;
- **DisplayWidth**(display, screen_number)

giving the width and the height (in pixels) of the screen.

4

Window Hierarchies

Once the Client-Server connection is established, we can generate our first windows:

```
Window = XCreateSimpleWindow(display,parent,x,y,width,height,
                             border_width,border_color,background_color);
Display         *display
Window          parent;
int             x,y;        /* position with respect to the upper left corner */
                            /* of the parent window */
unsigned int    width, height, border_width;
unsigned long   border_color;
unsigned long   background_color;
```

x,y,width,height and border_width do not need any explanation. background_color specifies the background color. Since colors will only be explained later we will put this to:

unsigned long **WhitePixel**(display,DefaultScreen(display))

where WhitePixel and DefaultScreen are Macros returning the pixel value for "white" and the default screen number respectively. The border_color parameter specifies the color for the window border and is set to:

unsigned long **BlackPixel**(display,DefaultScreen(display)).

The parent parameter will need some more explanation: All windows are inserted into a window hierarchy, where each window has a parent window. The great-grandfather of all windows is the root window who's id can be obtained by:

Window **RootWindow**(display,DefaultScreen(display)).

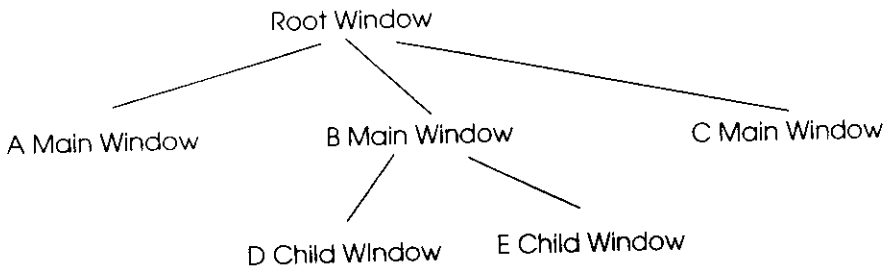
The root window

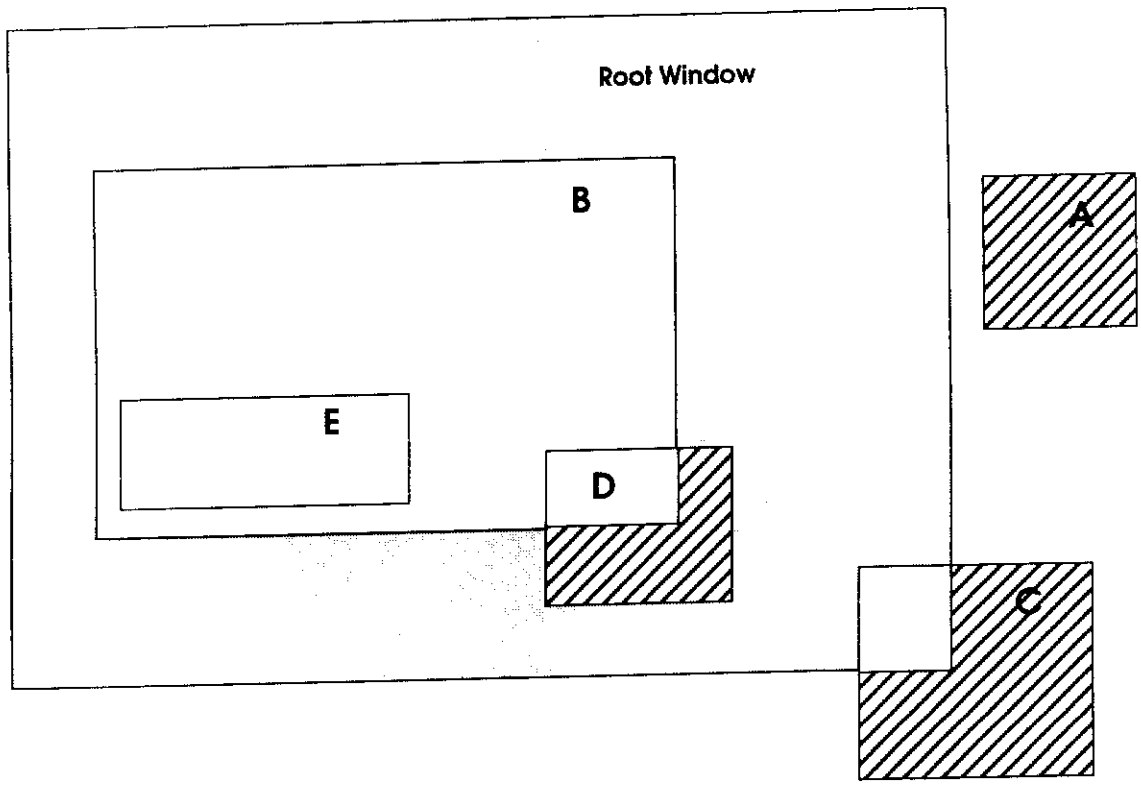
- covers the screen completely
- cannot be moved or resized
- is the parent of all other windows
- has all window attributes like background color, patterns etc.

You can draw onto the root window as on to all other windows

The return value from XCreateSimpleWindow is used to build up the window hierarchy. In fig. 4-1 a complete hierarchy is shown. Since all windows are clipped to the boundaries of their parents some of the windows may be completely invisible.

Figure 4-1 Window Hierarchies





Legend:

clipped

root window

visible part of window

After the XCreateSimpleWindow all the data structures needed for the management of the window will be created; however the window will still not be visible.

XMapWindow (display,window_id)
Display display;
Window window_id;

will map the window and all of its subwindows, for which the XMapWindow routine has been called. Once the window is mapped, there are several XLib calls to change its layout:

- **XMoveWindow**(display,window_id,x_offset,y_offset)
- **XResizeWindow**(display,window_id,width,height)
- **XMoveResizeWindow**(display,window_id,x_offset,y_offset,width,height)
- **XSetWindowBorderWidth**(display,window_id,border_width)
- **XSetWindowBackground**(display,window_id,background_pixel)
- **XChangeWindowAttributes**(display,window_id,value_mask,attributes)

and many more.

The last call allows to change any of the window attributes in a single call. "attributes" is a XSetWindowAttributes structure, having a certain number of fields. The value_mask tells the system, which of the attribute fields are to be taken into account. Only these values will be changed. It is a bitwise inclusive OR of the valid attribute mask bits (see fig. 4-2).

Using this mechanism windows can also be created with:

XCreateWindow(display,parent,x,y,width,height,border_width,depth,class,visual, valuemask,attributes)

If in the situation of fig 4.1 we would call
XUnmapWindow(display,B_main_window)
the window B and all of its subwindows (D and E) would disappear.

Figure 4-2 Structure and Value Mask

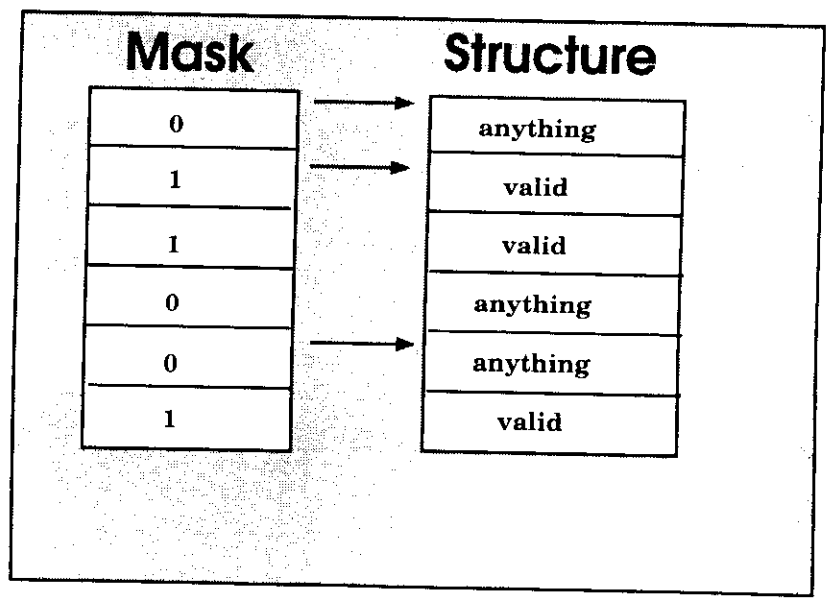


Fig 4-3 shows the results of such a Map call for a single main window.

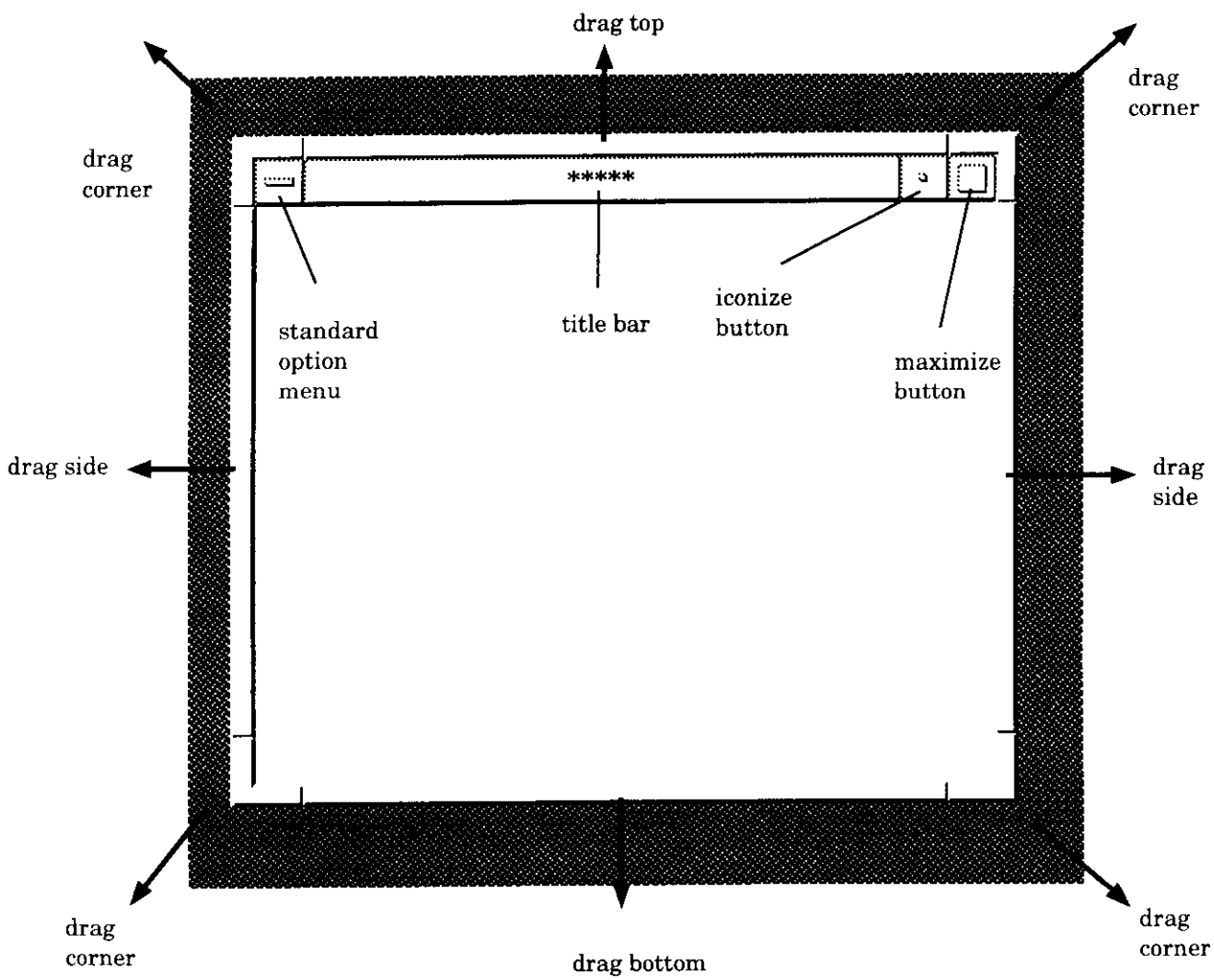
It has been explained before, that a window simply consists of a rectangle. Here on the contrary many more items like the three buttons on top of the window, the stars, the triangles on each corner etc. can be seen. The layout and the functionality depend on the look and feel (the policy) of the window system. It is another X-Client, the **window manager**, which is responsible for the decoration of the main window (child of root window). It allows to change the stacking order of windows, displace and resize windows, iconize them and even killing them (and the application).

The XLib provides calls to communicate easily with the window manager in order to modify the decorations like

XStoreName(display,window_id,title_bar_text);

This call will communicate the title to be put into the title bar to the window manager. The communication is done through the Inter Client Communication Convention (ICCCM, M=manual) using so called window properties, which are data that can be attached to windows. We will not be able to go into any detail because of lack of time.

Figure 4-3 A Main Window



5

Drawing, the Graphics Context

Let us consider the simplest possible drawing instruction: drawing a line between 2 points. This is done with the call

XDrawLine(display,drawable,graphics_context,x1,y1,x2,y2)
Display display;
GC graphics_context;
int x1,y1,x2,y2;

The meaning of all parameters except "drawable" and "graphics_context" should be obvious. The "drawable" tells the system where to draw. In fact there are 2 possibilities. Either we draw into a window on the screen or into a window simulated in memory, called a pixmap (Details on pixmaps are found in chapter 6). Remember that the root window is treated like any other window, so it is possible to generate background pictures by drawing onto the root window.

Coming back to our draw line primitive: When drawing the line, several questions remain open:

- what is the line width?
- what color?
- straight line or dashed, dotted ... and what are the distances between dashes?
- how to join lines.

and there are many more **drawing attributes**.

Since it is the X-Client who generates these graphic requests and it is the X-Server who executes them, all attributes must be sent to the server. This could be done on a per primitive basis, however network traffic would be strongly increased and the performance would suffer. For this reason graphics contexts containing all these attributes can be prepared on the server. In the drawing call the identifier of the graphics context resident on the server is specified.

GC **XCreateGC**(display,drawable,values,value_mask)
Display display;
Drawable drawable;
XGCValues values;
unsigned long value_mask;

creates a graphics context for us.

The value structure of type XGCValues has over 20 entries. In the following table some of the entries and their corresponding value_mask bit names are given.

Entry	value_mask bit	usage and possible values
values.line_width	GCLineWidth	
value.line_style	GCLineStyle	LineSolid — draw full line LineDoubleDash — odd lines fill differently from even lines LineOnOffDash — only even dashes are drawn
values.cap_style	GCCapStyle	How to draw the end point: CapButt — line square at the end point CapNotLast — as CapButt, but the last point is not drawn CapProjecting — as CapButt, but the line is longer half the projection CapRound — round end points
values.join_style	GCJoinStyle	how to join fat lines JoinMiter — outer edges extend to meet at an angle JoinBevel — corner is cut off JoinRound — round off edges
values.fill_style	GCFillStyle	FillSolid — uses foreground color for filling FillTiled — uses a colored tile pattern FillStippled — same as FillSolid but uses a stipple pattern bitmap as mask FillOpaqueStippled — same as FillTiled but uses a stipple pattern as mask in addition
values.function	GCFunction	logical operation for drawing possible values see later
values.foreground	GCForeground	foreground color
values.background	GCBackground	guess what!
values.tile	GCTile	tile pixmap
values.stipple	GCStipple	stipple bitmap
values.clip_mask	GCClipMask	clip mask
values.ts_x_origin	GCTileStipXOrigin	shifting the tile of stipple pattern origins same for y
values.font	GCFont	font for text drawing

In order to create a graphics context that allows drawing of a dashed line of width 4 the following code segment would do the job:

```
XGCValues       values;
unsigned long   value_mask;
GC              graphics_context;

value_mask = GCLineStyle | GCLineWidth;   /* setup the value mask */
values.line_style = LineOnOffDash;        /* define the fields indicated in mask*/
values.line_width = 4;
graphics_context = XCreateGC(display,main_window,values,value_mask);
```

All other GC values will be defaulted.

Another way is to generate a default GC using
DefaultGC(display,screen_number)
and then use
XChangeGC(display,graphics_context,value,value_mask)
to do the necessary changes.

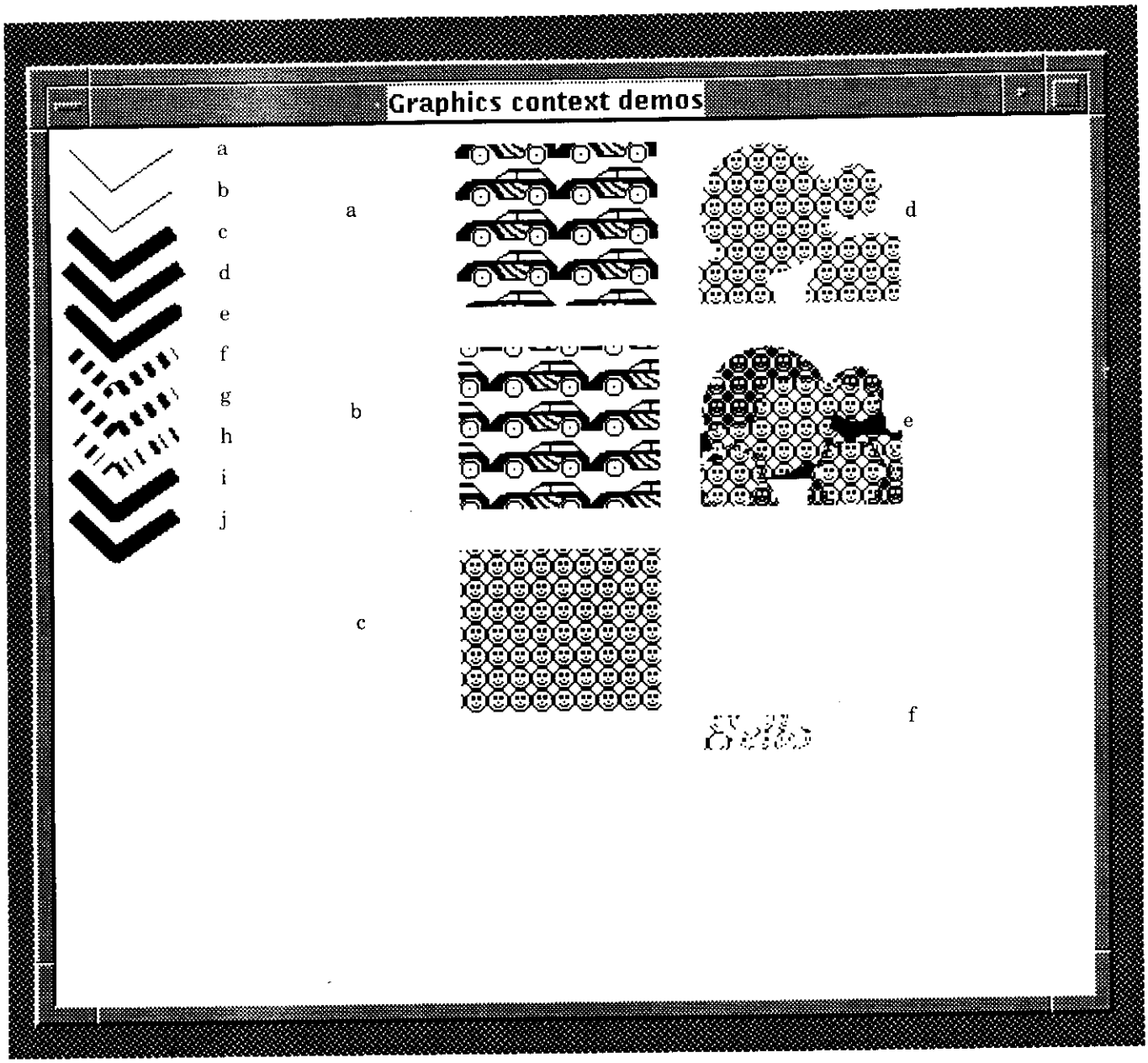
There are also lots of convenience functions changing a single entry in the value structure:

- **XSetForeground**(display,graphics_context,foreground)
- **XSetBackground**(display,graphics_context,background)
- **XSetLineAttributes**(display,graphics_context,line_width,line_style, cap_style,join_style)
- **XSetDashes**(display,,graphics_context,dsh_offset,dash_list,n)
- **XSetFillStyle**(display,,graphics_context,fill_style)
- **XSetTile**(display,,graphics_context,tile)
- **XSetStipple**(display,,graphics_context,stipple)
- **XSetClipMask**(display,graphisc_context,clip_mask)
- **XSetFont**(display,graphics_context,font)

and many more. Fig 5-1 shows the result of these graphics context manipulations.

Last but not least there is an entry value.function, which sets the binary function that is applied to the existing pixel value when drawing onto the screen (src is the pixels to be drawn newly, dest is the actual pixel value)

- GXClear 0
- GXand src and dest
- GXandReverse src and (not dest)
- GXcopy src (this is the default of course!)
- GXnoop dest
- GXxor src xor dest
- GXnor (not src) and (not dest)
- GXequiv (not src) xor dst
- GXinvert not dst
- GXorReverse src or (not dest)
- GXcopyInverted not src
- GXorInverted (not src) or dst
- GXnand (not src) or (not dest)
- GXset 1



a. default line

b. CapNotLast, last point is not drawn (difficult to see!)

c. line width set to 8, CapButt

d. CapProjecting

e. CapRound

f. LineDoubleDash

g. LineOnOffDash (would need fill patterns to see the difference)

h. different dash lengths

i. JoinBevel

j. JoinRound

a. FillTiled

b. Changed the tile origin

c. fill stippled

d. use a clip mask

e. use XOR graphic function

f. filled text

6

Bitmaps and Pixmaps

In the previous chapter we were talking about pixmaps as drawables for drawing primitives. Therefore the questions: What exactly is a pixmap? What is the difference between a bitmap and a pixmap? and how can we generate pixmaps?

As already explained before a pixmap is a sort of a simulated window in memory. As long as we work on a black and white workstation we need 1 bit for each pixel to be displayed. An array, describing such a pixelplane is called a **bitmap**. Once we use a color display several bitplanes are needed depending on the number of colors, that can be displayed. This collection of bitmaps with a certain **depth** is called a **pixmap**.

An empty pixmap can be allocated with the call:

Pixmap **XCreatePixmap**(display,drawable,width,height,depth)

```
Display     display;
Drawable    drawable; /* needed to determine which screen the pixmap is stored on */
unsigned int width,height;
unsigned int depth;
```

The pixmap will be stored on the X-Server, which is the reason for the drawable parameter. Just specify the id of your main window. Once you allocate the pixmap you can use it as drawable in any of the drawing primitives. In order to visualize your pixmap you must copy its contents onto a window:

XCopyArea(display,source_drawable,dest_drawable,gc,src_x,src_y,
copy_width,copy_height,dest_x,dest_y);

If you have a bitmap which you want to convert to a pixmap or simply visualize on a color display you use:

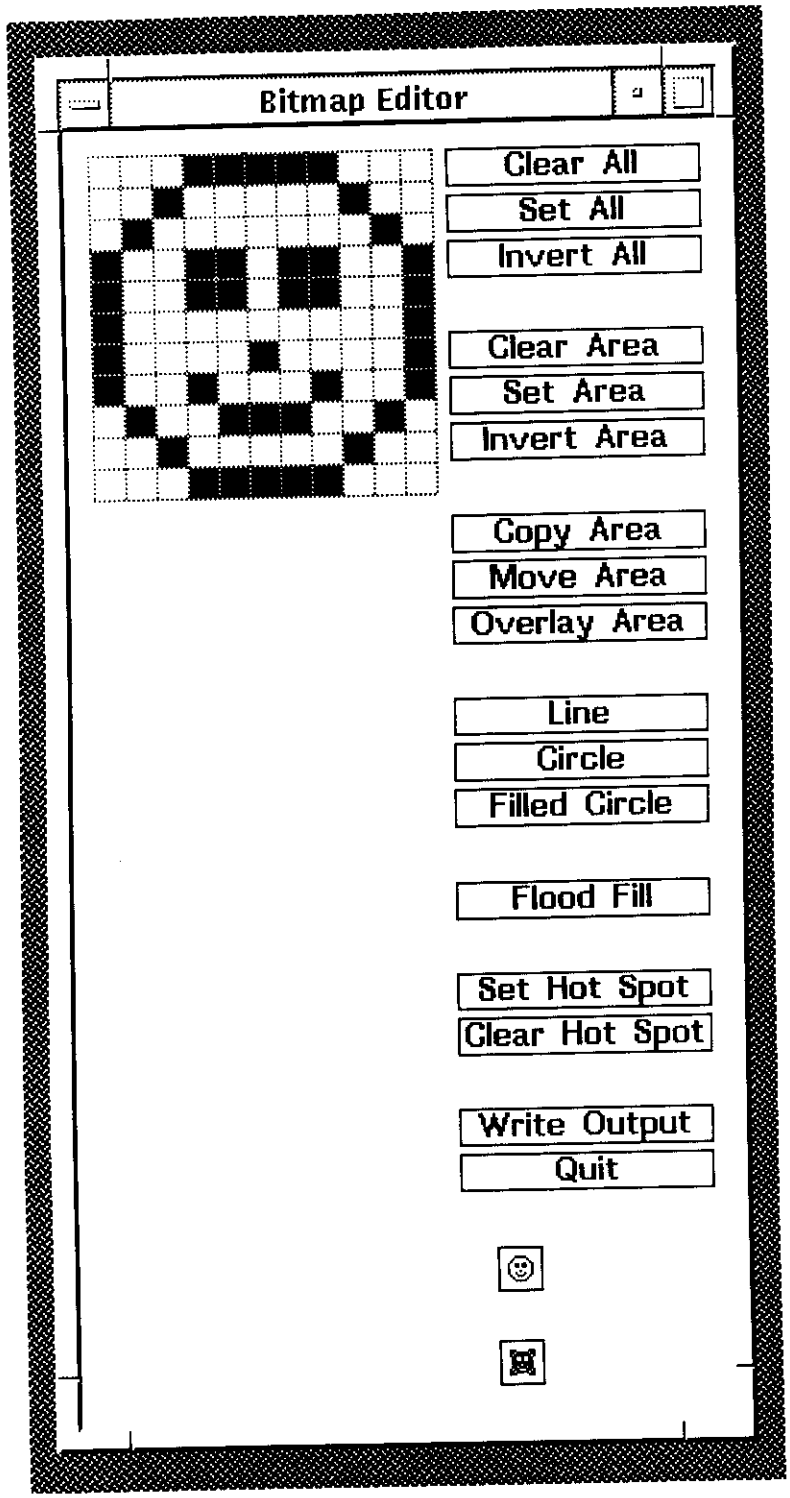
XCopyPlane(display,source_drawable,dest_drawable,gc,src_x,src_y,
copy_width,copy_height,dest_x,dest_y,plane);

with plane =1 (bitmap).

Of course it might be rather difficult to build up bitmaps using only drawing primitives. For this reason X provides a utility, the bitmap editor.

The command "**bitmap**" will bring up the application shown in fig. 6-1

Figure 6-1



The result of the editor is a C source file which can be included into you application:

```
#define smiley_width 11
#define smiley_height 11
static char smiley_bits[] = {
   0xf8, 0x00, 0x04, 0x01, 0x02, 0x02,
   0xd9, 0x04, 0xd9, 0x04, 0x01, 0x04,
   0x21, 0x04, 0x89, 0x04, 0x72, 0x02,
   0x04, 0x01, 0xf8, 0x00};
```

This code can be used to create a pixmap:

Pixmap **XCreatePixmapFromBitmapData**(display,drawable,smiley_bits,
smiley_width,smiley_height,foreground,background,
DefaultDepth(display,screen_number));

Here the macro DefaultDepth is used to find the number of bitplanes used for the display. The same result can be obtained by reading in the bitmap file directly.

int **XReadBitmapFile**(display,drawable,bitmap_file_name,&width,&height,
&bitmap,&hot_x,&hot_y);

(hot_x, hot_y give the coordinates of the "hot spot" used for cursors). Now the bitmap can be converted to a pixmap with the XCopyPlane call.

There is also a freely distributable library and a pixmap editor which can be used to generate pixmaps (in color) directly. (Try "pixmap" on your machine!)

Pixmaps are used for cursors, tiles, stipples, icons etc. They can also be used to restore pictures which have been destroyed by overlapping windows (see chapter on events).

7

Drawing Primitives

X-Windows is NOT a graphics system! This can be easily seen by the limited number of graphics primitives and by their simplicity:

There are a few functions to clear out an area to be drawn in

- **XClearArea**(display,window_id,x,y,width,height,exposures)
- **XClearWindow**(display,window_id)
- **XFillRectangle**(display,drawable,graphics_context,x,y,width,height)

While most graphics primitives work on a drawable, XClearWindow and XClearArea work only on windows.

Here are the primitives which actually draw graphic objects:

XDrawPoint(display,drawable,x,y)
XDrawPoints(display,drawable,points,npoints,mode)
where points is an array of type

```
typedef struct {
  short       x,y;
} XPoint;
```

npoints, the number of XPoint entries in the array "points"
and mode = CoordModeOrigin (x,y is given in absolute pixel coordinates)
or mode = CoordModePrevious (x,y are the relative distances to the last point)

XDrawLine(display,drawable,gc,x1,y1,x2,y2)
XDrawLines(display,drawable,gc,points,npoints,mode)
XDrawRectangle(display,drawable,gc,x,y,width,height);
XDrawRectangles(display,drawable,gc,rectangles,nrectangles)
where rectangles is an array of type

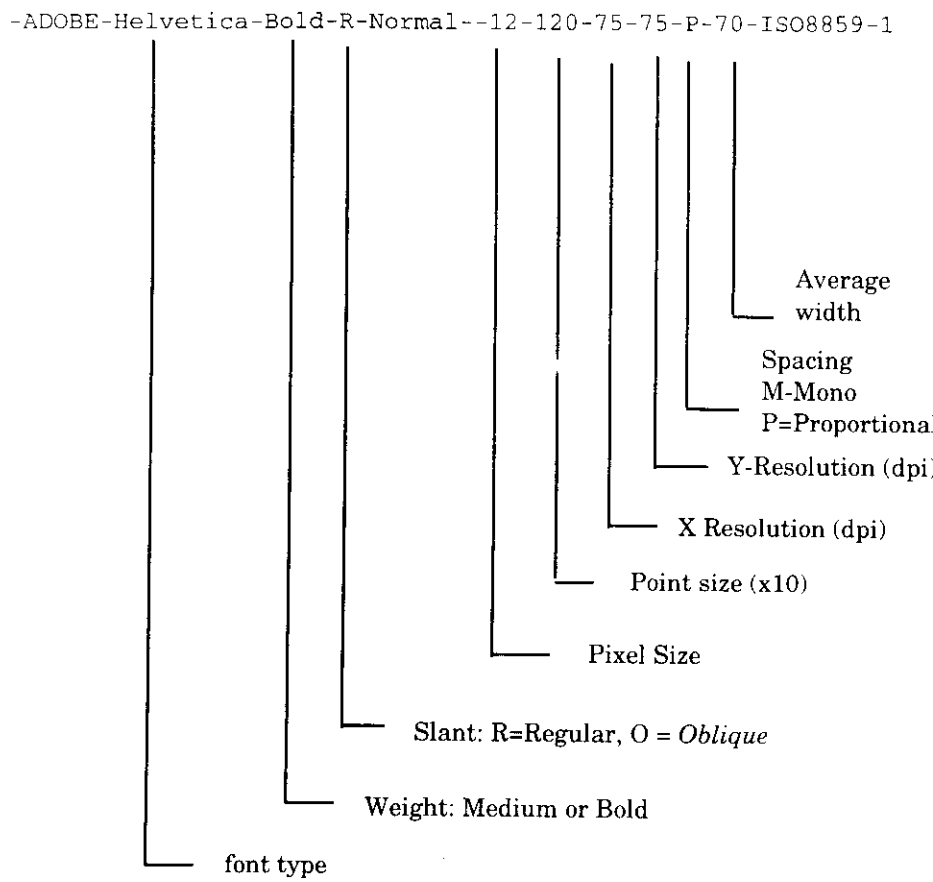
```
typedef struct {
  short                x,y;
  unsigned short       width,height;
}XRectangle;
```

XFillRectangle(display,drawable,gc,x,y,width,height);
XFillRectangles(display,drawable,gc,rectangles,nrectangles)

and there are some more for drawing arcs, segments etc.

For text drawing lots of different fonts are available. The command **xlsfont** prints the names of all available fonts. If you want to know how the font looks like, try **xfd** (x font display).

The font names are standardized as follows:



First the font must be loaded with

Font XLoadFont(display,font_name)

then the font must be specified in the graphics context and last but not least we can draw our text using **XDrawString**(display,drawable,gc,x,y,string,length).

It is also possible to fill an array of text items:

```
typedef struct {
  char      *chars;
  int       nchars;
  int       delta;      /* distance between strings, is added to horizontal origin */
  Font      font;
} XTextItem.
```

and use **XDrawText**(display,drawable,gc,x,y,item_array,nitems)
which allows drawing of multiple font text strings.

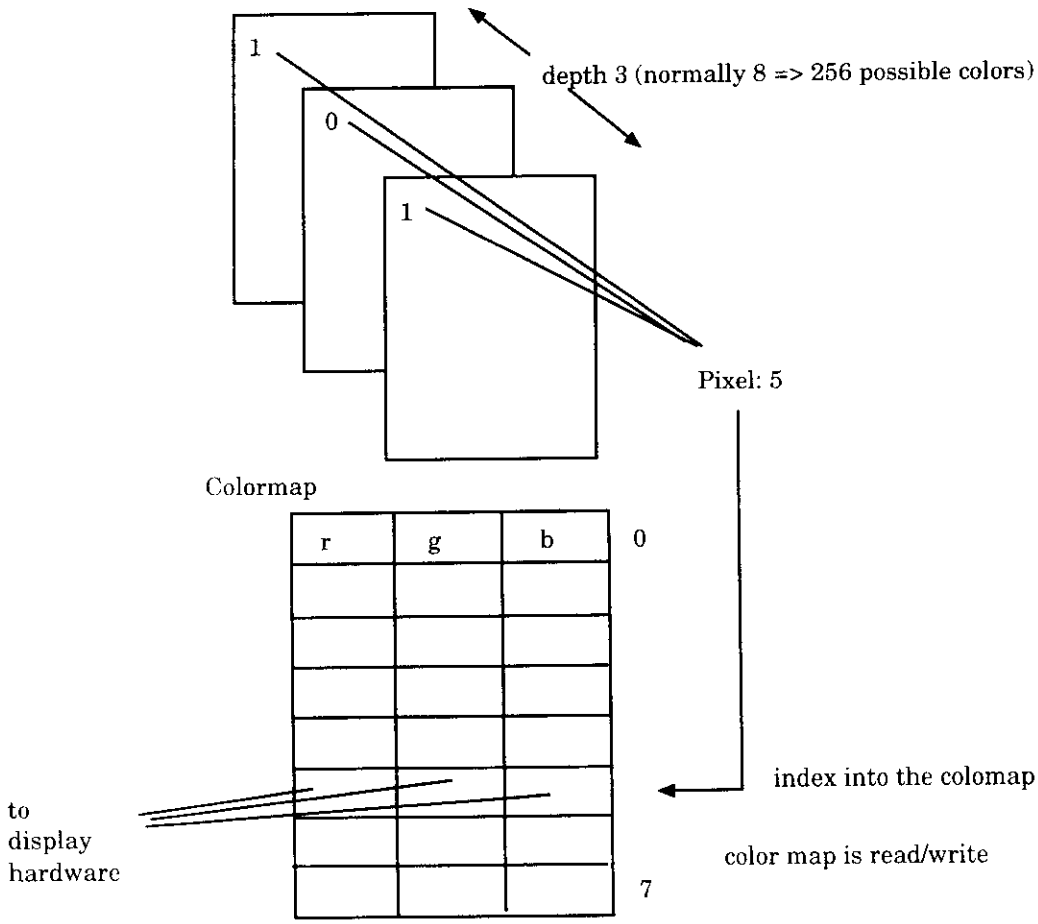
8

The Color Model

The hardware for color displays varies very widely depending on the needed graphics performance of the system. Since X is supposed to be hardware independent, there must be a common color model that can be converted for the different devices. X knows of the following hardware types, referred to as **visuals**:

- **Pseudo color**
  This is the most common type of device (and probably the one you have in your PC!) The image is described through a pixel array, where each pixel is interpreted as an index into a color table, containing the r,g,b values sent to the screen.

Figure 8-1 The Pseudo Color Display



- **Static color**
  Like pseudo color, except that the values in the color map are read only
- **Direct color**
  The pixel is composed of 3 bit fields, each of which is used as an index into one of 3 R,G,B color maps. The values in the three color maps can be changed
- **True color**
  Like direct color, but uses three linear readonly color maps
- **Grey scale**
  Like pseudo color. There are still 3 color maps but only 1 of them is used
- **Static grey**
  Like grey scale, but color map is readonly and linear

An X application can install its own color map, but it will then disturb other applications running on the same screen, because their colors will be wrong. The color maps are switched depending on input focus.

So normally a single default color map is used. We can get an identifier to it by

Colormap **DefaultColormap**(display,screen_number)

This colormap (when the visual depth is 8) usually contails 6*6*6 preprogrammed colors and 40 freely programmable color cells.

In order to allocate color cells in the color map there a 2 possible methods:

- allocating readonly color cells
  These cells are shared between applications and are allocated by color name
  XAllocNamedColor(display,DefaultColormap(display,screen_number,
  "dark olive green",&closest_color,&exact_color);
- allocating read/write color cells:
  XAllocColorCells(display,DefaultColormap(display,screen_number,
  contig_flag, &plane_masks,nplanes,&pixels,npixels);
  We can then use
  XStoreColor(display,DefaultColormap(display,screen_number,my_color) to store a color into the default colormap.

The structure

```
typedef struct {
  unsigned long pixel;
  unsigned short red,green,blue;
  char flags;                    /*DoRed | DoGreen | DoBlue */
  char pad;
} XColor;
```

describes a color. The red,green,blue values are always in the range of 0-65535 and are scaled to the number of bits actually in use by the display hardware. Flags allow to use only the red,blue,or green component (or any combination thereof).

Most graphics routines ask for a pixel value. When allocating readonly color cells, a color structure is returned (closest_color) and the pixel entry in this structure (closest_color.pixel) can be used. In XAllocColorCells, the pixel values are returned directly.

Fig 8-2 shows part of the rgb database file:

Figure 8-2 The RGB database

```
255 250 250             snow
248 248 255             ghost white
248 248 255             GhostWhite
245 245 245             white smoke
245 245 245             WhiteSmoke
220 220 220             gainsboro
255 250 240             floral white
255 250 240             FloralWhite
253 245 230             old lace
253 245 230             OldLace
250 240 230             linen
250 235 215             antique white
250 235 215             AntiqueWhite
255 239 213             papaya whip
255 239 213             PapayaWhip
255 235 205             blanched almond
255 235 205             BlanchedAlmond
255 228 196             bisque
255 218 185             peach puff
255 218 185             PeachPuff
255 222 173             navajo white
255 222 173             NavajoWhite
255 228 181             moccasin
255 248 220             cornsilk
255 255 240             ivory
255 250 205             lemon chiffon
255 250 205             LemonChiffon
255 245 238             seashell
240 255 240             honeydew
245 255 250             mint cream
245 255 250             MintCream
```

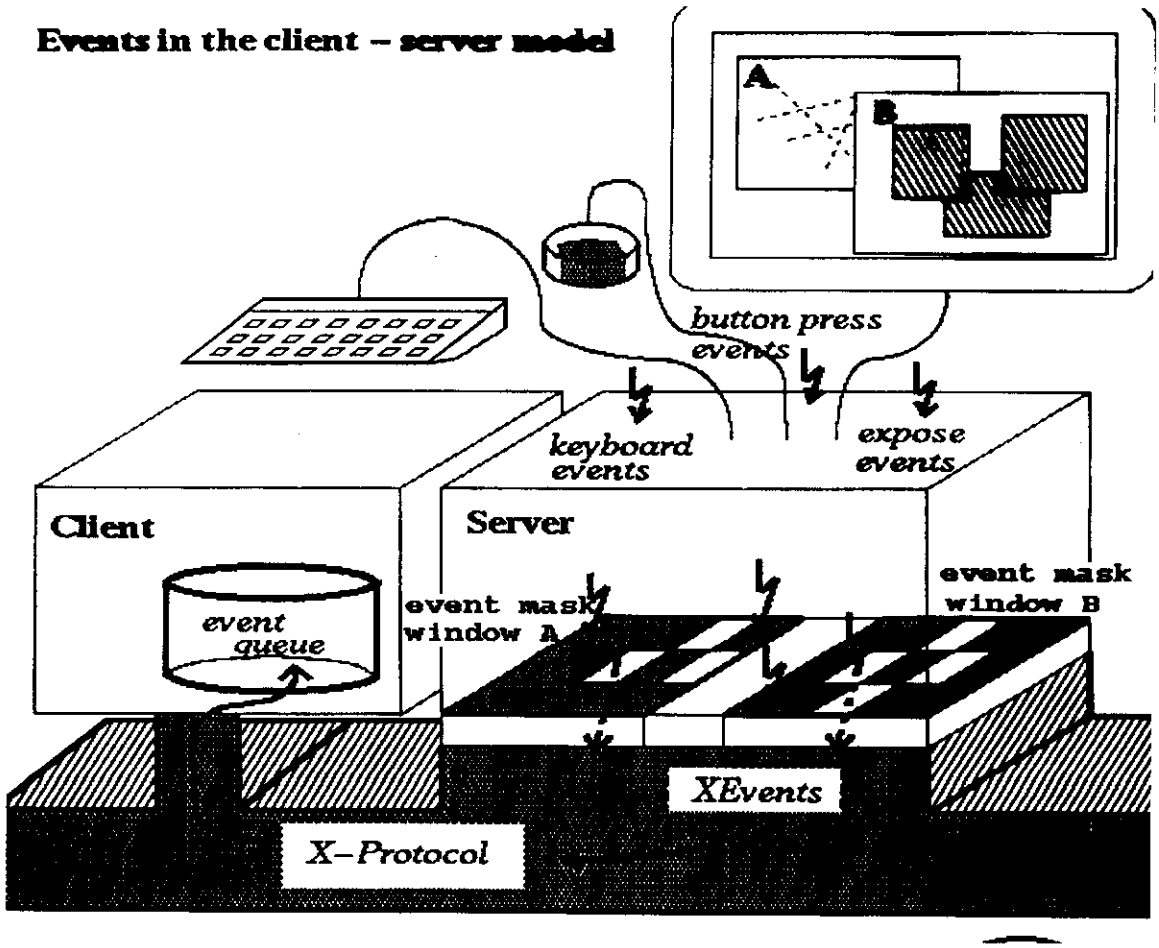
9

Event Handling

Once the client-server connection is opened the X-Client sends requests for bringing up windows, changing them, drawing things into them etc. but the X-Server can also inform the X-Client of certain events like exposure of a window, mapping of a window or user initiated, asynchronous events like mouse clicks or keyboard button presses.

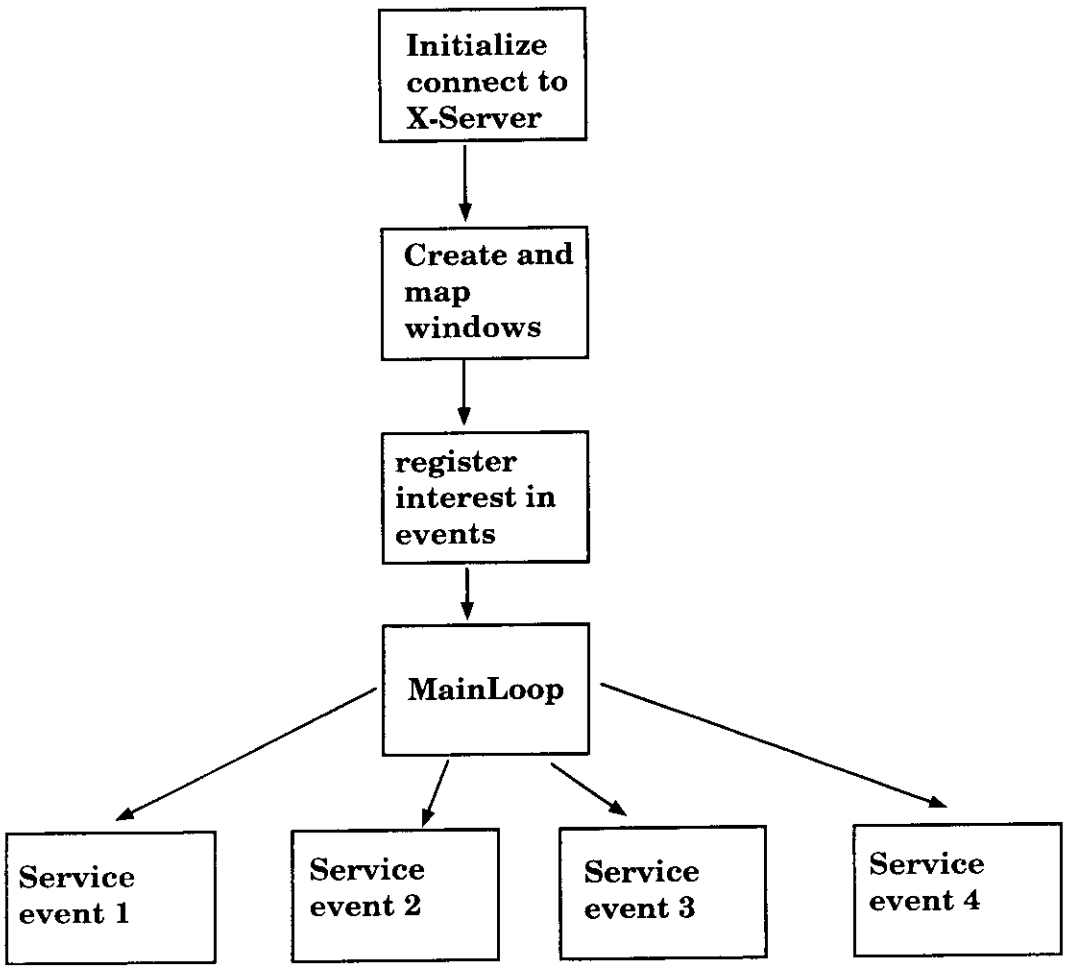
Due to the enormous amount of possible events (think of mouse movement only!) and the relatively small number of events the X-Client is actually interested in, it is much more efficient to filter the events on the server side. Before treating any events the X-Client must therefore register interest in a certain type of event on a per window basis.

Figure 9-1 The event chain



The general layout of an X-Client is therefore given by the diagram below:

Figure 9-2 Flow of control in an X-Client



While in the "usual" programming style the program asks for user input at the moment it is needed and convenient (the program controls the user!) in X-Windows programs the user can change the flow of control in any manner choosing functions provided by the program in a completely random manner.

There are 2 possible ways for the X-Client to register interest in events:

- at the moment of window creation we can set the entry "event_mask" and the corresponding bit "CWEVENTMASK" in the value mask to the event types we are interested in
- **XSelectInput**(display,window_id,event_mask)

If one of the selected events arrives at the XServer (say a mouse click) it sends this event into the XClients event buffer. There the main loop can pick it up and analyse it.

The call

XNextEvent(display,&event)

Display display
XEvent event

retrieves the next event from the event queue and blocks if no events are available.

The XNextEvent returns an **XEvent** structure of the following form:

```
typedef struct {
int type;
unsigned long     serial;
Bool              send_event;
Display           *display;
Window            window;
} XAnyEvent;

typedef union {
int               type;
XAnyEvent         xany;
XButtonEvent      xbutton; ... many more ...
XExposeEvent      xexpose; ... many more ...
XKeyEvent         xkey;
XMapEvent         xmap;    ... many more ....
} XEvent;
```

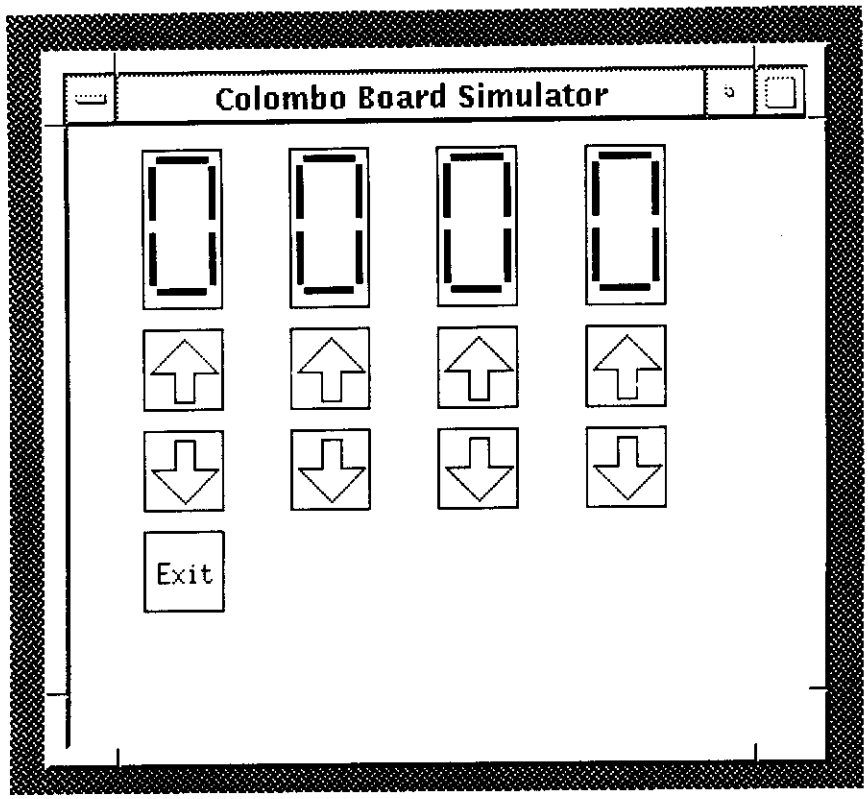
From the event.type we can find out which sort of event has happened. The following table gives a few examples. The event mask enabling reception of the event type and the symbol for the event type are given.

Event Mask	Event Type	Event Structure
KeyPressMask	KeyPress	XKeyPressedEvent
KeyReleaseMask	KeyRelease	XKeyReleasedEvent
ButtonPressMask	ButtonPress	XButtonPressedEvent
ButtonReleaseMask	ButtonRelease	XButtonReleasedEvent
PointerMotionMask	MotionNotify	XPointerMovedEvent
LeaveWindowMask	LeaveNotify	XLeaveWindowEvent
EnterWindowMask	EnterNotify	XEnterWindowEvent
ExposureMask	Expose	XExposeEvent

The event loop in an X-Client therefore has the following structure: (see also fig 9.3)

```
/* before the loop: */

for (i=0;i<MAX_DIGIT;i++)
 XSelectInput(display,digit_ids[i],ExposureMask);

for (i=0;i<MAX_UP_BUTTON;i++) {
 XSelectInput(display,up_button_ids[i],ExposureMask | ButtonPressMask);
}
/* and now the event loop */

for (;;) {
 XNextEvent(display,&event);

 switch (event.type) {
  case (Expose):
     if (event.xany.window == digit_ids[0]
         /* redraw first digit .... */
    break;

  case (ButtonPress):

        /* check from which window it comes and treat it */

     if (event.xany.window == exit_button_id)
        exit(0);
}
```

Figure 9-3 Events and Event Masks



In the above example Expose events need to be enabled for each of the subwindows and ButtonPress events must be enabled for the up/down arrows and the exit button.

When drawing into a window using the X drawing primitives, the window must already be mapped onto the screen and all window properties like position, size, id ... must be known. Since the visualization is done by the X-Server there is a problem of synchronization. Therefore we usually create and map the windows to be drawn in during the program initialization. Interest in Expose events are declared as well. As soon as the X-Server has mapped the window (all information on the window is available) an expose event in generated. The drawing is then down in the expose event handler.

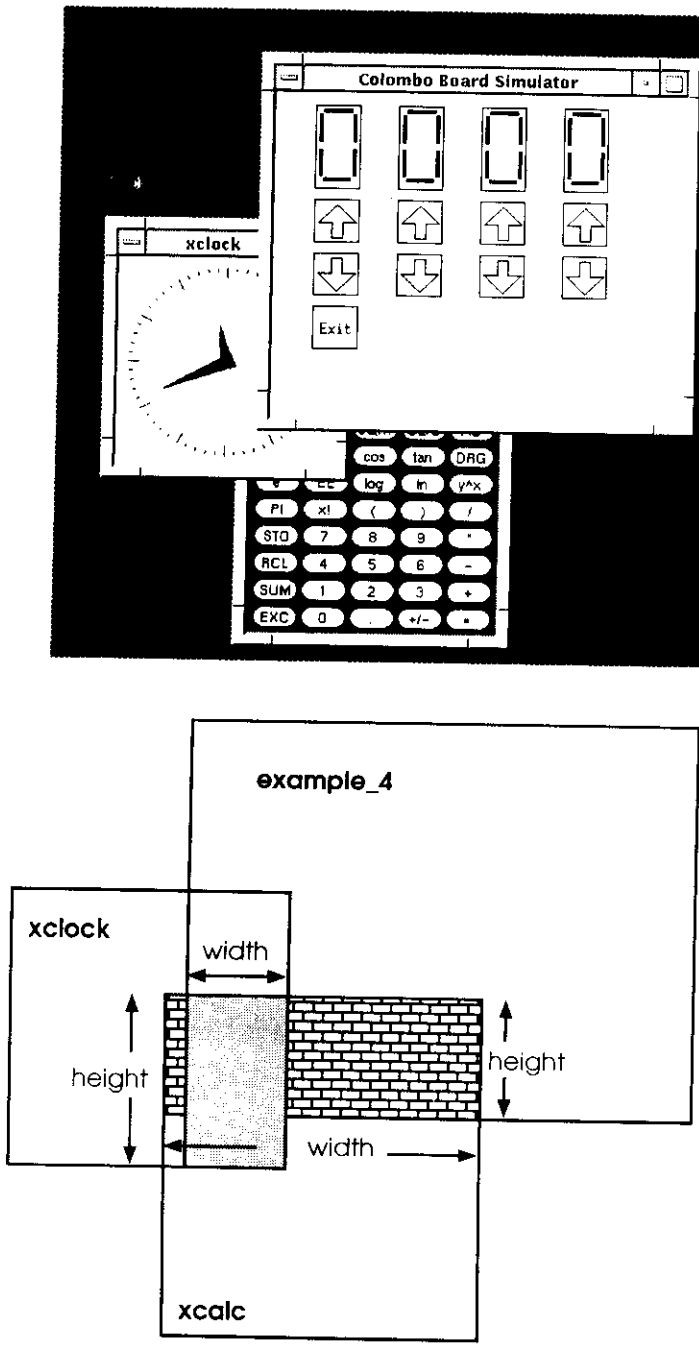
The **XExposeEvent** structure has the following form:

```
typedef struct {
  int               type;
  unsigned long     serial;
  Bool              send_event;
  Display           *display;
  Window            window;
  int               x,y;
  int               width,height;
  int               count;
} XExposeEvent;
```

The x,y,width,height parameters in the structure describe a rectangle of pixels which must be redrawn. As can be seen in fig 9.4 redrawing of several rectangles may be needed. The count entry indicates how many more such expose events are going to follow. The easiest method to treat these events discards all expose events with nonzero count and redraws the full window on the last (count=0) expose event.

If the window size does not change, we can put our drawing into a pixmap of same size as the window and on expose events copy the pixmap onto the window using XCopyArea.

Figure 9-4 Expose Events



This diagram shows the rectangles to be updated if the calculator is brought into foreground.

10

The Motif Widgets

Up to now we only used XLib calls . We managed with some difficulties to implement a "button", treating mouse button clicks within the up/down windows and a sort of label containing the digits. It seems to be a good idea however to standardize on how such a button should react and on how it should look like (contain some text or bitmap, be activated when mouse button1 is pushed and released within its window). This is the so called "look and feel" which is implemented in libraries lying above the XLib. A window together with an input/output semantic is called a **widget**. Typical examples are:

- labels
- push buttons and toggle buttons
- pulldown and popup menus
- boxes and forms containing other widgets
- text input widgets and many more.

For us widgets are simply user interface objects which are the building blocks for our applications. There are several widget sets available on the market. The most common ones are:

- Motif
- OpenLook
- the Athena Widgets

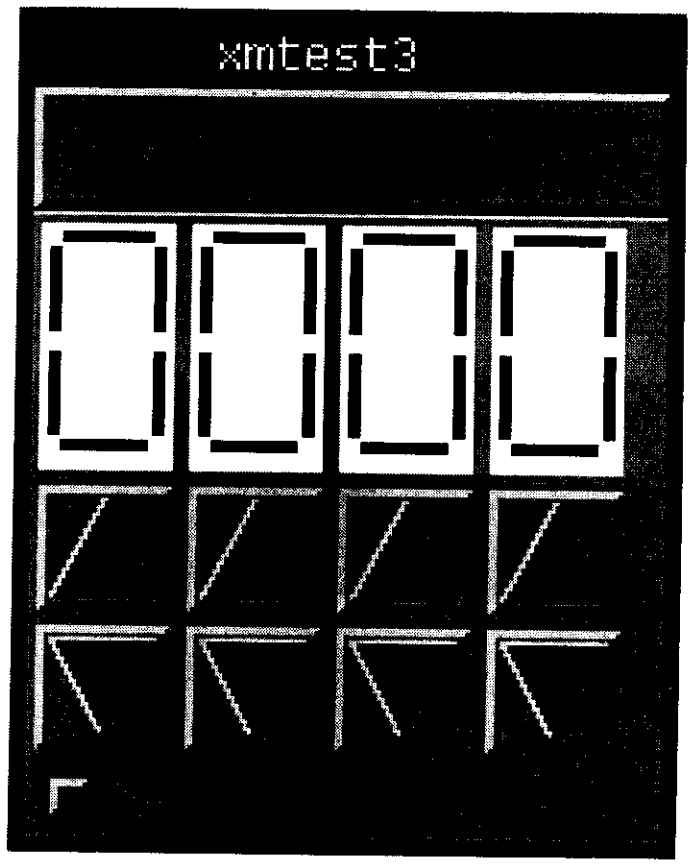
Fig 10-1 shows some of the Motif widgets.

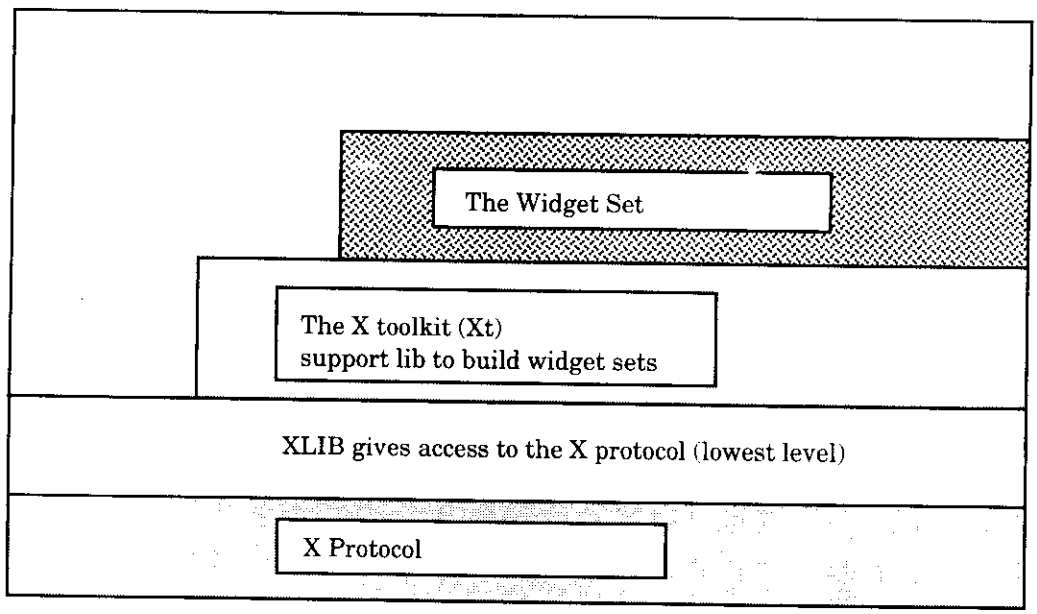
The most commonly used widget set is **Motif**, a commercial software package distributed by the **O**pen **S**oftware **F**oundation (OSF). For approximately 2 years now a group of people from the free software community has set out to provide a free Motif clone, which they call **LessTif**. This package is supposed to be entirely compatible with Motif 1.2 once finished. The problem is that the package is still in alpha stage, several parts of Motif are still missing (mainly UIL the User Interface Language) and there are still quite a few bugs left. Still, the exercises needed for the course can be implemented without too much hassle and we decided to teach LessTif because of the widespread use of its commercial counterpart Motif.

The Athena widget set was intended as a testbed for the X Toolkit Intrinsics (Xt), a library providing routines for building and accessing widgets. It is rather small and simplistic but quite a few applications have been implemented with it. During the last courses we used the Athena set because LessTif was too unstable.

In this chapter we will learn how to write the Colombo example with the help of Motif widgets.

Figure 10-1 The Motif Widgets





Instead of building the window hierarchies the application now builds up widget instance hierarchies. Again we have a "root widget" called the TopLevelShell. This widget communicates with the window manager to set up the decoration of its window. The child of the TopLevelShell is usually a XmMainWindow widget, which contains an area for

- Menu bar with pulldown menus
- a container widget used as workarea and containing other widgets
- optional scroll bars for the work area
- a command area (which we will not use!)
- and a message area, which may display error messages

As for the windows in XLib the windows of the widgets are clipped to their parent widget window boundaries.

Even though most applications use a XmMainWindow as their base window it is perfectly possible to use a container widget or even a simple widget (see the "hello world" example) for that purpose. In Motif we have essentially 2 types of different container widgets, the XmBulletinBoard, where the positions of the children (widgets within the container) are specified as absolute values and the XmForm, where all positions are relative to the container or relative to other widgets within the container.

The widgets provide a data structure containing so called resources, which describe them fully. When creating a widget instance all resources are defaulted to reasonable values but they can be changed at creation or later during runtime.

The widgets we will be using for our Colombo Board Simulator are the following:

- **XmMainWindow**
- **XmForm**
- **XmLabel (for the digits)**
- **XmArrowButton** for the up and down buttons
- **XmCascadeButton** and **XmRowColumn** for the pulldown menu
- **XmToggleButton** for the horn

In additon we will use **XmScale** for a more simple exercise at the beginning.

Before using any widget the X toolkit must be initialized and the TopLevelShell must be created. This can be done with the call:

Widget
XtVaAppInitialize(app_context,applicationclass,options,num_options,argc,argv,
resource_name,resource_value ... NULL)

```
XtAppContext        *app_context; /* opaque type containing app specific data*/
char                *application_class; /* should be "ICTP_examples" */
XrmOptionDescRec    options[];   /* You may specify X specific options in the */
                                 /* command line. The command line parser */
                                 /* will pick out all these options and leave  */
                                 /* the non X specific ones. Put NULL here     */
Cardinal            num_options; /* we don't treat special X options, put 0    */
Cardinal            *argc;
char                *argv[];
```

a NULL terminated list of resource/value pairs.

This initializes the toolkit, opens the display and creates the TopLevel Shell whos identifier is returned and which is the great-grandfather of all other widgets. The TopLevelShell communicates with the window manager and gets its decoration.

Now we can start to build the widget instance hierarchy. For each type of widget an include file containing widget specific definitions is provided. In order to create a widget call

Widget **XtVaCreateManagedWidget**(widget_name,widget_class,parent,
resource_name,resource_value ,... NULL)

```
String          widget_name;        /* give it the name you like */
WidgetClass     widget_class;       /* defined in the include file */
Widget          parent;             /* used to build the hierarchy */
```

a NULL terminated list of resource/value pairs.

The following tables show the widget names, their class name and the corresponding include file name for the widgets we will be using:

Widget Type	Widget Class Name	include file
XmMainWindow	xmMainWindowWidgetClass	<Xm/MainW.h>
XmBulletinBoard	xmBulletinBoardWidgetClass	<Xm/BulletinB.h>
XmFrame	xmFrameWidgetClass	<Xm/Frame.h>
XmPushButton	xmPushButtonWidgetClass	<Xm/PushB.h>
XmArrowButton	xmArrowButtonWidgetClass	<Xm/ArrowB.h>
XmLabel	xmLabelWidgetClass	<Xm/Label.h
XmToggleButton	xmToggleButtonWidgetClass	<Xm/ToggleB.h>
XmCascadeButton	xmCascadeButtonWidgetClass	<Xm/CascadeB.h>
XmRowColumn	xmRowColumnWidgetClass	<Xm/RowColumn.h>

As an example we show how to create an XmMainWindow and an XmForm widget with default resources:

```
#include <Xm/Form.h>
#include <Xm/MainW.h>

Widget toplevel,main_window,form;

main(argc,argv) ....

toplevel = XtAppVaInitialize( ...

main_window = (Widget) XtVaCreateManagedWidget(,"main_window",
        xmMainWindowWidgetClass,toplevel,NULL);

form = (Widget) XtCreateManagedWidget("form",
        xmFormWidgetClass,main_window,NULL);
```

A widget tree (widget instance hierarchy) can easily be built using several of these XtCreateManagedWidget calls. In order to bring the windows corresponding to these widgets onto the screen we must "realize" the root of the tree. The command:

XtRealizeWidget(toplevel)

does this for us. Contrary to our window examples the widgets already contain code for treatment of the X events. However the programmer must be notified of certain sequences of events like ButtonPress followed by ButtonRelease within the window of a command widget, which corresponds to "pressing the pushbutton" on the screen.

This can be done with **callbacks**: Almost all widgets allow the user to connect callback routines to certain actions. Use the routine:

XtAddCallback(widget_id, callback_name, callback, client_data);

```
Widget          widget_id;
String          callback_name;      /* In our case: XtNcallback */
XtCallbackProc  callback;           /* address of callback proc */
caddr_t         client data;        /* address of data to be passed */
```

XtCallbackProc is defined as:

typedef void (***XtCallbackProc**) (widget_id, client_data, call_data);

```
Widget    widget_id;
caddr_t   client_data;      /* specified in XtAddCallback */
caddr_t   call_data;        /* call specific data, depends on the
                               widget */
```

For the exit button in our example programs we will therefore construct a callback procedure:

```
void exit_callback(w,client_data,call_data)
Widget    w;
caddr_t   client_data,call_data);

{ /* cleanup if needed */
exit(0);
}
```

After creation of the exit button

```
exit_button = XtVaCreateManagedWidget("exit_button",
                                      XmPushButtonWidgetClass,
                                      main_widget,NULL);
```

(creation of a command widget) we connect this routine as "activate callback" to the widget:

XtAddCallback(exit_button,XmNactivateCallback,exit_callback,NULL).

Once this widget tree is complete and all callbacks are connected, control is given back to the window system, which will call the registered callback routines as soon as the corresponding event sequence has happened. The call

XtAppMainLoop(theApp);

does this.

Sometimes it is desired to map non X events to callback routines. You may for example want to execute a callback when a certain time has elapsed or when a device driver has data to be read. This can be accomplished with:

XtIntervalId **XtAppAddTimeOut**(app_context, interval, timer_proc, client_data)

where interval is in msec and XtIntervalId is an identifier which allows you to distinguish the timer events in case you use more than one. The corresponding callback routine must be defined as:

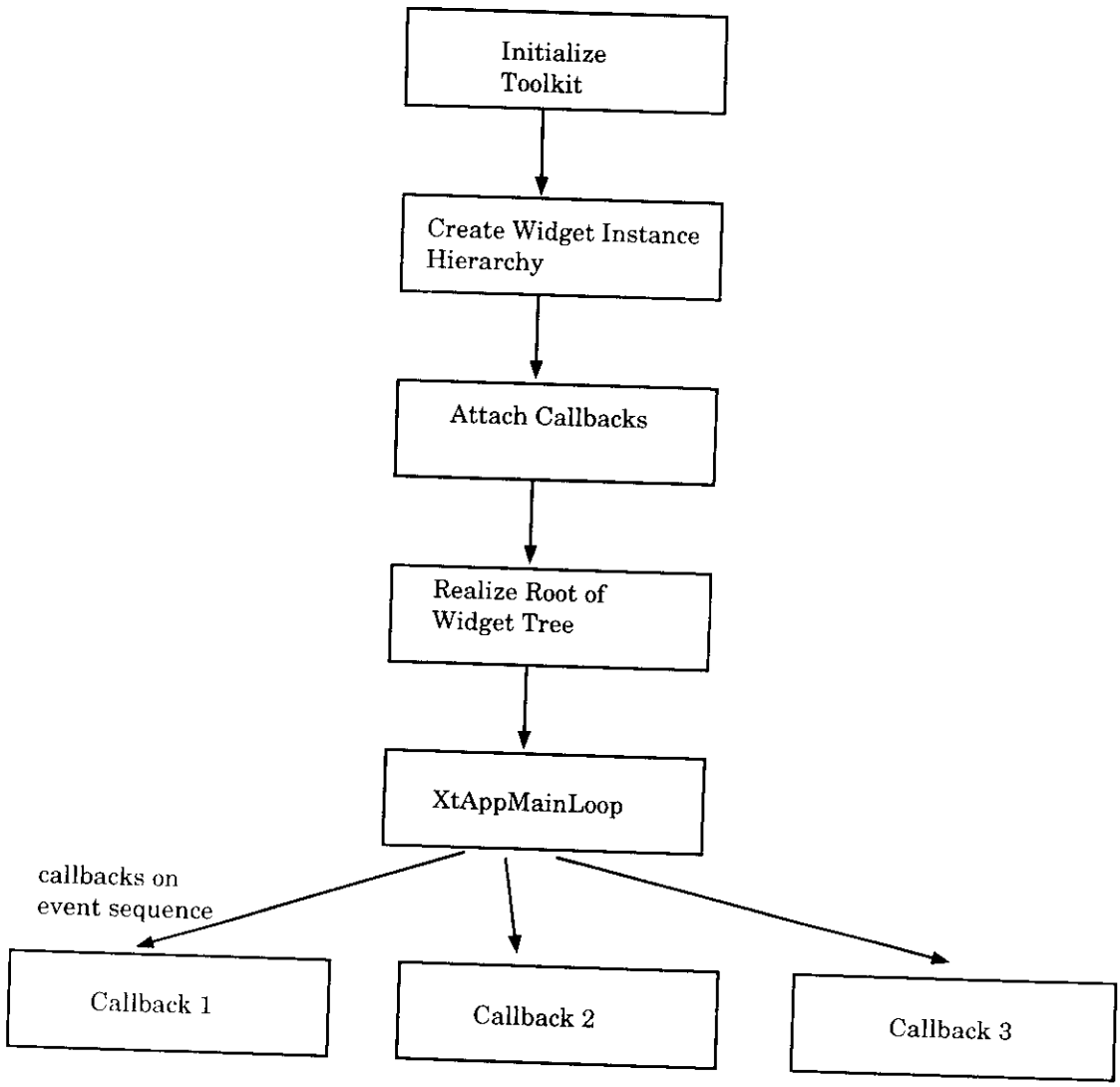
XtTimerCallbackProc timer_proc(XtPointer interval_id, XtPointer client_data);

Similar routines are available for the driver case. Here we would use

XtInputId XtAppAddInput(app_context, source, input_proc, client_data)

The flow of control in an application program using widgets has therefore the following form:

Figure 10-3 Typical flow of control in a Motif application

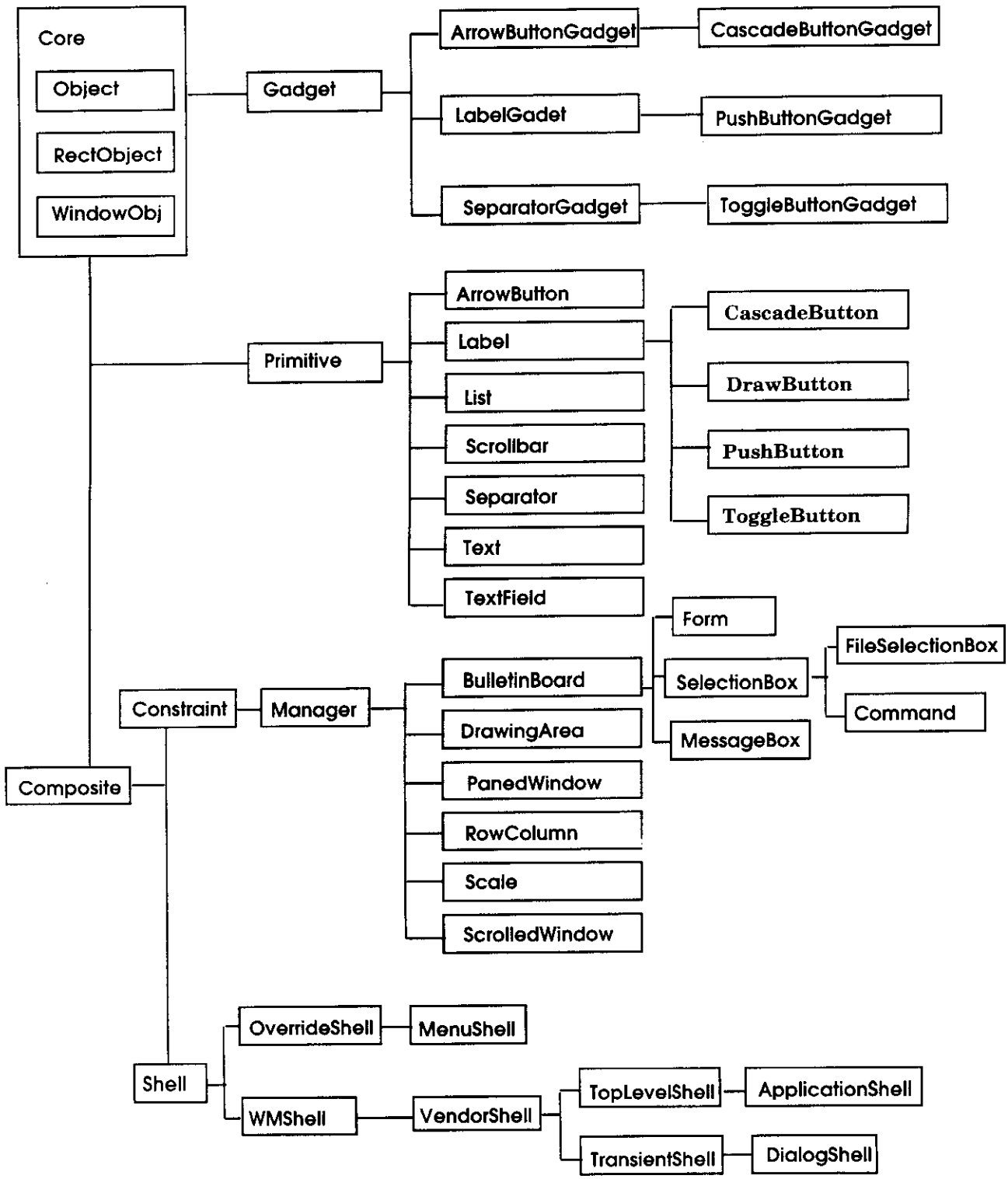


The Widget Class Hierarchy

Widgets are built using object oriented programming techniques. This means, that there are some "basic" widgets (basic data structures and access routines, so called methods) which are defined in the Toolkit intrinsics. These basic widgets are the **Core** widget and several **Shell** widgets. A new widget uses part of the data structures and methods defined in these basic widgets and augments them with new data entries and new access routines or modifies some of the properties. Consider a label and a pushbutton (command widget): The label has some basic resources like width, height, border width, foreground/background color etc. which it inherits from the core widget (all widgets have these properties!). In addition it has a string or a bitmap associated with it. A command button can be considered to be a label widget having the additional features of being active. In the case of the Athena command widget the window is highlighted when the cursor enters its window and a callback for activation can be attached.

When the programmer calls XtVaCreateManagedWidget a widget instance of the specified class is created. This instance contains the individual values for the label string, the colors etc. while the class provides some additional data fields valid for all instances and all the access routines. This is why we always insist on talking about the widget **instance** hierarchy and not just the widget hierarchy!

Figure 10-4 The Motif Class Hierarchy



In order to change the default layout of a widget we must access its resources. These changes can be performed during widget creation or using the Xt routine **XtVaSetValues** at runtime.

```
XtVaSetValues(widget_id,name1,value1,name2,value2 .... NULL)
Widget      widget_id;
String      name1;
XtArgVal    value1; etc.
```

Its counterpart for fixed length argument lists (resource name - resource value pairs) is XtSetValues. As you may expect, similar routines for toolkit initialization (**XtAppInitialize**) are also available. Before using these routines we must fill an argument list, where each element is of the following type:

```
typedef struct {
 String     name;      /* name of resource to by modified */
 XtArgVal   value;
} Arg, *ArgList;
```

A Macro has been defined to accomplish this:

```
XtSetArg (arg,resource_name,value)
Arg         arg;                /* argument to be set */
String      resource_name;      /* e.g. XtNwidth, XtNheight ... */
XtArgVal    value;              /* value of the resource */
```

This argument list and the number of entries may be specified in the widget instance creation routine or in XtSetValues:

```
void XtSetValues(widget_id,args,num_args);
Widget      widget_id;
ArgList     arg;
Cardinal    num_args;
```

Imagine we want to set the string "Quit" in the exit button and set its width and height to fixed values:

```
Arg         args[5];
XtSetArg(args[0],XtNlabel,"Quit");
XtSetArg(args[1],XtNwidth,100);
XtSetArg(args[2],XtNheight,50);
XtSetValues(exit_button,args,3);    /* will set these 3 resources at runtime */)
```

In the same manner it is possible to read back resources from a widget:

```
XtSetArgs(arg[0],XtNlabel,&return_string);
XtGetValues(exit_button,args,1);
```

will return the label string into return_string. Of course the variable length counterpart **XtVaGetValues** is also available.

The next step is to give you a compilation of resources that you will need for development of the exercises on widgets. This list is of course far from being complete. We therefore encourage you to have a look into the LessTif Manual, which you have online. You can access it from the root window pulldown menu.

The compound string (XmString)

You would expect that putting text onto the widgets should be one of the simplest things you can do. Well, you are wrong! In fact the Athena widgets set uses simple text strings (char *) for its labels, titles etc. Motif however goes a step further. Imagine you want to intermix greek and latin characters, you want to write a text in German (having those strange umlauts) or in french (with it accents) or you want to intermix cyrillic and latin text. All this is possible with the Motif compound string concept. So you can have labels like "ε = 25 π mm mrad" (used very often in accelerator physics!) very easily.

For our course we will restrict ourselves to simple english text but of course you are encouraged to play and try things out. These are the calls that wiil be enough for you:

- **XmString XmStringCreateLtoR**(text,tag)
  char *text;
  char *tag
  In our case text will be the character string we want to convert into a XmString and tag will be set to XmSTRING_DEFAULT_CHARSET. The call will also treat embedded '\n' correctly.

- **XmString XmStringCreateLocalized**(text)
  char *text.
  Here text must be a NULL terminated string without embedded '\n'.

The normal use of XmStrings can be demonstrated by setting a new text within a label:

```
XmString new_string;
Widget   label;
some code initializing label...
new_string =XmStringCreateLocalized("My new String");
XtVaSetValues(label,XmNlabelString,new_string,NULL);
XmStringFree(new_string).
```

The last line is needed because the calls creating XmStrings allocate the memory space they need for the XmString. The deallocation however is left to the user of the call.

Pixmaps

As you can see from the table of resources later, an XmLabel not only can display text strings in all variations, it is also possible to show Pixmaps. The easiest way to build bitmaps is the bitmap editor (see chapter on X Pixmaps) which stores bitmaps in an X specific way onto disk. Motif provides a series of calls allowing someone to read these files and convert them into Pixmap structures that can be used in labels, pushbuttons etc.
We need this feature when we want to generate labels which ressemble the seven segment displays closely.

```
Pixmap XmGetPixmap(screen,image_name,foreground,background)
Screen      screen;                 /* try XtScreen(widget) */
char        *image_name (the bitmap filename);
Pixel       foreground;             /* BlackPixel(XtDisplay(widget),SCREEN */
Pixel       background;             /* WhitePixel(XtDisplay(widget),SCREEN) */
```

If things go wrong the result will be XmUNSPECIFIED_PIXMAP. If the call finds out that the same pixmap had been loaded before, it will not need to go onto disk, but it can pick up the pixmap from the pixmap cache.

A typical sequence to show Pixmaps is:

```
#define SCREEN 0
Display *display;
Screen  *screen;
display = XtDisplay(widget);
screen = XtScreen(widget)
my_pixmap = XmGetPixmap(screen,"eight.bm",
                        BlackPixel(display,SCREEN),,WhitePixel(display,SCREEN));
if (my_pixmap == XmUNSPECIFIED_PIXMAP) {
        alert ("pixmap no good\n");
        return ERROR;
}
XtVaSetValues(label,XmNlabelType,XmPIXMAP,
                        XmLabelPixmap(my_pixmap,NULL);
```

The Core Widget

As we have seen in the Motif Class hierarchy, all widgets have **Core** as a superclass. For this reason all widgets inherit the resources defined in **Core**. We only put those resources into the table that you will definitely need for the solution to the exercises but many more are available. Please have a look at the LessTif docs.

Resource Name	Type	Default	Desciption
XmNx	Position	0	x positon relative to the origin of the parent
XmNy	Position	0	x positon relative to the origin of the parent
XmNwidth	Dimension	dynamic	width of the widget. By default the geometry management decides which width is needed
XmNheight	Dimension	dynamic	height of the widget. Some comments as for width

The XmMainWindow

Here are a few resources for the XmMainWindow. Again the list is far from being exhaustive. So please have a look at the LessTif documentation. There you will find different sets of resources: Firstly the specific resources for the widget and secondly all the resources of its super classes. You will find back the widget class hierarchy explained previously (The XmLabel will have its own resources, then the resources of its superclass XmLabel, then the resources of XmPrimitive and so on (see figure 10-4)).

Resource Name	Type	Default	Desciption
XmNworkWindow	Widget	NULL	Container widget that constitues the work area. Most of the different user widgets will be put here.
XmNmenuBar	Widget	NULL	The menu bar containg pulldown menus. Usually there are at least 3 of them: the File menu the Edit menu the Help menu
XmNshow Separator	Boolean	False	Use XmSeparators to separate the different XmMainWindow areas.

The XmBulletinBoard

Sorry, there is no description of this widget within this script. For the exercises we only use resources inherited from the Core Widget namely:

- XmNx
- XmNy
- XmNwidth
- XmNheight
- XmNbackground

The XmForm

The XmForm widget is a container widget performing geometry management on its children. The children of a form may specify their position relative to each other or relative to their parent. When a widget is child of a form it has the following additional resources:

Resource Name	Type	Default	Desciption
XmNtopAttachment	unsigned char	XmATTACHMENT_NONE	describes where to attach the widget. Some possibilies: XmATTACH_FORM XmATTACH_WIDGET XmATTACH_OPPOSITE_WIDGET XmATTACH_POSITION
XmNbottomAttachment	unsigned char	XmATTACHMENT_NONE	see above
XmNleftAttachment	unsigned chat	XmATTACHMENT_NONE	see above
XmNrightAttachment	unsigned char	XmATTACHMENT_NONE	see above
XmNtopWidget	Window	NULL	widget onto which we hook on
same for bottom, left, right			
XmNtopOffset	int	0	Offset for the attachment
same for bottom, left, right			
XmNfractionBase	int	100	used for relative positioning
and there are many more see the LessTif docs			

The XmScale

The XmScale widget is going to be used in an introductory (and therefore simpler) example where we use it as a linear indicator for an analog value. We want the scale to be vertical with the maximum value on top. The actual value should also be printed as a number. The range of values is defined to be 0 - 5000,, which may stand for 0 mV to 5000 mV, the digital values we get from Angs IO board.

The XmScale widget also has so-called convenience routines which ease the reading and writing of scale values:

- XmScaleGetValue(Widget w, int *value_return) and
- XmScaleSetValue(Widget w, int value)

do what you would expect.

It is also possible to give the position in percentage of the total XmForm width and height:

The is done with the resource XmNfractionBase

- Set the attachment type to XmATTACH_POSITION
- Set XmNfractionBase to say 100
- Now if you set XmNtopOffset to 30 then the widget will be placed ad 30% of the XmForm height

Resource Name	Type	Default	Desciption
XmNshowValue	Boolean	False	show not only the analog value by setting the position of the scale, but also its numerical value
XmNtitleString	XmString	NULL	the title
XmNorientation	unsigned char	XmVERTICAL	XmVERTICAL or XmHORIZONTAL
XmNprocessingDirection	unsigned char	dynamic	XmMAX_ON_TOP XmMAX_ON_BOTTOM XmMAX_ON_LEFT XmMAX_ON_RIGHT

The XmLabel

Labels have two different visual aspects: They may either display text or pictures in form of pixmaps. Since we will use both in the Colombo exercise here are the resources to be changed:

Resource Name	Type	Default	Description
XmNlabelString	XmString	label name	String to be displayed in label
XmNlabelPixmap	Pixmap	none	Bitmap to be displayed instead of text string
XmNlabelType	unsigned char	XmSTRING	How to justify the label

The XmArrowButton

The **XmPushButton widget**, being a subclass of the XmLabel widget, has got all the label widgets resources with the possibility to connect an activation callback in addition.

The XmArrowButton reacts like a XmPushButton but already provides arrows as labels. The arrow direction can be specified by means of resources:

Resource Name	Type	Default	Description
XmNarrowDirection	unsigned char	XmARROW_UP	direction of Arrow: XmARROW_UP XmARROW_DOWN XmARROW_LEFT XmARROW_RIGHT

Pulldown Menus

Still missing is the way to construct menus. As explained above the children in the widget instance hierarchy are always clipped to their parent windows. When creating menus this is not acceptable and we must therefore create another shell widget, which will contain the menu. On the other hand we don't want decoration of the window coming up when we activate the menu. This can be accomplished by creating a "Popup shell".

Luckily enough Motif provides a "convenience routine" which does all the work for us:

```
file_menu = XmCreatePulldownMenu(parent_widget,widget_name,
                                 args,no_of_args)
Widget parent_widget;
char   *widget_name;
Arg    *args;
int    no_of_args;
```

creates the pulldown menu. However the menu as yet has no entry in it and is neither hooked onto a button which will pop it up , nor placed into a menu bar. Even worse, it will not appear on the screen because it is not managed. Managing a widget is somehow similar to mapping a window in XLib. It can be accomplished by

XtManageChild(widget_id);

In order to get rid of the other problems we first create a pushbutton and place it into the pulldown menu:

```
label_string = XmStringCreateLocalized("Quit");
quit_button = XtVaCreateManagedWidget("quit_button",
                                      xmPushButtonWidgetClass,
                                      file_menu,
                                      XmNlabelString,label_string,NULL);
XmStringFree(label_string);
```

then we create the menu bar with another convenience routine:

```
menu_bar = XmCreateMenuBar(main_window,"menu_bar",args,(Cardinal)NULL);
```

and finally the button that pops up the menu:

```
label_string = XmStringCreateLocalized("File");
file_button = XtVaCreateManagedWidget("file_button",xmCascadeWidgetClass,
                                      menu_bar,
                                      XmNlabelString,label_string,
                                      XmNsubMenuId,file_menu,NULL);
XmStringFree(label_string);
```

The resource XmNsubMenuId tells the CascadeButton, which menu to pop up once it is activated.

Of course the menu_bar must be placed into the main window which can be accomplished with

XtVaSetValues(main_window,XmNmenuBar,menu_bar).

Now

- **menu_bar** is the standard menu bar of the XmMainWIndow that forms the base of our application
- it contains a pulldown menu named **file_menu**
- this file_menu contains a single button (that can be activated once we attach a callback procedure to it) namely the **quit_button**
- the file menu is visible and can be popped up through a XmCascadeButton named **file_button**

I know this looks pretty complex, but even though pulldown menus are extremely common in GUIs they are amongst the most complex structure you can have in user interface programming.

Dialog Boxes

Up to now all widgets came up onto the screen once the toplevel widget has been realized. Very often however we want a box with an error or warning message to pop up only if an error condition has been encountered. This can be done with a **XmMessageBoxDialog** widget.

We create it with

XmCreateErrorBox("error_box",parent,args,no_of_args) and we manage it only once the error has happened. This box contains several buttons, one of which will automatically unmanage the box and thus make it disappear.

Resource Name	Type	Default	Description
XmNmessageString	XmString	""	error message

Connections of widgets to XLib

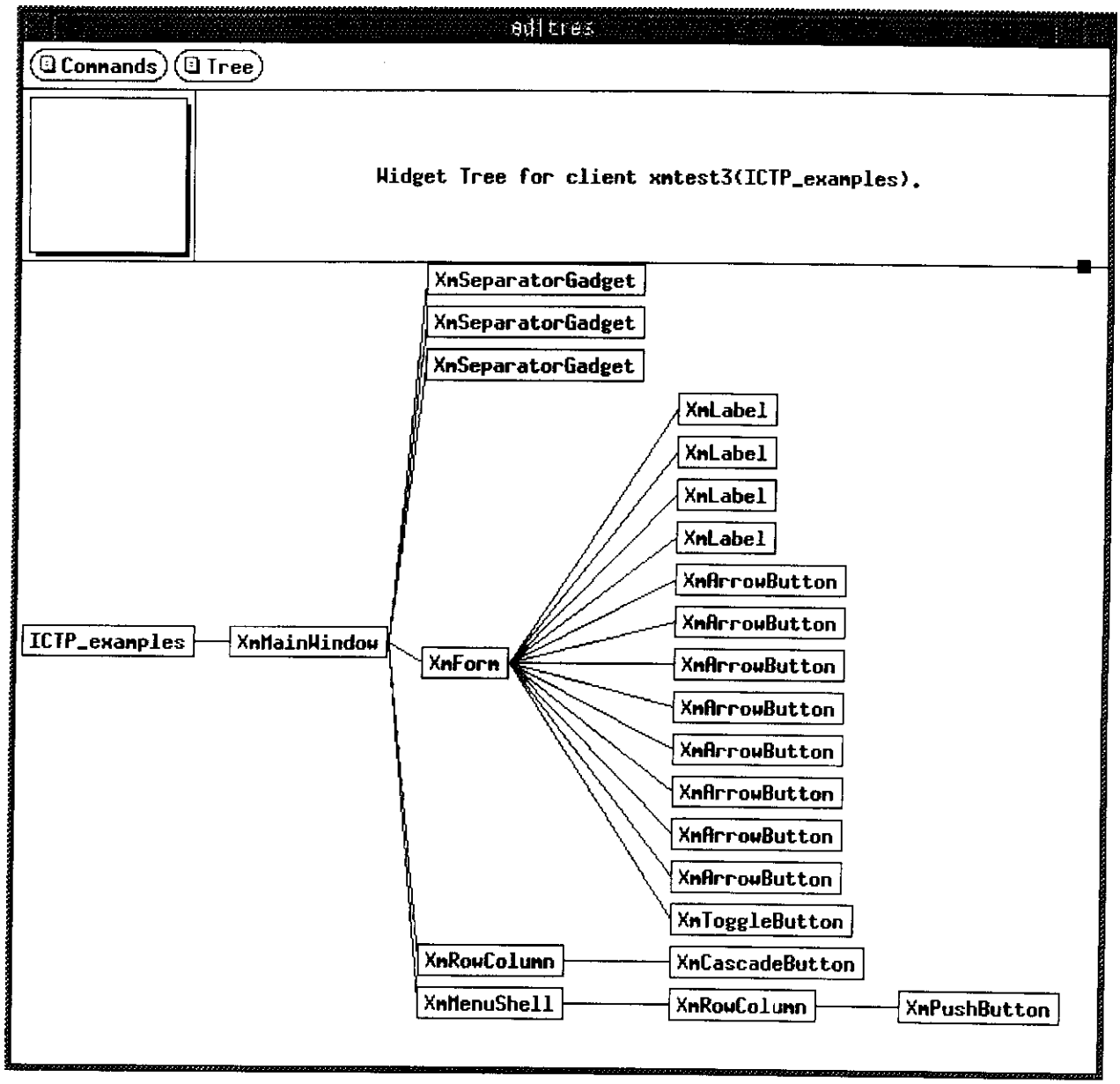
For several widgets a bitmap id can be used in order to display pictures in buttons, labels etc. When creating a bitmap however we need the identifier of the opened server connection (display variable) or a window id. In order to get this information for a specific widget (which window corresponds to the main_widget for example) several calls are available:

Display	**XtDisplay**(widget_id)	returns the id of the server connection
Window	**XtWindow**(widget_id)	returns the widgets window id.
Screen	**XtScreen**(widget_id)	returns the widgets screen structure

Using these calls you may now happily intermix Xm, Xt and Xlib calls.

Now that we know everything needed to build the GUI for the Colombo example, here is a picture of the widget classes needed for the program:

Figure 10-5 Widget Classes for the Colombo example:



11

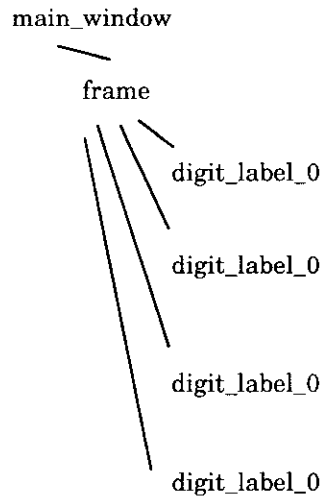
Resources and the Widget Creation Library

In the last chapter we have seen that each widget has associated with it a large number of resources (XmNwidth, XmNlabelString, XmNbackground, XmN....) which describe it. These resources can be initialized during the creation procedure of the widget and modified by the running program. Many of the resources need only initialization (or even keep their default values) and are untouched during run time. Think of the label string on a label widget for example.

The Xt library allows another very elegant way to modify resources: The resource file. This file contains resourcename-resource value pairs and it is ready during program startup. A typical resource file is $HOME/.Xdefaults, which you should have a look at.

Now the question is: How do we specify a widget and its resources. The resource names are the same as the names used within XtSetValues, with the leading "XmN" taken away (XmNwidth -> width, XmNlabelString -> labelString etc.)

The widget is specified by giving its path through the widget tree: The digit_label_0 would then be called: **colombo.mainwindow.frame.digit_label_0**.width: 45



Like with filenames in Unix wildcards are allowed. The * stands for "any widget" and *.digit_label_0.width: 50 would most probably have the same effect as the full specification above. Now you also understand why we always give names to the widgets (the string in the widget creation routine XtCreateManagedWidget). These names are used for widget identification in the resource file.

In order to go even one step further we can also specify widget classes instead of widget instances: *XmLabel stand for any XmLabel widget within any application and there are usually many more XmLabel widgets than there are widget of name digit_label_0 within an application.

In the toolkit initialization you can also give a classname to your program. With this you could for example. group all editors into a common class Editor or (as we have done) group all solutions to the college exercises into a class ICTP_examples.

You may immediately spot interesting possibilities by applying the concept of a resource file:

- Have language dependent resource files. You can then modify the text in an application for a given language by just creating a language dependent resource file. This allows you to change the language for different users without touching a single line of program code.
- Colors are a matter of taste. Is your taste different from a program authors taste? No problem, create a resource file and change the colors.

Here is part of your .Xdefaults file:

```
!
! ICTP examples
! ITCP_examples*bitmapFilePath:    /usr/local/include/X11/bitmaps
ICTP_examples*font:                *times-bold-i-*-140-*
ICTP_examples*main_widget*form.background: dark olive green
ICTP_examples*main_widget*Label.foreground: red

IOsimulator*background: grey75 IOsimulator*XmText*background: ivory
IOsimulator*XmText*fontList:-adobe-*-*-r-*-*=24-*-*-*-*-*-*
```

For those who want to observe the effect of changing resources before putting them into the resource file a very neat program has been written: **editres.** You find it under "System Management" on the root window menu. This program shows the complete widget instance hierarchy and gives you access to any resource for any widget within your application. The figures 11-1 and 11.2 show typical screen dumps for our Colombo program

Figure 11-1 The widgets instance hierarchy for the Colombo example

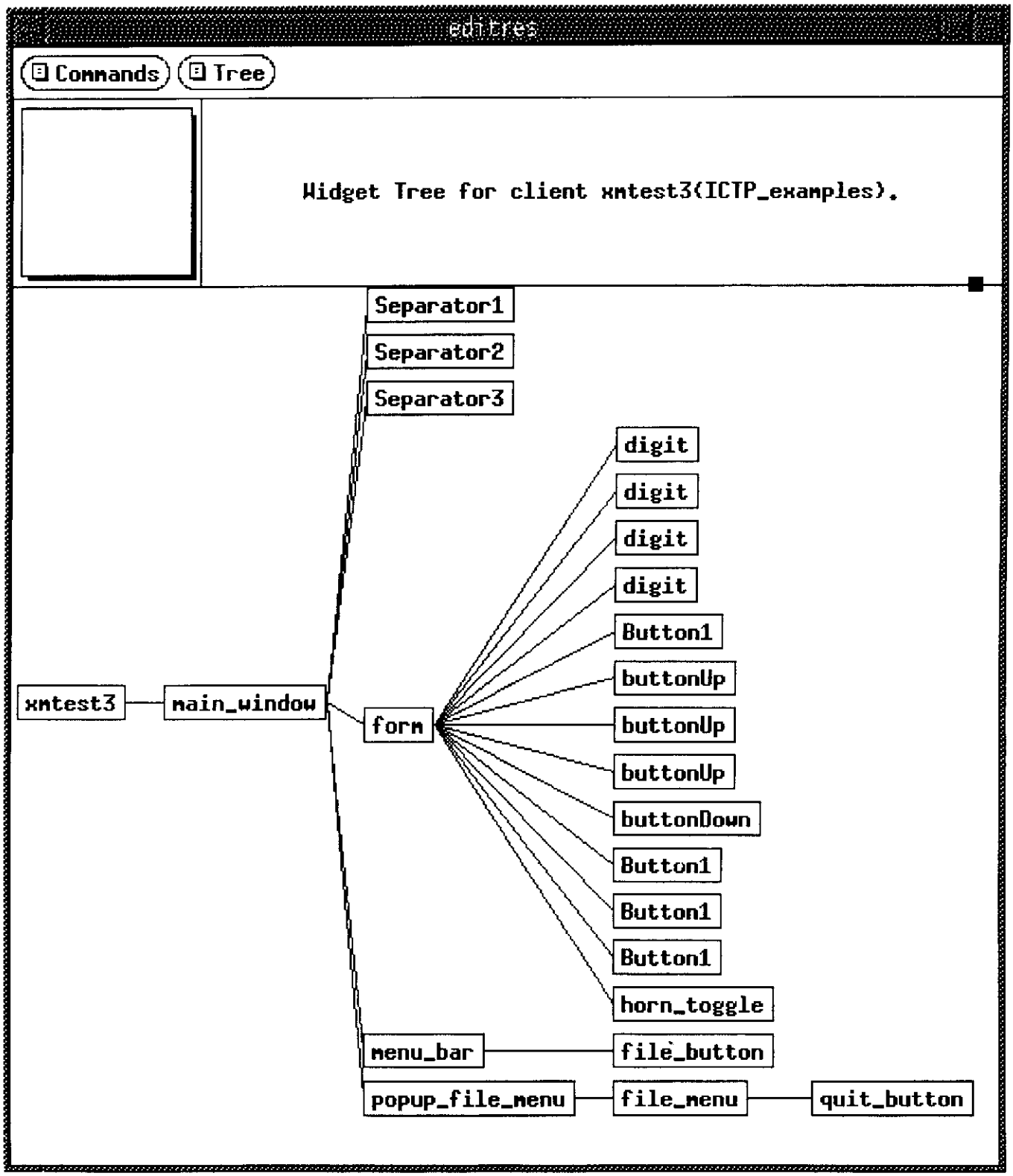
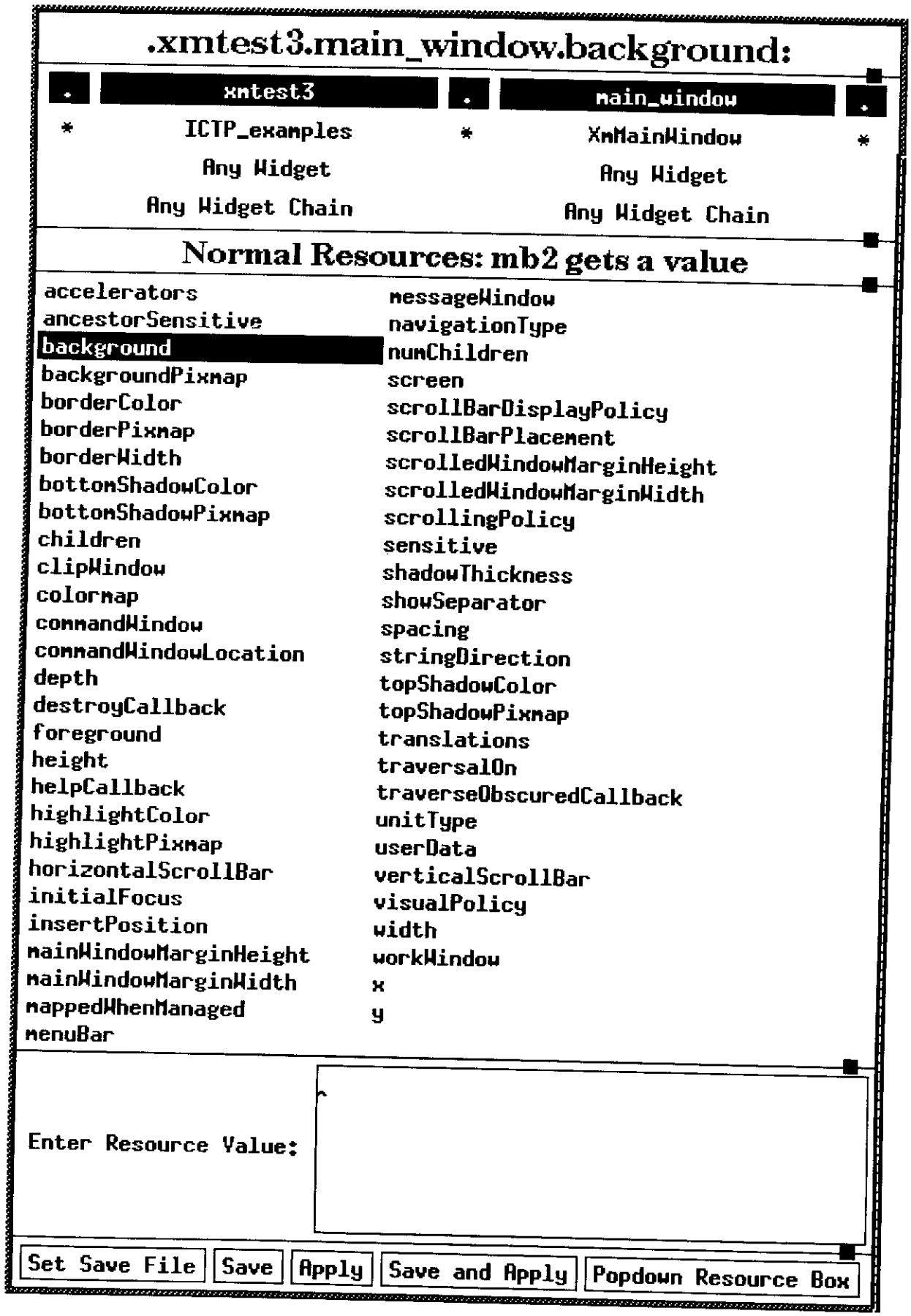


Figure 11-2 The resources for the XmMainWindow



Since we can specify all details of a widget including callbacks with resources it seems logical to also describe the widget instance hierarchy in the resource file. This feature however is **not** provided in the standard system. (Motif privides a very simple specialized language to accomplish this: the **U**ser **I**nterface **L**anguage UIL. Unluckily this is not yet working correctly in LessTif and we must renounce from using it during this college)

The **W**idget **C**reation **L**ibrary (Wcl) provides these extensions to the resource description and a few additional routines which react on them. Through registration routines widget classes and creation procedures can be made known to Wcl making the system widget set independent. Another library (Xmp) contains routines which do the job of registration of all the Motif widget creation routines for you.

An application then consists of

- a resource file
  - — containing the widget instance hierarchy and all resource specs
  - — attaching the associated user defined callbacks
  - — attaching standard callback routines provided by Wcl
- a C program
  - — registering the widget set to be used
  - — registering all callback routines
  - — creating the widget instance hierarchy from the resource specs
  - — executing the callbacks.

Within the Wcl distribution you find a C source file which acts like a frame for your own application. There is one such program for each of the Wcl supported widget sets:

- Mri the Motif resource interpreter
- Ari the Athena resource interpreter
- Ori the OpenLook resource interpreter.

These are the simplest possible application programs working with Wcl. They simply bring up the widget instance tree defined in a resource file and then fall into XtAppMainLoop doing nothing. Still these programs (Mri for our case) are very useful. They allow the programmer to test the resource file by visualizing it.

The steps for creating an application with Wcl are:

- creation of the resource file
- test with Mri
- copy Mri.c to your own application.c
- add the callbacks needed.

The following shows the Mri code:

```
#include <X11/Wc/COPY.h>

/*
* SCCS_data: %Z% %M% %I% %E% %U%
*
* Motif Resource Interpreter - Mri.c
*
* Mri.c implements a Motif Resource Interpreter which allows proto
* type Motif interfaces to be built from resource files.
* The Widget Creation library is used.
*
*****************************************************************
*/

#include <Xm/Xm.h>            /* Motif and Xt Intrinsics   */
#include <X11/Wc/WcCreate.h> /* Widget Creation Library   */
#include <X11/Xmp/Xmp.h>     /* Motif Public widgets etc. */


/*****************************************************************
**  Private Data
*****************************************************************/

/* All Wcl applications should provide at least these Wcl options:
*/
static XrmOptionDescRec options[] = {
    WCL_XRM_OPTIONS
};


/*****************************************************************
**  Private Functions
*****************************************************************/

/*ARGSUSED*/
static void DeleteWindowCB( w, clientData, callData )
    Widget          w;
    XtPointer       clientData;
    XtPointer       callData;
{
    /* This callback is invoked when the user selects 'Close' from
    ** the mwm frame menu on the upper left of the window border.
    ** Do whatever is appropriate.
    */
    printf("Closed by window manager.\n");
}

/*ARGSUSED*/
static void RegisterApplication ( app )
    XtAppContext app;

/*
 -- Useful shorthand for registering things with the Wcl library
*/
#define RCP( name, class) WcRegisterClassPtr ( app, name, class );
#define RCO( name, constr) WcRegisterConstructor( app, name, con-
str );
#define RAC( name, func ) WcRegisterAction ( app, name, func );
```

```
#define RCB( name, func ) WcRegisterCallback( app, name, func,
                                                NULL );

    /* -- register widget classes and constructors */
    /* -- Register application specific actions */
    /* -- Register application specific callbacks */
}


/*****************************************************************
*   MAIN function
*****************************************************************/

#if defined(XtSpecificationRelease) && XtSpecificationRelease == 4
#define CARDINAL(argc) (Cardinal*)(&argc)
#else
#define CARDINAL(argc) (&argc)
#endif

main ( argc, argv )

    int     argc;
    String  argv[];
{
/*
-- Intialize Toolkit creating the application shell
*/
    Widget appShell = XtInitialize (
          WcAppName(  argc, argv ),  /* application shell name  */
          WcAppClass( argc, argv ),  /* class name is 1st
                                        resource file name */
          options, XtNumber(options),/* resources which can be
                                        set from argv */
          CARDINAL(argc), argv
    );
    XtAppContext app = XtWidgetToApplicationContext(appShell);

/*
-- Register all application specific callbacks and widget classes
*/
    RegisterApplication ( app );

/*
-- Register all Motif classes, constructors, and Xmp CBs & ACTs
*/
    XmpRegisterAll ( app );

/*
-- Create widget tree below toplevel shell using Xrm database
*/
    if ( WcWidgetCreation ( appShell ) )
          exit(1);

/*
-- Realize the widget tree
*/
    XtRealizeWidget ( appShell );
```

```
/*  -- Optional, but frequently desired:
** Provide a callback which gets invoked when the user selects
** Close' fromthe mwm frame menu on the top level shell.  A real
** application will need to provide its own callback instead of
** DeleteWindowCB, and probably client data too.  MUST be done
** after shell widget is REALIZED!  Hence,
** this CANNOT be done using wcCallback (in a creation time call
** back).
*/
    XmpAddMwmCloseCallback( appShell, DeleteWindowCB, NULL );

/*
-- and finally, enter the main application loop
*/
    XtMainLoop ( );
}
```

Our own Colombo program written in Wcl will look pretty much like Mri.c except that some code must be added:

Firstly we must register our own callbacks using the Macro RCB defined in Mri.c:

RCB("ICTP_incrProc",ICTP_incrProc)

This tells Wcl that the string "ICTP_incrProc" which stands for the callback procedure in the resource file must be mapped onto the address ICTP_incrProc, which is the starting address of the callback procedure.

Secondly we must read the bitmap files as we did in the standard C examples.

Of course WcWidgetCreation() is not the only routine available in Wcl but there are some 50 routines which help you in designing your application. All these routines are described in the Wcl man page.

The only missing part is the widget instance hierarchy description in the resource file. We start off with the simplest possible examples, the world famous "hello world" program (now even simplified as compared to the C example):

```
Mri.wcChildren:                          pushbutton
*pushbutton.wcCreate                     XmPushButton
*pushbutton.labelString:                 Hello World!
*pushButton.activateCallback:            WcExitCallback
```

Running this program with Mri will bring up a pushbutton with the text "Hello World!" and pushing the button will exit the application.

Again many more Wcl specific resources are available but we leave it to the interested reader to have a look at the Wcl man page.
Just one more resource needs further explication:
In the standard C example we assigned pixmaps to the digit labels. Pixmaps are created at runtime however and their addresses are not known before their creation. It is therefore impossible to define them in the resource file. What can we do about this? Wcl supplies the resource wcCallback allowing us to attach a callcack procedure which is called after the creation of the digit label. Within this callback we would add the pixmap resource to the digit labels using the standard toolkit XtVaSetValues routine.

12

The ICTP device driver

This chapter has nothing to do with X-Windows programming. It has been attached for reference and it is needed to complete the exercise on the Colombo board.

A sample device driver for the ICTP board has been developed. The following gives a summary of its functions.

The ictp driver expects an I/O board using an Intel 8255 chip at I/O adress 0x300. The connections to the ICTP board must be made as follows:

- Port A: ICTP displays
  Port A is therefore programmed as **output** port.
- Port B: ICTP switches
  Port B is therefore programmed as **input** port.
- If you open minor device 0, the port B of the 8255 will be set to mode 0 (non strobed input, allowing to read directly the state of the switches. Port A is set to mode 1 (strobed I/O).
- When opening minor device 1 the 8255 chip is initialized such that both, port A and port B are set to mode 1 (strobed I/O). This allows interrupts for both ports.
- In mode 1, with port A output, the bits 4 and 5 of port C may be used as normal I/O pins, while the other bits are used as handshake signals or interrupt lines. Bit 4 of port C must be connected to CA2 (the ICTP buzzer).
- Bit 2 and Bit 6 of port C are strobe lines which must be connected to one of the interrupt generating line CA1,CA2 or CB1.

The driver functions:

Read calls:

The driver uses major number 31 and 3 minor numbers:

- read on minor number 0: read the switches
  read on minor number 1: returns the number of interrupts arrived since the last read call.
- read on minor number 2: same as above for interrupt 2

Reads for interrupts exist in 2 flavors:

- **non blocking**: The number of interrupts since the last read is immediately returned, even if it is zero.
- **blocking**: If the number of interrupts is zero, it blocks the calling process until the next interrupt (or other signal like ^C) arrives.

Write calls:

Writing works on any of the four minor devices. There are 3 different write modes which may be set up by ioctl calls (see later).

- ICTP_MODE_RAW: in this mode the data coming from the user are sent untreated to the I/O port. In order to make the displays work correctly, the user must select the suitable data/chipselect sequences (cs high + data, cs low + data, cs high + data for all digits). 12 data bytes are expected and the driver will return EINVAL if the count is wrong
- ICTP_MODE_SINGLE_DIGIT: a single data byte is accepted. The high nibble contains the digit number (0-3) and the low nibble contains the data.
- ICTP_MODE_FULL_NUMBER: a short is expected. This number will be put onto the digits.

ioctl calls:

- ICTP_SET_WRITE_MODE: sets up the writing mode. The following values are accepted:
  - — ICTP_MODE_RAW: 12 data bytes expected but anything allowed
  - — ICTP_MODE_SINGLE_DIGIT: only 1 data byte allowed
  - — ICTP_MODE_FULL_NUMBER: a short needed;
- ICTP_SET_READ_MODE: set the read mode
  - — ICTP_MODE_BLOCKING: if count = zero, block process until interrupt arrives
  - — ICTP_MODE_NON_BLOCKING: return current count immediately
- ICTP_GET_WRITE_MODE: return the current write mode
- ICTP_GET_READ_MODE: return the current read mode
- ICTP_SET_BUZZER: controls the buzzer. Valid args are:
  - — ICTP_BUZZER_ON
  - — ICTP_BUZZER_OFF guess, what they are doing!
- ICTP_GET_BUZZER: read the current buzzer state.

Collected Adventures of Writing a Linux Device Driver

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

Trieste, 7 Oct–1 Nov 1996

Ulrich Raich
CERN - European Organisation for Nuclear Research
P.S. Division
CH–1211 Geneva
Switzerland.

email: Ulrich.Raich@cern.ch

Collected Adventures of Writing a Linux Device Driver

1 Introduction

It may be best to tell you right from the beginning:

Device Driver Writing is a tricky business!

This in fact was the first thing I had to learn myself when preparing this series of lectures. I was very proud when I was attributed the course on device driver writing by ICTP because this subject has the reputation of being rather difficult. So I was thinking of a course explaining

- all the complicated data structures needed order to hook up the device driver with the kernel
- the context switch from user to supervisor mode with all its details
- lots of computer science theory of why device drivers are important
- and of course all the details of interrupt and DMA driven device drivers
- connection of file system with block device drivers etc.

I think you see what I mean. "*The Theory of Device Driver Writing*" might have been the right title. Then I had the splendid idea that it might be good to actually write a driver myself before trying to explain how to do it. That was the moment when everything started to go wrong! I started the project some 3 months before the course started and 2 weeks before I had to give my lectures the driver still did not work! (and of course the transparencies were not prepared either!). The goals of my lectures became much more modest and I ended up with a course that tells you about all the mischieves I encountered when trying to implement my device driver. No theory! No block device drivers and file systems. Just the story of how I finally managed to get my device driver going. Nevertheless (or perhaps just because the course is now much simpler!) I hope that you will get some insight of

- what a device driver is
- how it works
- and how it is connected to the operating system.

And the best thing of all:

- You get the full source code of the driver
- you can use it
- and you may modify it as ever you like (taking the risk of crashing the system)

2 Generalities

What exactly is a device driver? You can say, a device driver is a software module that manages some hardware device. The operating system then consists of the scheduler (which may be considered a very special device driver managing the CPU), memory management, file system access and a collection of device drivers which are controlled by the operating system kernel.

Due to the different device categories

- block oriented devices like disks
- character oriented devices like ADCs, DACs etc (Terminals are somewhat special, because their driver must implement lots of control functions for the different types of terminals)
- high speed serial devices (networks)

we also distinguish different types of device drivers:

- block device drivers, devices that are accessed only in fixed blocks. Access to these devices pass through buffers in memory (buffer cache). Block device drivers als have hooks for the file system.
- character device drivers providing sequential access. This type is the easiest to implement and it is the only category we will have a closer look to.
- network device drivers are a special character device drivers which must provide very high access speed. Networks transfer their data in packets. For efficiency reasons these drivers have a different programmer interface (you won't find /dev/net or /dev/eth0 (see later)).

From now on we will forget about the more complicated types of devices drivers and have a look exclusively at character oriented drivers. Before going into medias res some more general remarks:

One of the most important commandments of software engineering is:

Before coding you should think !

This is particularly true for writing system level software. So before sitting in front of your computer and calling the editor you should carefully plan your driver. The following questions must be answered before writing the first line of code:

- What functions do you want to implement
- How does the equipment work? (the ICTP board)
- How do you interface it ? (the I/O board and the ICTP / I/O board interface and the cabling)
- What is the speed needed? (In our case we don't want to miss pulses on the 100 Hz clock)
- Do you need interrupts or even DMA?
- Do you have all the documentation needed? (equipment and interface software, description of the library calls that may be used within system level software)
- How does the application programmer use your driver?
- With all that: Write a software specification document (which may later on become the final driver documentation)

So lets start to work through the above list: Getting the hardware documents is not an easy business because companies selling the hardware are usually very sparse in distributing documentation. However I found in my documents at least a list on IO addresses with the interfaces connected.

When accessing any type of hardware several problems arise:

- Firstly there are many people who understand often rather complex electronics that make up computer interfaces and there are many people who write splendid software. Finding somebody who understands the operating system writes nicely structured and very robust code (software) and who is capable to read circuit diagrams, understands timing signals and can read datasheets of electronic chips is already a different business. It is therefore reasonable to isolate the code that needs to know all about the intricacies of the hardware and which in addition must be extremely robust (a small bug can bring the whole system down!) into a separate module. Think of a disk driver for example. This module is written and thoroughly tested once and can then be used by everybody.

- In a multi tasking system several concurrent tasks may want to use the same resource, e.g. a line printer and generate a mess!
- As already mentionned above, hardware access should only be given to trusted users since an error may easily blow the whole system. This is particularly true for multi user systems.
- Access to fixed memory locations is needed e.g. registers or memory in an interface or even specialized I/O instructions. When a device is capable of generating interrupts or of performing DMA then the story becomes even more complex: For interrupts the program context is changed: A new stack frame is in use and the CPU running mode is changed from user mode to supervisor mode. Therefore device drivers must very tightly cooperate with the operation system kernel.

3 Testing the Hardware

Going through the problems one by one I decided that understanding the hardware was first priority. So, what hardware should I use for my demo device driver? At ICTP the only easily available hardware was the Colombo board, which has been designed for the course 82 in Colombo (Sri Lanka). This board actually consists of 2 parts: a processor/memory/interface part (which in our case will be permanently disabled and which I will describe any further) and a part simulating some sort of external process to be controlled. This I/O part consists of

- 4 hex seven segment displays (BCD displays in the original version)
- toggle buttons
- 2 pushbutton switches
- a rotary switch with 16 positions
- a voltage to frequency converter, allowing to simulate a simple ADC
- 3 fixed frequencies
- a buzzer

The pushbuttons, the voltage to frequency converter and the fixed frequencies may be used to generate interrupts.

This board was designed to be hooked up to a PIA (Motorola M6821, Parallel Interface Adapter) in a M6809 development system with the processor part disabled. Development of programs could then be easily done on the development system using its assembler/linker/loader/debugger. Once everthing worked fine a few addresses needed to be changed and the program was blown into EPROM

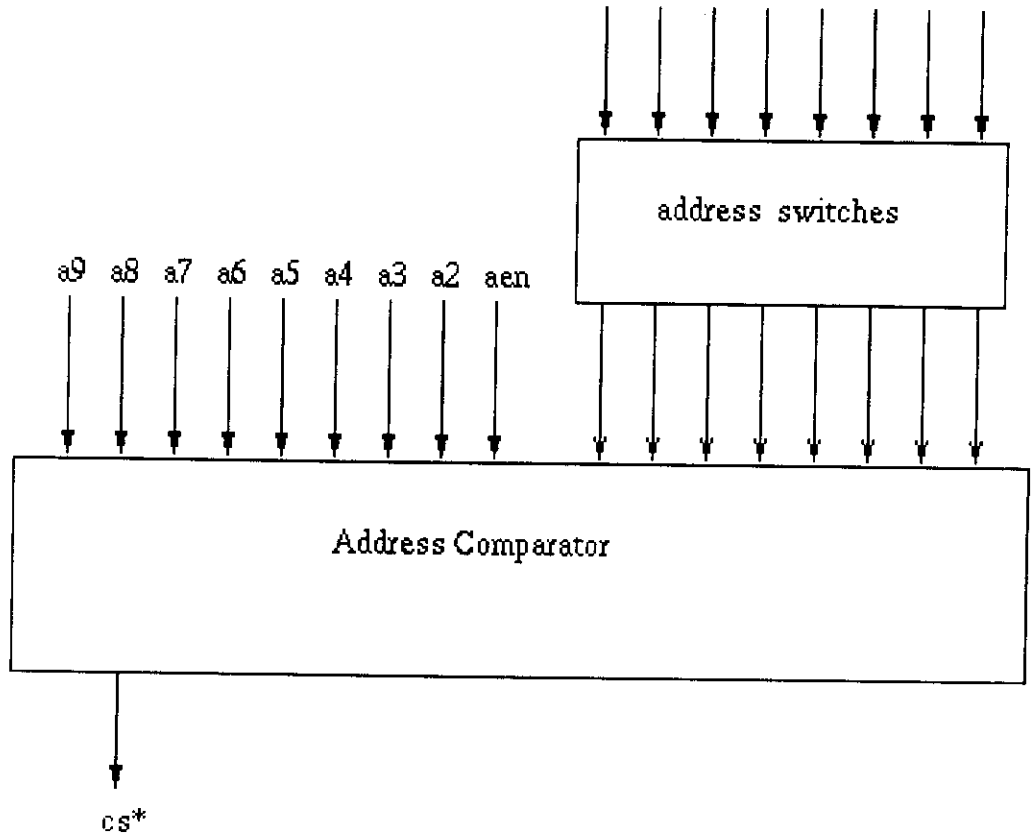


Figure 1: Address Selection

and installed on the Colombo board. Enabling the processor part allowed to run the system in "standalone".

As I said before, from now on we are only interested in the I/O part of the Colombo board. In order to use it for the device driver I still needed an interface for it on the PC (PIA equivalent). I therefore tried to use the lineprinter interface available on most PCs and I a m ged to get the displays going. However the number of I/O lines especially for input were simply not sufficient and also demonstration of interrupts turned out to be impossible. At the same time I learned that a parallel I/O board had been developped at LIP in Portugal

The I/O board consists of an Intel 8255 parallel input/output port, known under the name Programmable Peripheral Interface (PPI) and some interface logic connecting the 8255 to the PC bus. An 75LS682 comparator chip is used

A0	A1	Function
0	0	Port A
0	1	Port B
1	0	Port C
1	1	(write only Control Register

Table 1: Addressing the 8255 chip

for I/O address selection in conjunction with 8 dual inline switches (chip U3 in the circuit diagram)

The 8255 has got 2 general purpose 8 bit ports which may be configured as input or output port (Ports A and B). In addition there are 8 more I/O lines which may take over the function of additional input or output bits (4 bit ports) or may be used as handshake lines, depending on the (software) configuration of the chip. From the programmers point of you these ports are presented as 4 registers whose address layout are shown in the following table:

Before communicating with the outside world (the Colombo board in our case) we must configure the 8255 telling it, which port will be used for output and wich port for input. In addition we need to specify the type of transfer (latched or not). This is done by programming the control register.

Mode 0: Basic I/O mode (non latched). No interrupts

Mode1: Strobed I/O mode. Here some lines of port C will be used as handshake (strobe) lines and may be used to generate interrupts

Mode 2: Bidirectional mode

How does this map to our equipment hardware? Firstly we need 8 output lines in order to drive the displays. 4 lines are used for the data while the other 4 lines will generate the chip select signals for the registers holding the data of each display.

Then we need 8 input lines for reading of the rotary switch and the toggle and pushbutton switches. I ended up having: The hardware was complete and testing could start. I collected all information I was given with my PC and I found a table of hardware addresses, telling me that the address reserved for lpt2: was free.

I therefore set the address switches to ... I opened my PC (for the first time!) I broke out the metal protection for the I/O slot and I inserted the I/O card

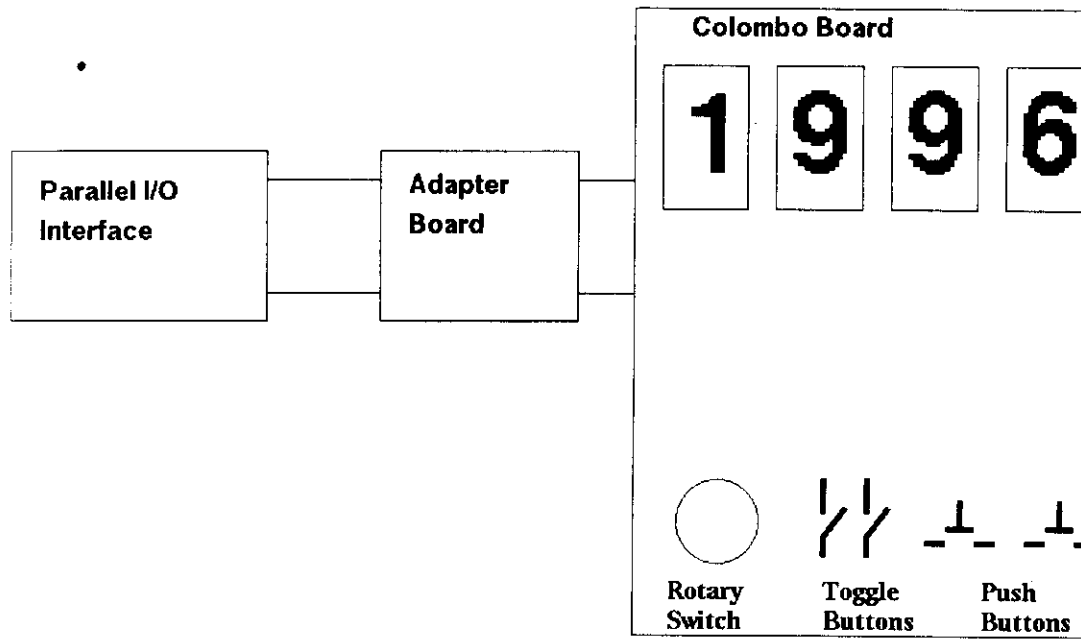


Figure 2: The hardware setup

I/O Addresses	Device
000-01F0	DMA Controller 1
020-03F	Interrupt Controller 1
040-05F	Timer
060-06F	Keyboard
070-07F	Realtime Clock
...	some left out
1F0-1F0	Fixed Disk
278-2FF	Parallel Printer Port 2
2F8-2FF	Serial Port 2
300-31F	Protytype Card
378-3FF	Parallel Printer Port 1
3F0-3F7	Floppy Disk Controller

Table 2: I/O Addresses on the PC Bus

into the slot I had selected. I hooked up the cables and the decisive moment has come! I felt quite nervous! Another serious check and...

I switched the PC **on !**

Oufff, there was no smoke coming out of the PC and the thing booted normally. However I quickly found out that the floppy did not work any more. Did I finally kill the floppy interface?. I re-opened the PC (and decided to leave it open until everything would work or I had to take the PC to the repair shop) took the I/O card out and tried booting again. The floppy worked fine again. So there must have been an address clash between the floppy and the I/O board. (I still don't really know how this comes about!) Looking through the addresses again I selected 0x300 (which turned out to confict with the ethernet card in Trieste, were we finally selected 0x310) which was marked "prototype board". I re-inserted the board and rebooted. This time the machine booted fine again and also the floppy worked normally. Real progress! However I was still unable to talk to the I/O board.

Having a look at the xclock program told me that midnight had passed again! and this was not the first time during the last week. My wife would be angry with me and I tried to find an excuse knowing I would have a hard time with that. The problem was, that she was somehow right but there was so little time left before the course and I absolutely wanted to get the thing going before. Still it was a wise decision to stop at that moment. The next night (I did all this after working hours) I had another look at the addressing on the I/O board. I controlled the address switches again and they seemed all ok. The only possible source of error was the enable line (en) which was marked without a "~" meaning: active high signal. Mostly enable lines are active low however, so I decided that this must have been simply a misprint and I switched the comparator input to active low. It turned out, that I was right. The PC booted, the floppy worked and I was able to talk to the I/O board.

This was not the end of the adventure but clearly the end of the first chapter of this novel. We need to become a little more technical now. You may have asked yourself: What do I mean by "talking to the board". How do I know if I was able to access the board or not. Which calls are provided within Linux to access external hardware? The main problem of checking the interface for the first time is to disentangle hardware and software problems. It is therefore important to get an indication that "something workes". I tried to find a register that I could just read for the first step e.g. a status register on the I/O card. Finding a "reasonable" bit combination could be a first indication of successful board access. After that I tried to find a read/write register which I could write with a known bit pattern and read back. Typical bit patterns are : 0x55 and 0xaa

(Why?) [1] Comparing written to read values gives you a fair idea if things are working, at least if you are successful. Of course checking out the hardware also means writing of very small and simple programs. After having checked read and write access to the interface I wrote a little program that gives a visible indication of something happening on the connected hardware, which was a routine lighting the seven segment displays with known numbers. The program is given below.

```
/*****************************************************/
/* Try to run the colombo board from the parallel    */
/* interface                                         */
/* U. Raich 14.3.94                                  */
/*****************************************************/
#include "/usr/include/stdio.h"
#include "/usr/include/fcntl.h"
#include <asm/io.h>
#include <asm/segment.h>
#include <asm/system.h>
#define ICTP_A 0x300
#define ICTP_B 0x301
#define ICTP_C 0x302
#define ICTP_S 0x303
/*
 * defines for 8255 control port
 * base + 2
 * accessed with LP_C(minor)
 */
#define ICTP_MODE_SELECT    0x80
#define ICTP_A_MODE_0       0x00
#define ICTP_A_MODE_1       0x20
#define ICTP_A_MODE_2       0x40
#define ICTP_B_MODE_0       0x00
#define ICTP_B_MODE_1       0x04
#define ICTP_INPUT_A        0x10
#define ICTP_OUTPUT_A       0x00
#define ICTP_INPUT_B        0x02
#define ICTP_OUTPUT_B       0x00
#define ICTP_INPUT_C_LOW    0x01
#define ICTP_OUTPUT_C_LOW   0x00
#define ICTP_INPUT_C_HIGH   0x04
#define ICTP_OUTPUT_C_HIGH  0x00
int          ioperm();
unsigned int sleep();
void main()
```

[1] Try to write the numbers in binary. You will see that every other bit is set starting from bit 0 for 0x55 and bit 1 for 0xaa.

```
{
        unsigned char command;
        unsigned char test_val;
        int           i;
/*
  get permission to access the I/O port
*/
        if (ioperm(ICTP_A,4,1)) {
          printf("ICTP cannot get permission \n");
          exit(-1);
        }
        command = ICTP_MODE_SELECT | ICTP_A_MODE_1 |
ICTP_B_MODE_1 |                      ICTP_INPUT_B;
        /* reset value */
        outb(command,(unsigned short)ICTP_S);
/*
   write to the displays
*/
        outb((char)0x3f,(unsigned short)ICTP_A);
        outb((char)0x3e,(unsigned short)ICTP_A);
        outb((char)0x3f,(unsigned short)ICTP_A);
        outb((char)0x2f,(unsigned short)ICTP_A);
        outb((char)0x2d,(unsigned short)ICTP_A);
        outb((char)0x2f,(unsigned short)ICTP_A);

        outb((char)0x1f,(unsigned short)ICTP_A);
        outb((char)0x1b,(unsigned short)ICTP_A);
        outb((char)0x1f,(unsigned short)ICTP_A);

        outb((char)0x0f,(unsigned short)ICTP_A);
        outb((char)0x07,(unsigned short)ICTP_A);
        outb((char)0x0f,(unsigned short)ICTP_A);
        exit(0);
}
```

It consists of two parts: Firstly permission is asked for reading and writing to/from absolute I/O addresses (the addresses of the parallel interface registers). As you might expect in a multitasking and multiuser system access to absolute addresses must be restricted (and they are restricted to the super user only) in order to guarantee system integrity. (A super user is supposed to know what he is doing!) After successful execution of *io_perm*() we have access to our I/O interface registers

The sequence therefore is:

ioperm(base_address,range,permission);

value = inb(IO_PortAddress);

outb(IO_PortAddress,value);

A similar sequence is available for memory access, e.g. if you want to write into the video memory of a video card directly. Here we would:

open(/dev/mem);

allocate a certain number of memory pages

and map this memory onto the absolute address of the video memory using mmap.

inb,outb etc. are Macros, which are defined in */usr/include/linux/asm/io.h* (please have a look at this file !) together with similar ones like *inw, inl* etc. Since these are "builtin macros" you **must** use the gcc option -O2 in order to get them included into your code.

```
gcc -O2 -o iotest iotest.c
```

Forgetting -O2 results in unresolved references at link time. After having checked output to the displays we should also check the input connections by reading from the switches:

```
/*
   the #defines are the same as in the writing program
*/
int ioperm();

void main()
```

```
{

        unsigned char command;
        unsigned char value;

        if (ioperm(ICTP_A,4,1)) {
          printf("ICTP cannot get permission \n");
          exit(-1);
        }

        command = ICTP_MODE_0 | ICTP_MODE_SELECT |
ICTP_INPUT_B;

        for (;;) {
          value = inb(ICTP_B);
          printf("Value read:  %x\n",value);
          sleep(1);
        }
        exit(0);
}
```

4 Accessing a device driver

You may think, since we now have access to our I/O card, we can read and write data form/to it, well ..., that's it, we have finished! Unfortunately this is not the case. As said before: Any program making use of *ioperm, inb, outb* or *mmap* will only run in super user mode.We want to give access to our board to the ordinary user however. In addition there is no resource protection (the board may be written to by several tasks in any wild sequence) and treatment of interrupts or even DMA are excluded. Only the device driver will give you access to these possibilities.

What exactly is a device driver then? and how may an ordinary user access it? We want to slowly approach this question by first looking at the drivers software interface, or said differently: the way a programmer would use the driver.

You have already written programs make use of files. You have seen the calls:

- open
- close
- read
- write

- lseek etc.

Accessing a device driver is exactly the same. You may think of a device as a **special file** (which is actually the technical term for it). The device is accessed through inodes defined in /dev using the same calls as normal file access. On order to open the device of our Colombo board we would write:

```
fd = open("/dev/ictp0",O_RDWR);
```

In order to write to the board we would fill a buffer and write it to the board:

```
buf[0] = 0x3f; /* fill the buffer */
buf[1] = 0x3e;
buf[2] = 0x3f;
write(fd, buf, 3); /* write to the board */
```

What exactly happens when we access the driver? The "calls" open, read, write are so-called system calls and differ from normal subroutine calls. System calls generate software interrupts and doing so change the running mode from (normal) user mode to supervisor mode. After that a subroutine within the system kernel is called and executed in supervisor mode (compare to fig.) You now immediately see that our driver routines are actually executed on the same level as the kernel, they are integral part of the kernel. This also means that errors within the kernel routines are usually unrecoverable (there is no such things as "segmentaion fault, core dumped") and will crash the entire system.

The following listing gives an example of access to the ictp driver.

```
/*****************************************************/
/* Access the ictp device driver                     */
/* This example writes the displays in RAW mode      */
/* U. Raich 14.3.94                                  */
/*****************************************************/
#include "/usr/include/stdio.h"
#include "/usr/include/fcntl.h"
#include <sys/ioctl.h>
#include "ictp.h"

void main()
{
  int fd,i,ret_code;
  unsigned long mode;
  unsigned char buffer[12];
  short    full_number;

/*
  open the device driver for writing
```

```
*/
  fd = open("/dev/ictp0",O_WRONLY);
  if (fd <  0) {
    perror ("Could not open ictp port:");
    exit(-1);
  }
  else
   printf("ictp port successfully opened for
writing!\n");


/*
  try out raw mode
  we must code data and chip select signals ourselves

    buf(0) buf(1) buf(2)  4 least significant bits
    ------       ---------------
          |------|
*/

  buffer[0]=0x1f;     /* 1 -> data lines; strobes all
high */
  buffer[1]=0x17;     /* 1 -> data lines; strobe 1
active low */
  buffer[2]=0x1f;     /* 1 -> data lines; strobes all
high */
  buffer[3]=0x2f;     /* same for displays 2-4 */
  buffer[4]=0x2b;
  buffer[5]=0x2f;
  buffer[6]=0x3f;
  buffer[7]=0x3d;
  buffer[8]=0x3f;
  buffer[9]=0x4f;
  buffer[10]=0x4e;
  buffer[11]=0x4f;

  if (write(fd,buffer,12) != 12)
    perror("after write ");

  close(fd);
}
```

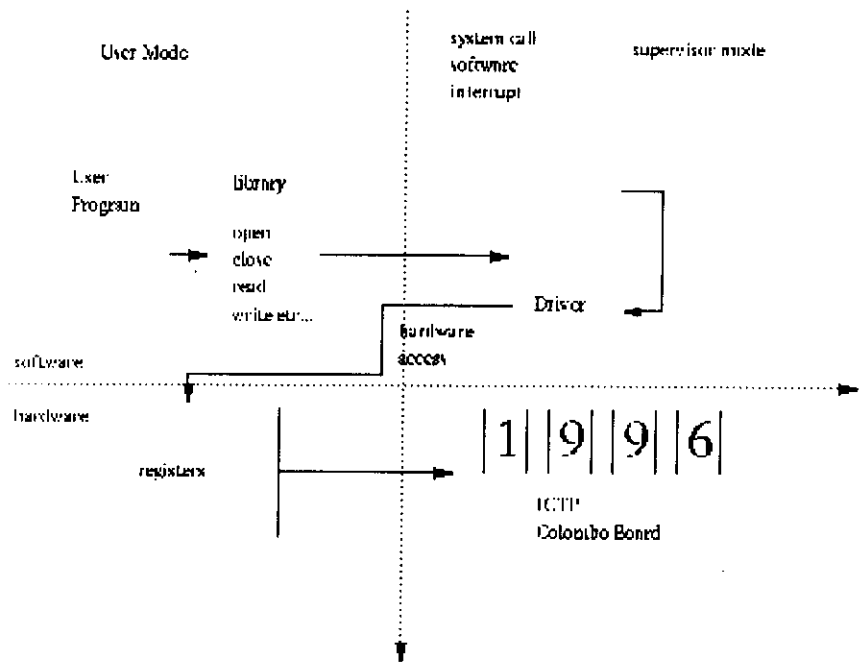


Figure 3: Accessing the Device Driver

5 Representation of the device driver in the system

Having seen how to access the driver from an application we must now figure out how the kernel finds its way into the driver. Trying:

*ls -l /dev/ictp**

will produce the following output:

```
crw-rw-rw-  1 root    root     31,  0 Jun  5 22:21 /dev/ictp0
crw-rw-rw-  1 root    root     31,  1 Jun  5 22:21 /dev/ictp1
crw-rw-rw-  1 root    root     31,  2 Jun  5 22:21 /dev/ictp2
```

where c tells us that the file is actually a character device driver , rw are the usual read and write permission bits and 31 is the major and 0,1,2 the minor device numbers. These numbers are unique in the system. By the way: the device special files are not created by and editor but by the command:

mknod /dev/ictp0 c 31 0

mknod /dev/drivername device type major number minor number

The major number defines the I/O device, the minor number usually indicates a channel number (a serial I/O device may have 4 UARTs representing 4 serial I/O channels, which a driven by a single software module)

As explained before, the driver is an integral part of the operation system and is usually linked into the kernel during system generation. However a software package has been developped for Linux, allowing us to install and de-install device drivers (or other "modules" like file systems etc) into a running kernel.

This **modules package** provides the following basic programs:

- *insmod*: install a module into the kernel
- *lsmod* : list all installed modules
- *rmmod*: remove a modules from the kernel
- *ksyms*: list exported kernel symbols

and the newer versions have in addition:

- *modprobe*: same as insmod but a standard path is searched for the modules while insmod needs the full pathname of the module to be installed
- *genksyms*
- *kerneld*: kernel daemon, allows demand loading of modules.

The system you are currently using has all its loadable modules installed in /lib/modules/current. This is a symbolic link to /lib/modules/linux-version (2.0.20), which is created at boot time. Some modules may be installed in /lib/modules/current/boot and will be automatically loaded and permanently installed at boot time. When you try to access a module (e.g. the ictp driver) that is not installed in the system, the kernel will ask *kerneld* to load the module. *kerneld* will look at strategic places (possibly defined in /etc/conf/modules) and if the corresponding module can be located it will try to load it. If loading is successful, the calling program will simply continue.

6 Implementing the Device Driver, first steps

A device driver always consists of at least 2 files:

- The driver include file (ictp.h in our case)
- and the driver code itself

The include file contains the definitions of hardware addresses, register names, and names for each and every bit used within the IO chip registers. In addition it contains definitions for error codes, names for driver operating modes, ioctl request names and thelike. It is used by the driver itself, but is is usually also included by any program using the driver.

The following listing shows the include file provided for the ictp driver:

```
/*****************************************************/
/* Definitions of 8255 addresses and control bits    */
/* U. Raich 31.8.94                                  */
/*****************************************************/

#include <sys/ioctl.h>

#define ICTP_MAJOR              31
#define ICTP_NO                  3

/*
 * defines for 8255 ports
 */

#define ICTP_A 0x300
#define ICTP_B 0x301
#define ICTP_C 0x302
#define ICTP_S 0x303
```

```
/*
 * defines ICTP status and control register bits
 */

#define ICTP_MODE_SELECT    0x80
#define ICTP_A_MODE_0       0x00
#define ICTP_A_MODE_1       0x20
#define ICTP_A_MODE_2       0x40

#define ICTP_B_MODE_0       0x00
#define ICTP_B_MODE_1       0x04

#define ICTP_MODE_BLOCKING     0
#define ICTP_MODE_NON_BLOCKING 1

#define ICTP_INPUT_A        0x10
#define ICTP_OUTPUT_A       0x00
#define ICTP_INPUT_B        0x02
#define ICTP_OUTPUT_B       0x00
#define ICTP_INPUT_C_LOW    0x01
#define ICTP_OUTPUT_C_LOW   0x00
#define ICTP_INPUT_C_HIGH   0x08
#define ICTP_OUTPUT_C_HIGH  0x00

#define ICTP_AVAILABLE      1
#define ICTP_NOT_AVAILABLE  0

#define ICTP_SILENCE        0x09
#define ICTP_NOISE          0x08
#define ICTP_BUZZER_BIT     0x10
#define ICTP_BUZZER_ON         1
#define ICTP_BUZZER_OFF        0

#define ICTP_MODE_RAW          0
#define ICTP_MODE_SINGLE_DIGIT 1
#define ICTP_MODE_FULL_NUMBER  2

#define ICTP_BUSY              1
#define ICTP_FREE              0

#define ICTP_READ_SWITCHES     0
#define ICTP_READ_IRQ5_COUNT   1
#define ICTP_READ_IRQ7_COUNT   2

#define ICTP_ENABLE_IRQ5       0x5
#define ICTP_DISABLE_IRQ5      0x4
```

```
#define ICTP_ENABLE_IRQ7        0xd
#define ICTP_DISABLE_IRQ7       0xc

#define ICTP_IBF_B              0x2
#define ICTP_DUMMY              0xff
/*
  the ioctl codes:
*/
#define ICTP_SET_WRITE_MODE     IOC_IN  | 0x0001
#define ICTP_GET_WRITE_MODE     IOC_OUT | 0x0001
#define ICTP_SET_READ_MODE      IOC_IN  | 0x0002
#define ICTP_GET_READ_MODE      IOC_OUT | 0x0002
#define ICTP_SET_BUZZER         IOC_IN  | 0x0003
#define ICTP_GET_BUZZER         IOC_OUT | 0x0004
#define ICTP_ENABLE_INTERRUPT             0x0005
#define ICTP_DISABLE_INTERRUPT            0x0006

/*
  for checking if the board is there
*/
#define ICTP_TSTBIT           0x20
#define ICTP_SET_TSTBIT       0xb
#define ICTP_RESET_TSTBIT     0xa

#define ICTP_MAX_DIGIT         3
#define ICTP_MAX_BUFFER       12
```

Before a user program can access the driver it must be included into the operating system. As already mentionned above this can be done by either linking it to the kernel at system creation time or we register the driver with the operating system once the module containing the driver gets installed with insmod. Therefore we must write 2 routines

- *init_module*, which is called by *insmod* and which will check if the parallel IO card can be accessed before asking the kernel to register the ictp device driver.
- *cleanup_module*, which is called by *rmmod* and which cleanly removes the driver from the system.

Since the driver it an integral part of the operating system and works in supervisor mode, it has no access to the normal C library functions. It cannot be debugged with the normal debugger neither (the debugger has no access to supervisor memory!). However a few calls are available to the device driver writer, one of which is *printk* which is the kernel equivalent (though less powerful) of *printf*.

For debugging purposes I therefore put a few statements at strategic places in order to be able to follow the execution of my code.

When registering the device driver with a system the address of the *fops* table is passed as a parameter. This table contains the the addresses of the driver routines needed for execution of the

- open
- close
- read
- write
- lseek
- ioctl

and a few more system calls. If a call is not implemented, the table gets a NULL entry. Here is the *fops* table of our ictp driver:

```
static struct file_operations ictp_fops = {
    NULL,               /* seek       */
    ictp_read,
    ictp_write,
    NULL,               /* readdir */
    NULL,               /* select  */
    ictp_ioctl,
    NULL,               /* mmap */
    ictp_open,
    ictp_release
};
```

The code for *init_module* is also given below:

```
/*
 * And now the modules code and kernel interface.
 */

int
init_module( void) {

        unsigned char testvalue = 0;

#ifdef ICTP_DEBUG
        printk(KERN_DEBUG "ictp:   init_module
called\n");
```

```
#endif

/*
   initialize the chip
*/
        ictp_reset();

        testvalue = inb(ICTP_B);
/*
   set bit 5 of port C and read back.  This bit is
unused
*/

        outb(ICTP_SET_TSTBIT,ICTP_S);
        testvalue = inb(ICTP_C);
#ifdef ICTP_DEBUG
        printk(KERN_DEBUG "ictp:  port C after set bit 5
%x\n",testvalue);
#endif
        if ((testvalue & ICTP_TSTBIT) == 0) {
          printk(KERN_ERR "ictp:  board not found!\n");
          return -ENODEV;
        }
        outb(ICTP_RESET_TSTBIT,ICTP_S);
        testvalue = inb(ICTP_C);
#ifdef ICTP_DEBUG
        printk(KERN_DEBUG "ictp:  port C after reset bit
5 %x\n",testvalue);
#endif
        if ((testvalue & ICTP_TSTBIT) != 0) {
          printk(KERN_ERR "ictp:  board not found!\n");
          return -ENODEV;
        }

/*
  register the device driver with the system
*/
       if (register_chrdev(HW_MAJOR, "ictp",
&ictp_fops)) {
         printk(KERN_ERR "register_chrdev failed:
goodbye world :-(\n");
          return -EIO;
        }
```

```
#ifdef ICTP_DEBUG
        else
          printk(KERN_DEBUG "ictp:  driver
registered!\n");
#endif
        return 0;
}
```

The first version of the driver registered a fops table with NULL entries only. This version clearly cannot do anything, however it should be possible to test installation and deinstallation into the system using *insmod* and *lsmod*. I expected to find the *printk* output on the xconsole and lsmod should allow be to check proper installation.

What did I find? Well, it was the worst possible options: lsmod told me it had fould the module:

```
Module:          #pages:   Used by:
ictp               1           0
```

but I had no trace whatsoever of my printk statements. I was not really sure who was right: *lsmod* or the missing output from *printk*. After several hours of research and some poking around on the network I found the email address of a *guru* who had written a device driver before. He told be I could check where to find the system console by trying the command:

```
date > /dev/console
```

When I tried this, I found the output of date on the xconsole as expected. Still I did not know where the *printk* output has gone. The only other test I could find was to recompile the kernel and link my device driver into the system. When booting the newly constructed system I saw the very first *printk* output on the system console but not the following ones. Of course I then checked the system log (*/var/log/messages*) in order to figure out what had happened and there I found all the output I had expected on the system console. The *printk* output had been written to the system log file! After having added the lines

```
# Send debug messages to the console
*.=debug /dev/console
```

to */etc/syslog.conf* I finally got the debugging messages where I wanted them to go, namely on the system console (the *xconsole* window).

7 The Driver Routines

Since we now

- understand the hardware

- know how to unstall the device driver into the system
- have the frame of our driver ready
- are able to produce debugging messages

we can actually start to implement the first driver routines that do the real work. The first routines to be implemented are of course the open and the close routines. When opening the device driver we initialize the 8255 chip writing the necessary bits into the command register.

In order to make sure that only one process at a time can access the device driver a busy flag is set when the driver is opened for the first time. All subsequent open requests will be refused (giving back the error -EBUSY) until the driver is closed again.

```
static int
ictp_open(struct inode * inode, struct file * file)
{
   unsigned char     command;
   int               ret_code;

   if (ictp_busy ==  ICTP_BUSY)
     return -EBUSY;

   ictp_busy         = ICTP_BUSY;
   ictp_write_mode   = ICTP_MODE_RAW;

   command = ICTP_MODE_SELECT | ICTP_A_MODE_0 |
ICTP_B_MODE_0
                                    | ICTP_INPUT_B;
   outb(command,ICTP_S);            /* setup to non
interrupt */

#ifdef ICTP_DEBUG
   printk(KERN_DEBUG "ictp:  opened for switch
reading\n");
#endif
   return 0;
}
```

The above code is a simplified version of the code actually in service for the ictp driver. In the real open routine we also switch off the buzzer and, depending on the selected minor mode (ictp0, ictp1 or ictp2), we also register interrupt service routines with the system.

In order to implement the write part of the driver we should first have a look at the library routines accessible to the device driver writer. Some of these routines we have already seen before, namely:

- register_chrdev(unsigned int major, const char *name, struct file_operations *fops)
- unregister_chrdev(unsigned int major, const char *name)
- printf(fmt)

There are also 2 macros that allow us to find out the current major and minor numbers:

- MAJOR(inode -> i_rdev) and
- MINOR(inode -> i_rdev)

As we have seen in the example code above, inode is a structure that is passed into the driver routines. In order to implement the read and write routines we need additional calls that allow us to transfer a data buffer from user space into supervisory space and back. This feature is provided by:

- char get_user(char *address)
- void put_user(char, char *address)

Their use is demonstrated by the (incomplete) ictp write routine:

```
/*
 * Write requests on the ictp device.
 */
static int
ictp_write(struct inode * inode, struct file * file,
           const char * buf, int count)
{

        char          c;
        const char    *temp=buf;
        unsigned char ctemp, digit;

        switch (ictp_write_mode) {
        case ICTP_MODE_RAW:
                    temp = buf;
                    if (count>ICTP_MAX_BUFFER)
                      return -EINVAL;
                    while (count > 0) {
                      c = get_user(temp);
                      outb(c,ICTP_A);
                      count--;
                      temp++;
                    }

                    return temp-buf;
                    break;
```

We have now seen the *open,close,write* routines (the read is very similar to the write once get_user has been replaced by put_user). The only missing code is the *ioctl*.

As a typical example we will have a look at the code that drives the buzzer. The buzzer is connected to the PC-4 line of the 8255 and can be programmed by specifying the bit number in bits 1-3 of the 8255 command register with bit 7 set to 0 and bit 0 defining on (bit 0 = 1) or off (bit 0 = 0). The ioctl call as seen from the driver users point of view has got 3 parameter: the file descriptior, a command code (defined in ictp.h: #define ICTP_SET_BUZZER IOC_OUT | 0x0004).

```
ioctl(ictp_fd,ICTP_SET_BUZZER,ICTP_NOISE)
```

will switch the buzzer on while

```
ioctl(ictp_fd,ICTP_SET_BUZZER,ICTP_SILENCE)
```

will switch it off again.

The command parameter in the driver code receives the ioctl command code while in arg the corresponding argument (buzzer on or off) is passed.

```
/*
 * Handle ioctl calls
 */

static int
ictp_ioctl(struct inode * inode, struct file * file,
           unsigned int cmd, unsigned long arg)
{
   unsigned int minor = MINOR(inode->i_rdev);
   unsigned char port_C_status;
   unsigned short dummy;
   switch (cmd) {
   case ICTP_SET_BUZZER:
#ifdef ICTP_DEBUG
     printk(KERN_DEBUG "ictp:  ioctl set buzzer function
entered!\n");
```

```
#endif
      if (arg == ICTP_BUZZER_ON) {
        outb(ICTP_NOISE,ICTP_S);
        return 0;
      }
      else if (arg == ICTP_BUZZER_OFF) {
        outb(ICTP_SILENCE,ICTP_S);
        return 0;
      }
      else
        return -EINVAL;
      break;

   case ICTP_GET_BUZZER:
#ifdef ICTP_DEBUG
      printk(KERN_DEBUG "ictp:  ioctl get buzzer function
entered!\n");
#endif
      port_C_status = inb(ICTP_C);
      if (port_C_status & ICTP_BUZZER_BIT)
        return ICTP_BUZZER_OFF;
      else
        return ICTP_BUZZER_ON;
      break;

   default:  return -EINVAL;
        }
}
```

The driver allows users to choose a write mode forcing subsequent write calls to be interpreted in different ways.

- ICTP_MODE_RAW will send the data transfered in the write's databuffer as is to the hardware. In this mode the driver user is responsible to set up the data and chip-select lines correctly.
- ICTP_MODE_SINGLE_DIGIT will write a single digit. Here a single byte containing the digit number in the high nibble and the data value in the low nibble
- ICTP_MODE_FULL_NUMBER takes a short containing the number to be displayed on all four digits.

The full ioctl driver code permits the user to set the write mode using the ICTP_SET_WRITE_MODE ioctl command. Several other commands are implemented for

- Enabling/disabling interrupts

- Setting the interrupt type (blocking or non blocking)
- Reading the buzzer state
- Reading the write mode

This terminates our excursion into the world of systems programmers. The Appendices contain the circuit diagram of the parallel IO card, a "user manual" and the full driver code.

8 Appendix

9 The User Manual

This section describes the functionality the driver supplies to its users.

The ICTP device driver expects an I/O board using an Intel 8255 chip at I/O address 0x300 (can be changed in the code) connected to the ICTP Colombo board. The connections to the Colombo board must be made as follows:

- Port A: ICTP displays. Port A is therefore programmed as an *output* port.
- PortB: ICTP switches. Port B is therefore programmed as an *input* port.

If you open minor device 0, the port B of the 8255 wil be set to mode 0 (non strobed input) allowing to read directly the state of the switches. Port A is set to mode 1 (strobed I/O).

When openinn minor device 1 the 8255 chip is initialized such, that port A and Port B are set to mode 1 (strobed I/O). This allows interrupts for both ports.

In mode 1, with port A output, the bits 4 and 5 of port C may be used as normal programmed I/O pins, while the other bits are used as handshake signals or interrupt lines. Bit 4 of port C must be connected to the Colombo board buzzer.

Bit 2 and bit 6 of port C are strobe lines which must be connected to one of the interrupt generating lines CA1. CA2, CB1 of the Colombo board.

The driver functions:

9.1 Read Calls

The driver uses major number 31 and 3 minor devices:

- read on minor number 0: read the switches
- read on minor number 1: read the number of interrupts on IRQ 5 since the last read call
- read on minor number 2: read the number of interrupts on IRQ7 since the last read call

An ioctl call allows the user to switch between blocking and non blocking calls for reads from minor number 1 and 2.

9.2 Write Calls

Writings works on any of the minor devices. There are 3 different write modes which may be set up by ioctl calls:

- ICTP_MODE_RAW: In this mode the data coming from the user are simply passed on to the hardware without any modification. In order to make the displays work the user must send 12 data bytes containing data and chip select information (data: high nibble, chip select: low nibble) in the sequence: data + cs high, data + cs low, data + cs high for each digit.
- ICTP_MODE_SINGLE_DIGIT: a single data byte is expected. This byte must contain the digit number in the high nibble and the data to be written to that digit in th low nibble.
- ICTP_FULL_NUMBER. a short (16 bits) is expected. This number will be written to the 4 digits.

9.3 IOCTL Calls

- ICTP_SET_WRITE_MODE: sets up the mode for future writing to the displays (see above) . The possibilities are ICTP_MODE_RAW, ICTP_MODE_SINGLE_DIGIT and ICTP_MODE_FULL_NUMBER.
- ICTP_GET_WRITE_MODE: returns the current write mode
- ICTP_ENABLE_INTERRUPT enables an interrupt. Possible parameters are ICTP_ENABLE_IRQ5 and ICTP_ENABLE_IRQ7
- ICTP_DISABLE_INTERRUPTS you guess
- ICTP_SET_BUZZER switches the buzzer on or off. Possible parameters are ICTP_NOISE or ICTP_SILENCE
- ICTP_GET_BUZZER gets the current buzzer state.

10 The Driver Code

```
/*
 * Implements the ICTP character device driver.
 * Create the device with:
 *
 * mknod /dev/ictp c 31 0
 *
 * - U. Raich
 * 13.3.94 :  First version working with PC parallel
printer port
 *
 * Modifications:
 * 30.08.94 :  U.R. complete rewrite for Manuel's board
 *  9.10.96 :  U.R. adapted to new Linux version
(2.0.20)
 *                implemented check for MAX_DIGIT and
MAX_BUFFER in order
 *                to make the driver more secure
 */


/* Kernel includes */

#include <linux/module.h>

#include <linux/mm.h>
#include <linux/errno.h>
#include <linux/kernel.h>
#include <linux/major.h>
#include <linux/sched.h>
#include <linux/malloc.h>
#include <linux/ioport.h>
#include <linux/fcntl.h>
#include <linux/delay.h>

#include <asm/io.h>
#include <asm/segment.h>
#include <asm/system.h>

#include "ictp.h"

#define HW_MAJOR    31 /* nice and high */
#define ICTP_DEBUG   1
```

```
/*
   some globals:
*/
unsigned long     ictp_write_mode  = ICTP_MODE_RAW;
unsigned long     ictp_read_mode   =
ICTP_MODE_NON_BLOCKING;
int               ictp_busy        = ICTP_FREE;
int               irq5 = 5, irq7 = 7;
unsigned char     irq5_count = 0, irq7_count = 0;
struct wait_queue *ictp_wait_q;

/*
 * The driver.
 */

static void
out_digit(unsigned char digit, unsigned char number)

{
  unsigned char mask,c;

  mask = 1 << digit;
  mask = ~mask;
#ifdef ICTP_DEBUG
  printk(KERN_DEBUG "ictp:  mask %x\n",mask);
#endif
  c = (number << 4) | 0xf;
  outb(c,ICTP_A);
  c &= mask;
  outb(c,ICTP_A);
  c |= 0xf;
  outb(c,ICTP_A);
}


/*
  first the tough part:  the interrupt code
*/

static void
ictp_irq7_interrupt( int irq)
```

```
 {

   outb(ICTP_DUMMY,ICTP_A);          /* this just clears
the interrupt */

   irq7_count++;
   if (ictp_read_mode == ICTP_MODE_BLOCKING)
     wake_up(&ictp_wait_q);
}

static void
ictp_irq5_interrupt(int irq)
{
  unsigned char dummy;
  dummy = inb(ICTP_B);          /* this just clears the
interrupt */

  if (ictp_read_mode == ICTP_MODE_BLOCKING)
    wake_up(&ictp_wait_q);
  irq5_count++;

}

static void
ictp_reset(void)
/*======================*/

/*
  initializes the 8255 chip
*/

{
       unsigned char command;

/*
  sets port A to output
       port A is connected to the ICTP module displays
       high order nibble:  data
       low  order nibble:  chip selects
 mode 1:  stobed I/O
       allows use of interrupts
       CA1:  on interrups
       CA2:  (Buzzer) on PC4
```

```
*/
        command = ICTP_MODE_SELECT | ICTP_A_MODE_0 |
ICTP_B_MODE_0
                                        | ICTP_INPUT_B;

        outb(command, ICTP_S);
/*
  kill the buzzer
  first setup port C to bit set (bit set/reset mode with
set bit on!  )
*/
        outb(ICTP_SILENCE,ICTP_S);
        return ;
}

/*
 * Handle ioctl calls
 */

static int
ictp_ioctl(struct inode * inode, struct file * file,
           unsigned int cmd, unsigned long arg)
{
   unsigned int minor = MINOR(inode->i_rdev);
   unsigned char port_C_status;
   unsigned short dummy;
   switch (cmd) {
   case ICTP_SET_WRITE_MODE:
#ifdef ICTP_DEBUG
    printk(KERN_DEBUG
           "ictp:  ioctl write function entered!  cmd:
%x, arg:  %lx\n",
           cmd, arg);
#endif
    if (arg > ICTP_MODE_FULL_NUMBER)
         return -EINVAL;
     else {
         ictp_write_mode = arg;
         return 0;
       }
     break;
   case ICTP_GET_WRITE_MODE:
#ifdef ICTP_DEBUG
     printk(KERN_DEBUG "ictp:  ioctl read function
entered!  cmd:  %x\n",cmd);
```

```
#endif
        return ictp_write_mode;
        break;

    case ICTP_SET_READ_MODE:
#ifdef ICTP_DEBUG
      printk(KERN_DEBUG
             "ictp:  ioctl write function entered!  cmd:
%x, arg:  %lx\n",
             cmd, arg);
#endif
      if (arg > ICTP_MODE_NON_BLOCKING)
          return -EINVAL;
       else {
          ictp_read_mode = arg;
          return 0;
        }
       break;
    case ICTP_GET_READ_MODE:
#ifdef ICTP_DEBUG
      printk(KERN_DEBUG "ictp:  ioctl read function
entered!  cmd:  %x\n",cmd);
#endif
      return ictp_read_mode;
      break;

    case ICTP_SET_BUZZER:
#ifdef ICTP_DEBUG
      printk(KERN_DEBUG "ictp:  ioctl set buzzer function
entered!\n");
#endif
      if (arg == ICTP_BUZZER_ON) {
        outb(ICTP_NOISE,ICTP_S);
        return 0;
      }
      else if (arg == ICTP_BUZZER_OFF) {
        outb(ICTP_SILENCE,ICTP_S);
        return 0;
      }
      else
        return -EINVAL;
      break;

    case ICTP_GET_BUZZER:
```

```
#ifdef ICTP_DEBUG
     printk(KERN_DEBUG "ictp:  ioctl get buzzer function
entered!\n");
#endif
     port_C_status = inb(ICTP_C);
     if (port_C_status & ICTP_BUZZER_BIT)
       return ICTP_BUZZER_OFF;
     else
       return ICTP_BUZZER_ON;
     break;

   case ICTP_ENABLE_INTERRUPT:
#ifdef ICTP_DEBUG
     printk(KERN_DEBUG "ictp:  enabling interrupts on
8255\n");
#endif
     if (minor == ICTP_READ_IRQ7_COUNT) {
       dummy = 0xff;
       outb(dummy,ICTP_A);              /* reset int
flag */
       outb(ICTP_ENABLE_IRQ7,ICTP_S);

       irq7_count = 0;
       dummy = inb(ICTP_C);
#ifdef ICTP_DEBUG
       printk(KERN_DEBUG "ictp:  port C after enable int
7:  %x\n",dummy);
#endif
       return 0;
     }
     else if (minor == ICTP_READ_IRQ5_COUNT) {
       outb(ICTP_ENABLE_IRQ5,ICTP_S);
       dummy = inb(ICTP_B);                /* now
interrupts should come in */

       irq5_count = 0;
       dummy = inb(ICTP_C);
#ifdef ICTP_DEBUG
       printk(KERN_DEBUG "ictp:  port C after enable int
5:  %x\n",dummy);
```

```
#endif
        return 0;
      }
      else
        return -EINVAL;
      break;

    case ICTP_DISABLE_INTERRUPT:
#ifdef ICTP_DEBUG
     printk(KERN_DEBUG "ictp:  disabling interrupts on
8255\n");
#endif
      if (minor == ICTP_READ_IRQ7_COUNT) {
        outb(ICTP_ENABLE_IRQ7,ICTP_S);
        return 0;
      }
      else if (minor == ICTP_READ_IRQ5_COUNT) {
        outb(ICTP_ENABLE_IRQ5,ICTP_S);
        return 0;
      }
      else
        return -EINVAL;
      break;

    default:  return -EINVAL;
      }
}
/*
 * Read the status of the ICTP board switches
 */

static int
ictp_read(struct inode * inode, struct file * file,
          char * buf, int count)
{

        unsigned int  minor = MINOR(inode->i_rdev);
        unsigned char testvalue;

        if (count != 1) return -EINVAL;

        switch (minor) {
        case ICTP_READ_SWITCHES:
          testvalue = inb(ICTP_B);        /* read the
switches */
```

```
#ifdef ICTP_DEBUG
          printk(KERN_DEBUG
                 "ictp:  switch value read from port:
%x\n",testvalue);
#endif
          put_user(testvalue,buf);
          return 1;
          break;
        case ICTP_READ_IRQ7_COUNT:
#ifdef ICTP_DEBUG
          printk(KERN_DEBUG "ictp:  irq7_count:
%d\n",irq7_count);
#endif
          if (ictp_read_mode == ICTP_MODE_BLOCKING){
            if (irq7_count == 0) {
#ifdef ICTP_DEBUG
              printk(KERN_DEBUG "ictp:  Going to sleep
...\n");
#endif
              interruptible_sleep_on(&ictp_wait_q);
            }
#ifdef ICTP_DEBUG
            printk(KERN_DEBUG "ictp:  returned from
sleep\n");
#endif
            put_user(irq7_count,buf);
            irq7_count = 0;
          }
          else {
            put_user(irq7_count,buf);
            irq7_count = 0;
          }
          testvalue = inb(ICTP_C);
#ifdef ICTP_DEBUG
          printk(KERN_DEBUG "ictp:  Port C data:
%x\n",testvalue);
#endif
          return 1;
        case ICTP_READ_IRQ5_COUNT:
#ifdef ICTP_DEBUG
          printk(KERN_DEBUG "ictp:  irq5_count:
%d\n",irq5_count);
#endif
          if (ictp_read_mode == ICTP_MODE_BLOCKING){
            if (irq5_count == 0) {
```

```
#ifdef ICTP_DEBUG
                printk(KERN_DEBUG "ictp:  Going to sleep
...\n");
#endif
                interruptible_sleep_on(&ictp_wait_q);
              }
#ifdef ICTP_DEBUG
              printk(KERN_DEBUG "ictp:  returned from
sleep\n");
#endif
              put_user(irq5_count,buf);
              irq5_count = 0;
            }
            else {
              put_user(irq5_count,buf);
              irq5_count = 0;
            }
            testvalue = inb(ICTP_C);
#ifdef ICTP_DEBUG
            printk(KERN_DEBUG "ictp:  Port C data:
%x\n",testvalue);
#endif
            return 1;
          default:
            return -EINVAL;
          }
}

/*
 * Write requests on the ictp device.
 */
static int
ictp_write(struct inode * inode, struct file * file,
           const char * buf, int count)
```

```
{
        char          c;
        const char    *temp=buf;
        unsigned char ctemp, digit;

        switch (ictp_write_mode) {
        case ICTP_MODE_RAW:
                    temp = buf;
                    if (count>ICTP_MAX_BUFFER)
                      return -EINVAL;
                    while (count > 0) {
                      c = get_user(temp);
                      outb(c,ICTP_A);
                      count--;
                      temp++;
                    }

                    return temp-buf;
                    break;
        case ICTP_MODE_SINGLE_DIGIT:
                    if (count != 1)
                      return -EINVAL;
                    c = get_user(temp);
                    digit = c >> 4;
                    if (digit > ICTP_MAX_DIGIT)
                      return -EINVAL;
                    ctemp = c & 0xf;
                    out_digit(digit,ctemp);
                    return 1;
                    break;
        case ICTP_MODE_FULL_NUMBER:
                    if (count != 2)
                      return -EINVAL;
                    temp = buf;
                    c = get_user(temp);
#ifdef ICTP_DEBUG
                    printk(KERN_DEBUG "write, mode 2,
first byte:  %x\n",c);
#endif
                    ctemp = c & 0xf;
                    out_digit(0,ctemp);
                    ctemp = c >> 4;
                    out_digit(1,ctemp);

                    c = get_user(temp+1);
```

```
#ifdef ICTP_DEBUG
                        printk("write, mode 2, second byte:
%x\n",c);
#endif

                        ctemp = c & 0xf;
                        out_digit(2,ctemp);
                        ctemp = c >> 4;
                        out_digit(3,ctemp);
/*
  get first nibble
*/

                        return 2;
                        break;
        default:
                        return 1;
                        break;
                }

}


static int
ictp_open(struct inode * inode, struct file * file)
{
        unsigned int        minor = MINOR(inode->i_rdev);
        unsigned char       command;
        int                 ret_code;

        if (minor >= ICTP_NO)
                return -ENODEV;
        if (ictp_busy ==  ICTP_BUSY)
                return -EBUSY;

        ictp_busy         = ICTP_BUSY;
        ictp_write_mode   = ICTP_MODE_RAW;

        switch (minor) {
/*
  this allows interrupts on the push button
```

```
*/
        case ICTP_READ_IRQ7_COUNT:
          ret_code = request_irq(irq7,(void
*)ictp_irq7_interrupt,

SA_INTERRUPT,"ictp",NULL);
          if (ret_code) {
              printk(KERN_WARNING "ictp:  unable to use
interupt 7\n");
            return ret_code;
          }
          else {
#ifdef ICTP_DEBUG
            printk(KERN_DEBUG "ictp:  irq7
registered\n");
#endif
            command = ICTP_MODE_SELECT | ICTP_A_MODE_1 |
ICTP_B_MODE_0
                                        | ICTP_INPUT_B;
            outb(command,ICTP_S);           /* strobed
output */
/*
  kill the buzzer
  first setup port C to bit set (bit set/reset mode with
set bit on!  )
```

```
 */
            outb(ICTP_SILENCE,ICTP_S);
           }
           break;

         case ICTP_READ_IRQ5_COUNT:

           ret_code = request_irq(irq5,(void
 *)ictp_irq5_interrupt,

 SA_INTERRUPT,"ictp",NULL);
           if (ret_code) {
               printk(KERN_WARNING "ictp:  unable to use
 interupt 5\n");
               return ret_code;
           }
           else {
           command = ICTP_MODE_SELECT | ICTP_A_MODE_0 |
 ICTP_B_MODE_1
                                      |
 ICTP_INPUT_B;
          outb(command,ICTP_S);           /* strobed
 input */
 /*
   kill the buzzer
   first setup port C to bit set (bit set/reset mode with
 set bit on!  )
 */
            outb(ICTP_SILENCE,ICTP_S);
 #ifdef ICTP_DEBUG
            printk(KERN_DEBUG "ictp:  interrupt 5
 registered\n");
 #endif
           }
          break;

        case ICTP_READ_SWITCHES:
          command = ICTP_MODE_SELECT | ICTP_A_MODE_0 |
 ICTP_B_MODE_0
                                    | ICTP_INPUT_B;
          outb(command,ICTP_S);           /* setup to
 non interrupt */
 /*
   kill the buzzer
   first setup port C to bit set (bit set/reset mode with
 set bit on!  )
```

```
*/
          outb(ICTP_SILENCE,ICTP_S);
#ifdef ICTP_DEBUG
          printk(KERN_DEBUG "ictp:  opened for switch
reading\n");
#endif
          break;

        default:
          return -EINVAL;
        }
        return 0;
}

static void
ictp_release(struct inode * inode, struct file * file)
{
        unsigned int minor = MINOR(inode->i_rdev);

/*
  free the interrupt
*/
        switch (minor) {
        case ICTP_READ_IRQ7_COUNT:
          free_irq(irq7,NULL);
#ifdef ICTP_DEBUG
          printk(KERN_DEBUG "ictp:  interrupt 7
free'd\n");
#endif
          break;
        case ICTP_READ_IRQ5_COUNT:
          free_irq(irq5,NULL);
#ifdef ICTP_DEBUG
          printk(KERN_DEBUG "ictp:  interrupt 5
free'd\n");
#endif
          break;
        default:  ;
        }
        ictp_busy  = ICTP_FREE;
}
```

```
 static struct file_operations ictp_fops = {
     NULL,                /* seek       */
     ictp_read,
     ictp_write,
     NULL,                /* readdir */
     NULL,                /* select  */
     ictp_ioctl,
     NULL,                /* mmap */
     ictp_open,
     ictp_release
 };

 /*
  * And now the modules code and kernel interface.
  */

 int
 init_module( void) {

         unsigned char testvalue = 0;

 #ifdef ICTP_DEBUG
         printk(KERN_DEBUG "ictp:   init_module
 called\n");
 #endif

 /*
    initialize the chip
 */
         ictp_reset();

         testvalue = inb(ICTP_B);
 /*
    set bit 5 of port C and read back.  This bit is
 unused
 */

         outb(ICTP_SET_TSTBIT,ICTP_S);
         testvalue = inb(ICTP_C);
 #ifdef ICTP_DEBUG
         printk(KERN_DEBUG "ictp:  port C after set bit 5
 %x\n",testvalue);
```

```
#endif
        if ((testvalue & ICTP_TSTBIT) == 0) {
          printk(KERN_ERR "ictp:  board not found!\n");
          return -ENODEV;
        }
        outb(ICTP_RESET_TSTBIT,ICTP_S);
        testvalue = inb(ICTP_C);
#ifdef ICTP_DEBUG
        printk(KERN_DEBUG "ictp:  port C after reset bit
5 %x\n",testvalue);
#endif
        if ((testvalue & ICTP_TSTBIT) != 0) {
          printk(KERN_ERR "ictp:  board not found!\n");
          return -ENODEV;
        }

/*
  register the device driver with the system
*/
        if (register_chrdev(HW_MAJOR, "ictp",
&ictp_fops)) {
          printk(KERN_ERR "register_chrdev failed:
goodbye world :-(\n");
          return -EIO;
        }
#ifdef ICTP_DEBUG
        else
          printk(KERN_DEBUG "ictp:  driver
registered!\n");
#endif
        return 0;
}

void
cleanup_module( void) {

#ifdef ICTP_DEBUG
  printk(KERN_DEBUG "ictp:   cleanup_module called\n");
#endif
  if (ictp_busy)
    printk(KERN_WARNING "ictp:  device busy, remove
delayed\n");

  if (unregister_chrdev(HW_MAJOR, "ictp") != 0) {
    printk("cleanup_module failed\n");
  }
```

```
#ifdef ICTP_DEBUG
  else
    printk(KERN_DEBUG "cleanup_module succeeded\n");
#endif
}
```