The Abdus Salam
International Centre for Theoretical Physics

United Nations
Educational, Scientific
and Cultural Organization

International Atomic
Energy Agency

310/1779-9

**Fourth Workshop on Distributed Laboratory Instrumentation Systems
(30 October - 24 November 2006)**

Legacy Systems

Anthony J. WETHERILT
UNIDO-ICHET
Sabri Ulker sok, 38/4
Cevizilbag, Zeytinburnu
34015 Istanbul
TURKEY

Legacy Systems

A.J. Wetherilt[†]*

[†] *UNIDO-ICHET*
Sabri Ulker Sok, 38/4,
Cevizlibag, Zeytinburnu
34015 Istanbul
Turkey

Lecture given at the:
The Fourth Workshop on
Distributed Laboratory Instrumentation Systems
Trieste, 30 October – 24 November 2006

LNS

*jwetherilt@unido-ichet.org

Abstract

Various items of hardware developed for previous Micro–processor workshops are presented as legacy devices for network communications. These devices are a small MC6811 based micro–controller and a MC6809 system running RInOS, a small real–time operating system. Interfacing to input/output devices, in the form of two boards based around LCD and LED displays is also presented.

Keywords: Linux Embedded systems, Real-Time kernels, 6811, 6809

Contents

1 Introduction

Modern micro-controller technology makes much use of Ethernet based technology to provide such features as ftp and web page servers. Often, these devices need to be interfaced to older equipment without such functionality. These *legacy devices*, as they are often named, are generally relatively older instruments that come equipped with serial or other ports which can be connected to another device equipped with Ethernet capability through one or other of the ports, thus providing an inexpensive method of combining current data distribion methods with older equipment. Such techniques are of practical use in almost any laboratory as few institutions can afford to throw away perfectly functioning equipment on the grounds that it does not communicate with the flavour–of–the–day in networking protocols.

Since the techniques are fairly general, in this Workshop, we illustrate how to interface legacy systems by using two micro–controller/processor boards developed for use in previous Colleges and Workshops. These systems are both equipped with serial and parallel ports and can be interfaced to LCD and LED based input/output modules. These notes present a review of these items of equipment in preparation for the various exercises in which they will be used.

2 Input/Output Hardware

2.1 The Colombo board

The Colombo board is actually a complete system with provisions made for a 6809 microprocessor, a 6821 programmable interface adapter, and RAM and ROM situated on one half of the board. The remainder of the board comprises a 4 digit, 8 segment LED display together with various switches and devices that can be interfaced via a 26 pin connector (J2) to either the on-board microprocessor or an external host machine (see Figure 1). It is this latter feature that has been used predominantly during the various colleges.

The 26 pin connector definitions are shown in Figure 2 and are to be considered the standard connections for College instrumentation. Two sets of data lines can be seen which reflect the characteristics of the 6821 PIA around which the board was designed. The set of A lines (PA0 - PA7) are connected to the latch/drivers of the 4 LED displays and data latched in the following manner (see Figure 1: 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 2): 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

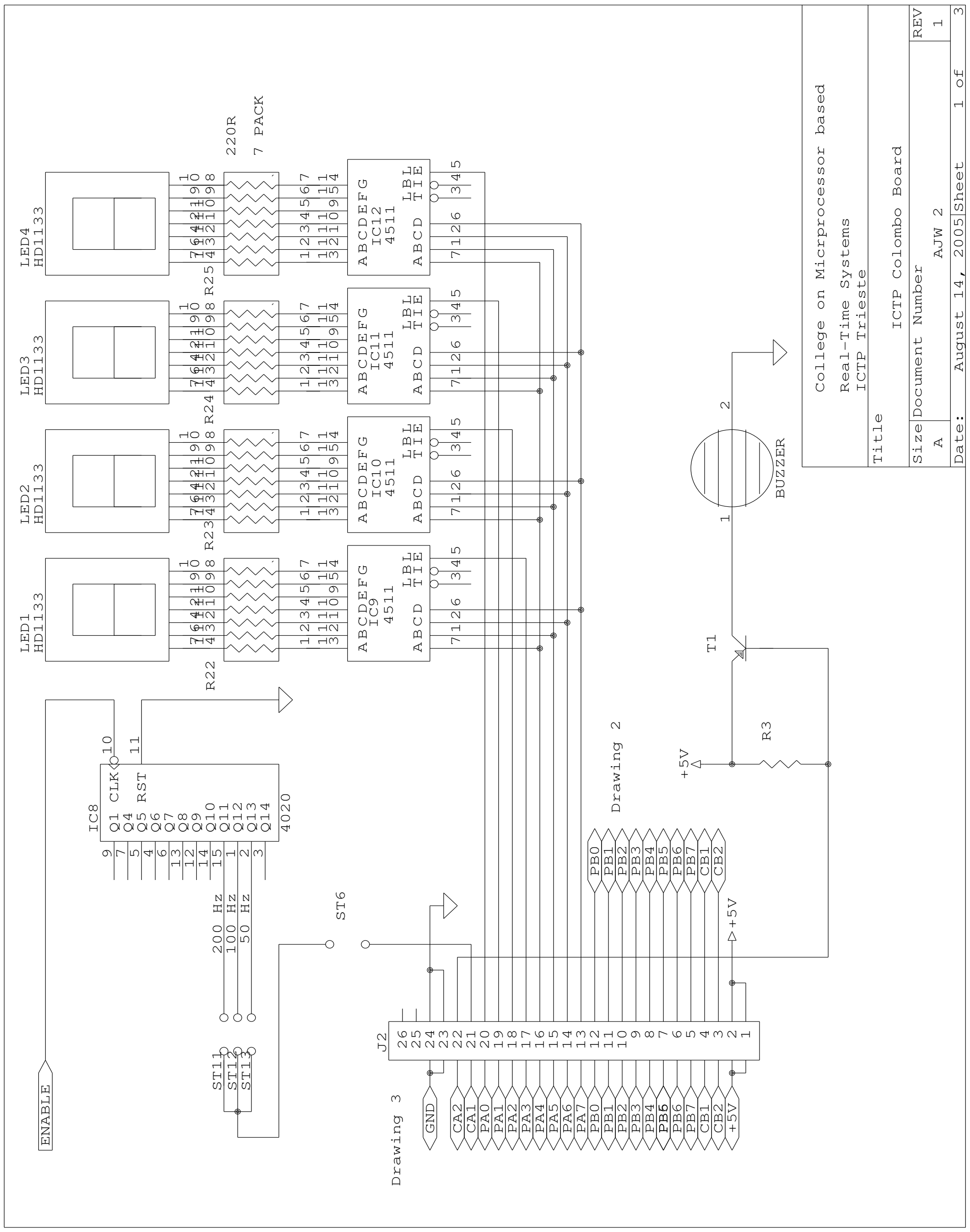

Figure 1: Schematic Drawing of the Colombo Board

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.

2.2 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 3). 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.

The LCD display is an Agena AA16102 module capable of displaying up to 16, 5x7 pixel alphanumeric characters. With CA2 held high, data are placed on lines PB0-PB7. The line PA1 is set low for a write operation and PA2 is set according to whether the operation is a write data (high) or a write instruction (low). The line PA0 is then strobed from low to high to low for the data to be latched. For full details of all possible operations, please refer to the manufacturer's data sheet.

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.

2.3 The M68HC11 board

This board was also designed by C.S. Ang and provides a complete but minimal system with but three IC components (Figure 4. It includes a serial port for communications with another system, both or programme development and downloading and subsequent networking. Header strip connectors are provided for intefacing to other equipment, specifically the LCD or Colombo boards.

When deveoping a programme to run on the HC11 board, the following steps should be followed:

1. Write the code to run on the target board on a host machine (typically a PC). The code can either be written in assembler or a high level language. A version of of the GCC, the GNU C compiler, is available for the 6811 and allows complete software development in C. Libraries are also available to take care of the nuts and bolts issues of using the various items of hardware. The output code is in a format known as *S19* code which is suitable for downloading to the 6811 target.

2. Select programming mode on the 6811 target by setting the mode selection switch to *BOOT* and download the PRGHC811 bootloader programme to the board. Once

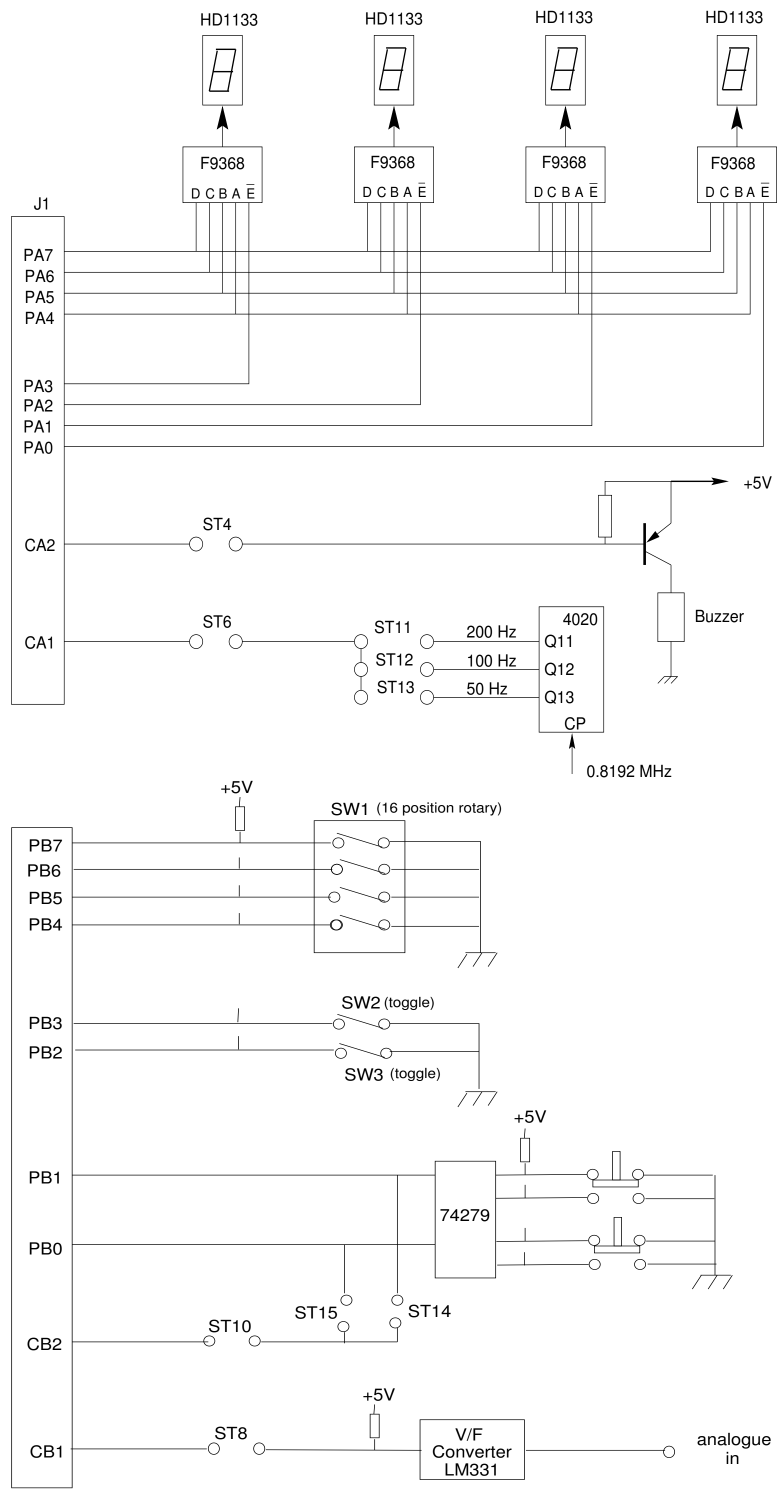

Figure 2: Simplified details of the Colombo Board showing A and B sides

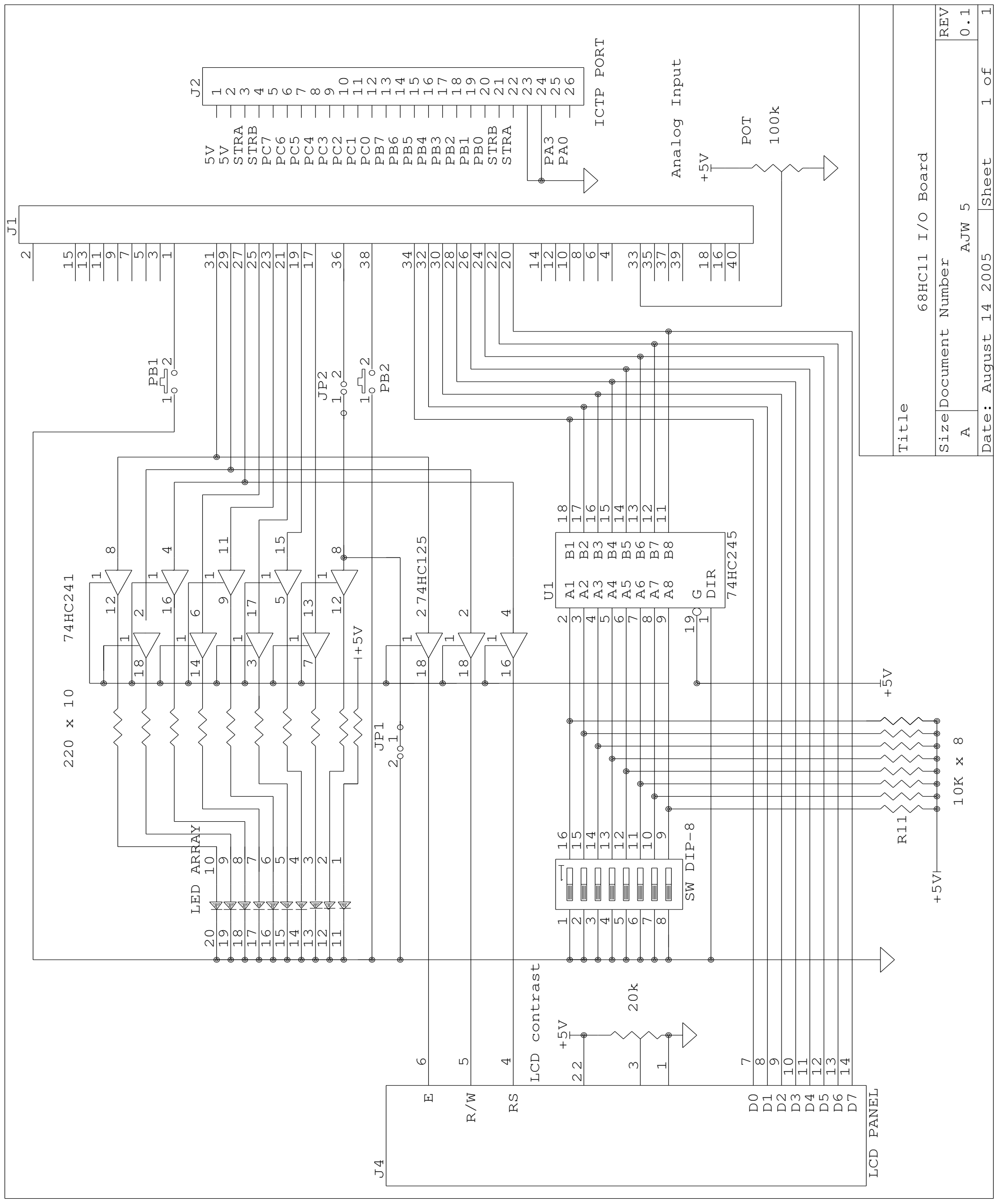

Figure 3: Schematic Drawing of the LCD Display Board

complete, the boot image will run and will wait for a file in .S19 format to be sent over the serial line.

3. Download the .S19 file to the target. This involves running a terminal emulator such

as *seyon* in download mode. When the 6811 target receives the file it decodes the S19 format and writes the included binary code into the on–board EEPROM of the 6811.

4. When the downloading has finished, set the switch to *NORMAL* and press the reset button. The downloaded code should now run on the target board.

Of course the above sequence of events is somewhat idealised. Under normal circumstances, almost all code needs to be debugged before it will run! This usually involves the trial and error approach of compiling testing, recompiling and retesting and so on until the programme behaves as intended. With the limited tools available with the HC11 target, this might involve a fairly long and drawn out process. In the absence of an in-circuit emulator, a better approach is to use a software simulator/debugger such as the db11. Using such a simulator, breakpoints can be set, registers examined and memory values changed, and the debugging process becomes relatively straightforward, with the real hardware coming into play only when the obvious errors have been rectified and there is areasonable chance of the code running correctly.

2.4 The ICTP09 board

2.4.1 Introduction

The ICTP09 board provides coverage of embedded systems programming. In contrast to the 6811 board, the ICTP09 board implements a large number of functions at the price of complexity: Some 24 integrated circuits are used in its construction (Figure 5, Figure 6, Figure 7, Figure 8, Figure 9). 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.

The memory map of the system is shown in Figure 11.

2.4.2 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 was always laid on those areas that would allow the various aspects of micro–processor teaching to be emphasised. For this reason two identical serial communications ports were provided. These allow communications drivers to

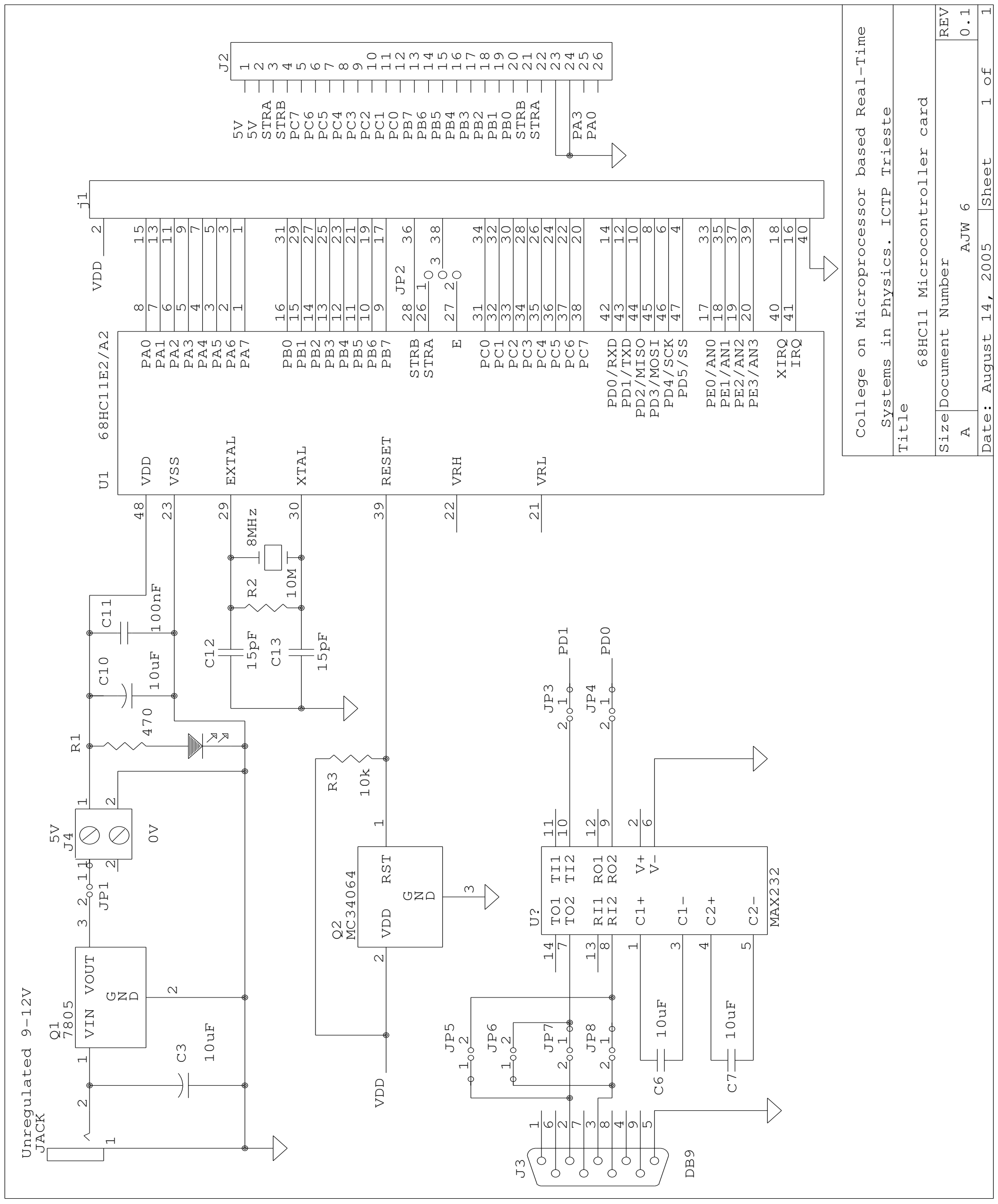

Figure 4: Schematic Drawing of the M6811 board

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 8) must

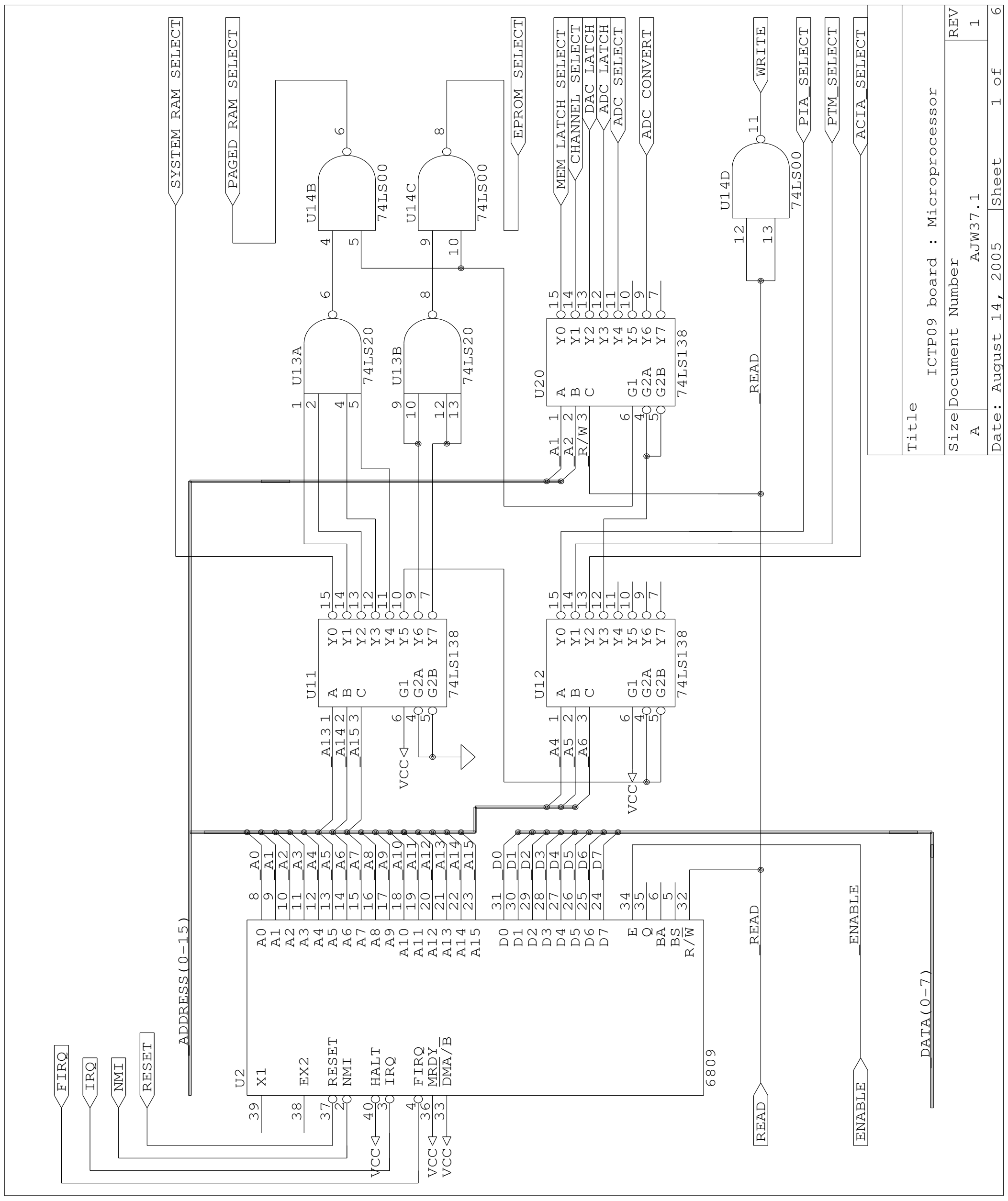

Figure 5: Schematic Drawing of the ICTP09 board:Microprocessor subsection

be configured so that the clock is divided by 1 rather than 16 and the jumpers adjusted accordingly. Communication uses only the TxD, RxD and ground return lines of the 9 pins of the RS 232 ports. For interconnection between the board and a host PC, null modem cables must be used.

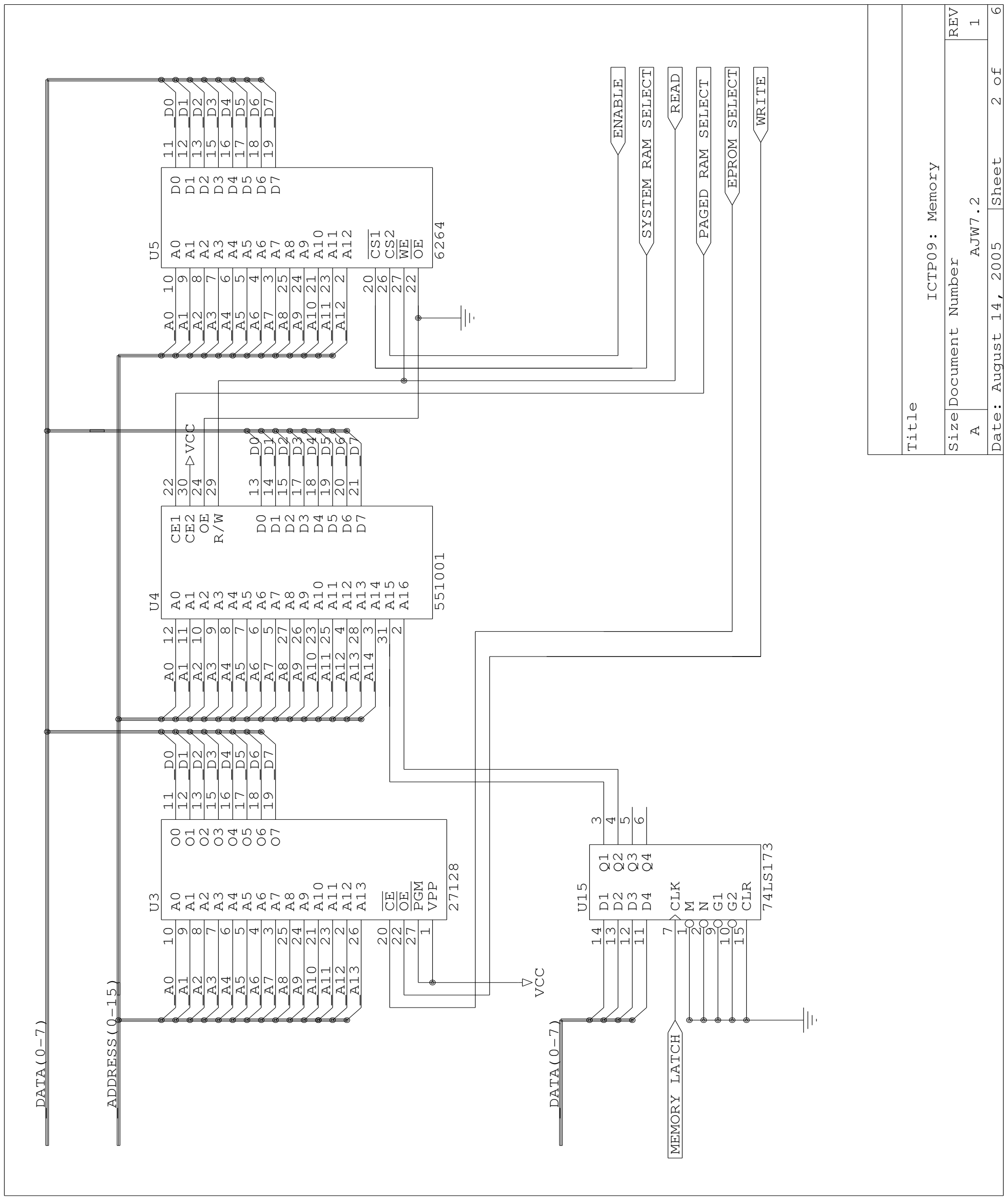

Figure 6: Schematic Drawing of the ICTP09 board:Memory subsection

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

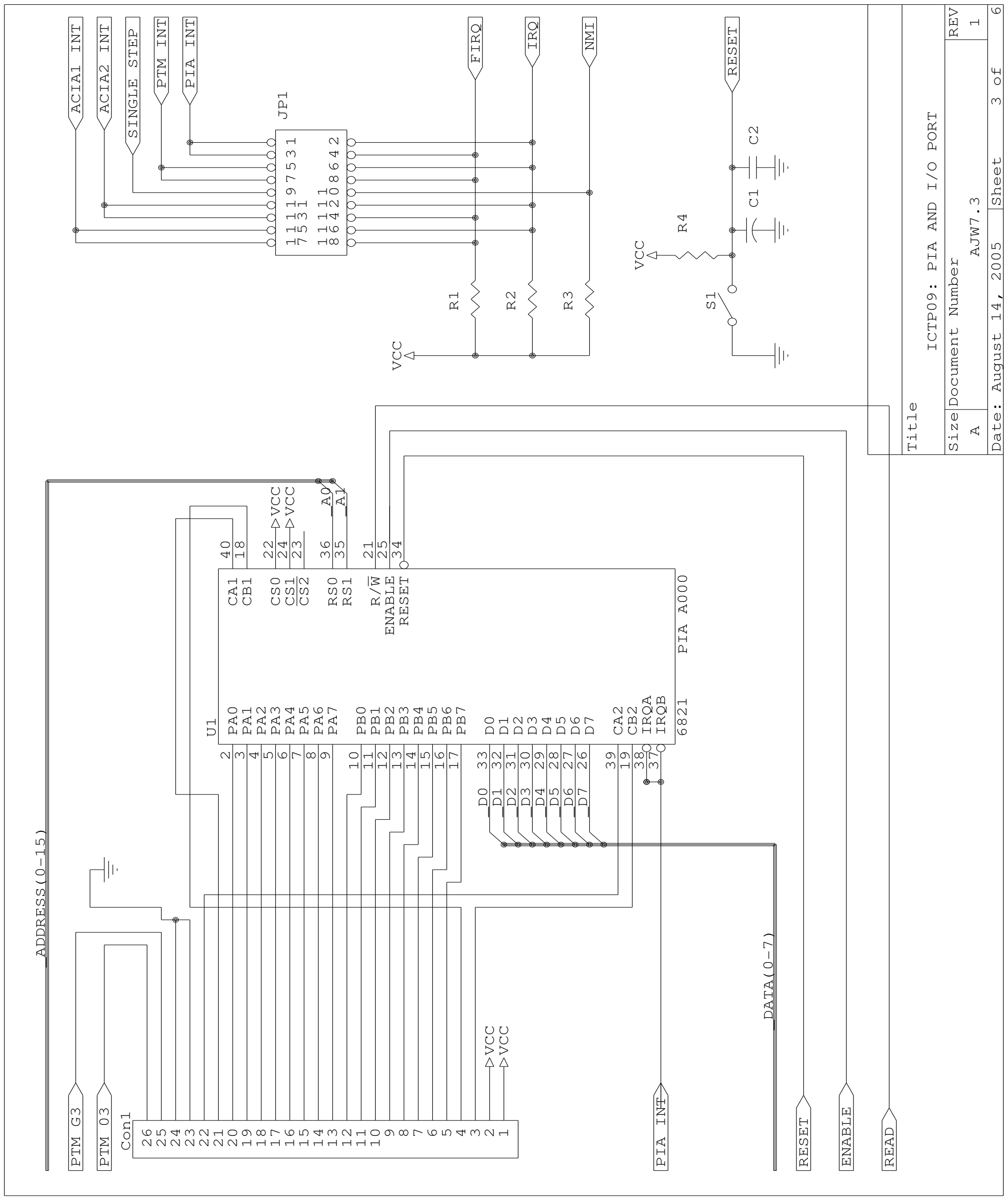

Figure 7: Schematic Drawing of the ICTP09 board:I/O ports subsection

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

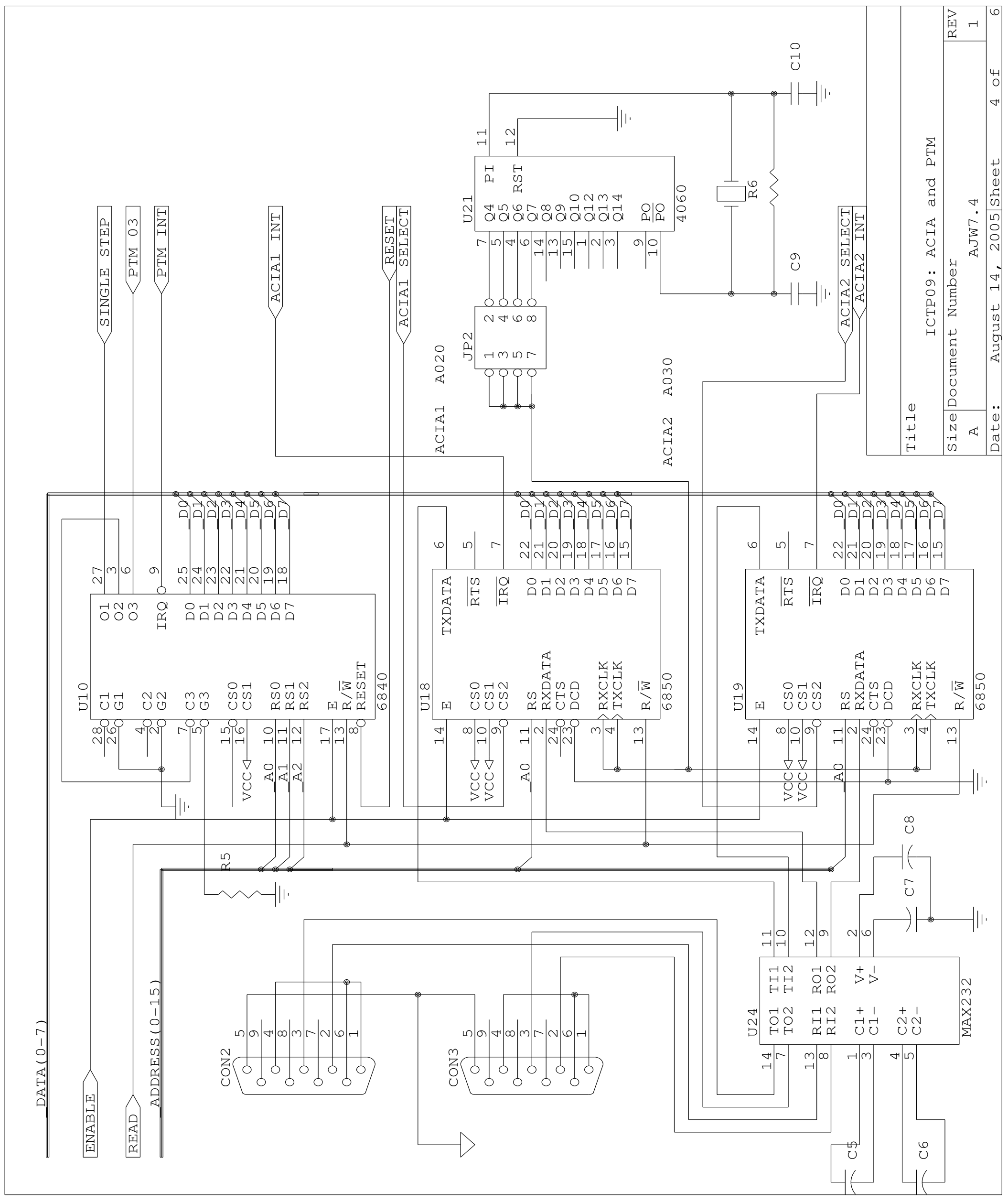

Figure 8: Schematic Drawing of the ICTP09 board:ACIA and Timer subsection

hang the system. Refer to Figure 2.4.2 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

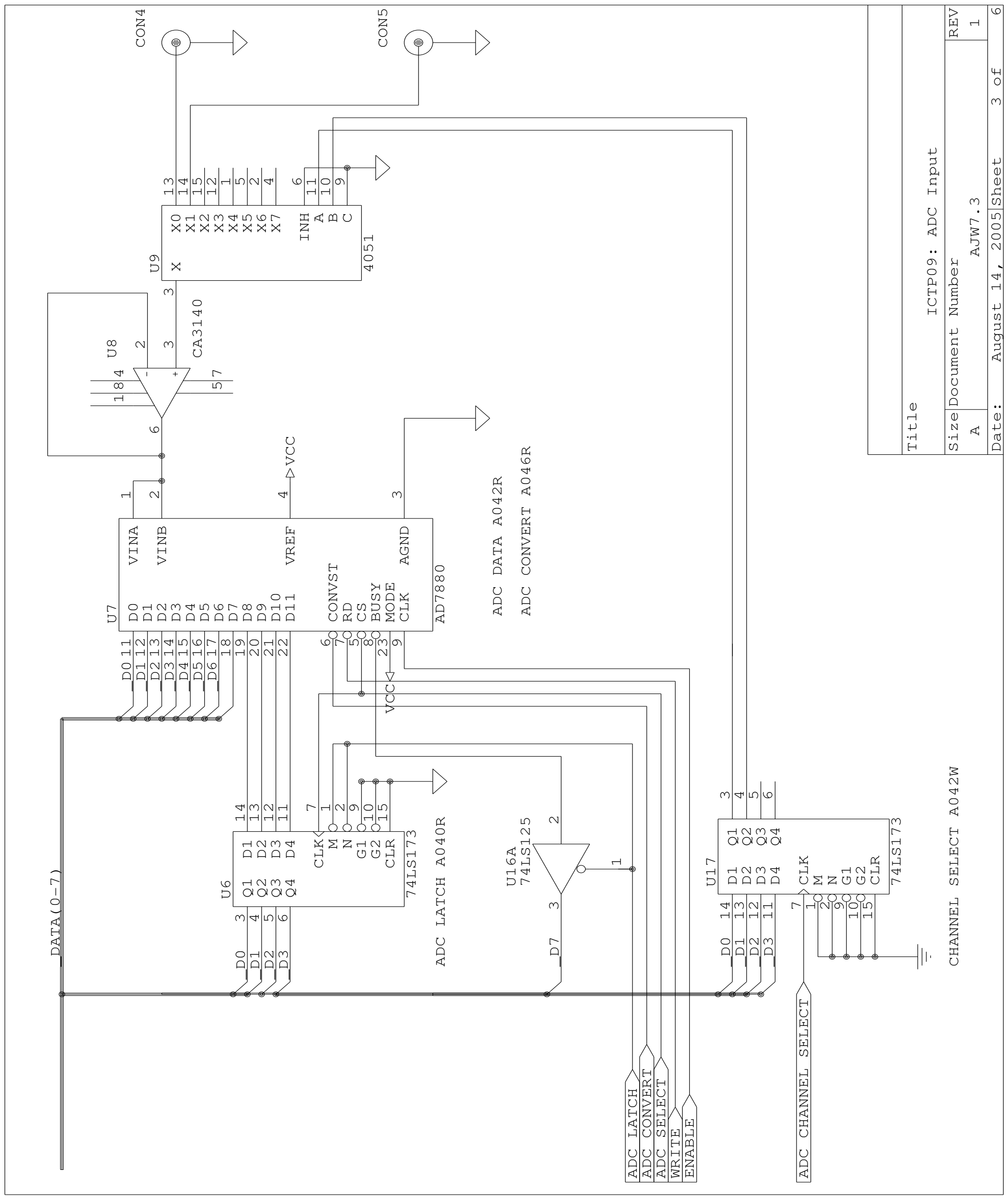

Figure 9: Schematic Drawing of the ICTP09 board:ADC input subsection

(see Figure 6). 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

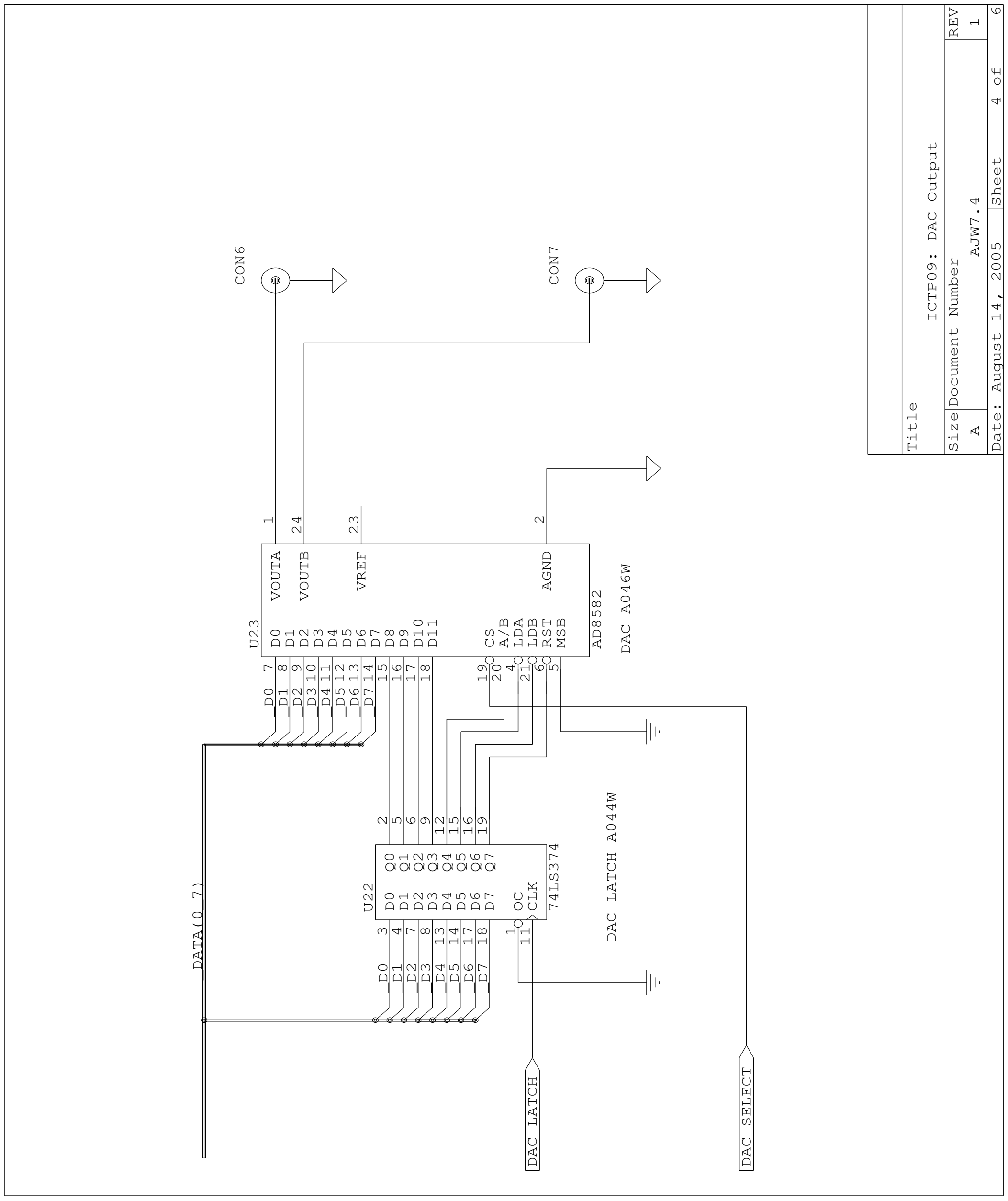

Figure 10: Schematic Drawing of the ICTP09 board:DAC outputsubsection

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

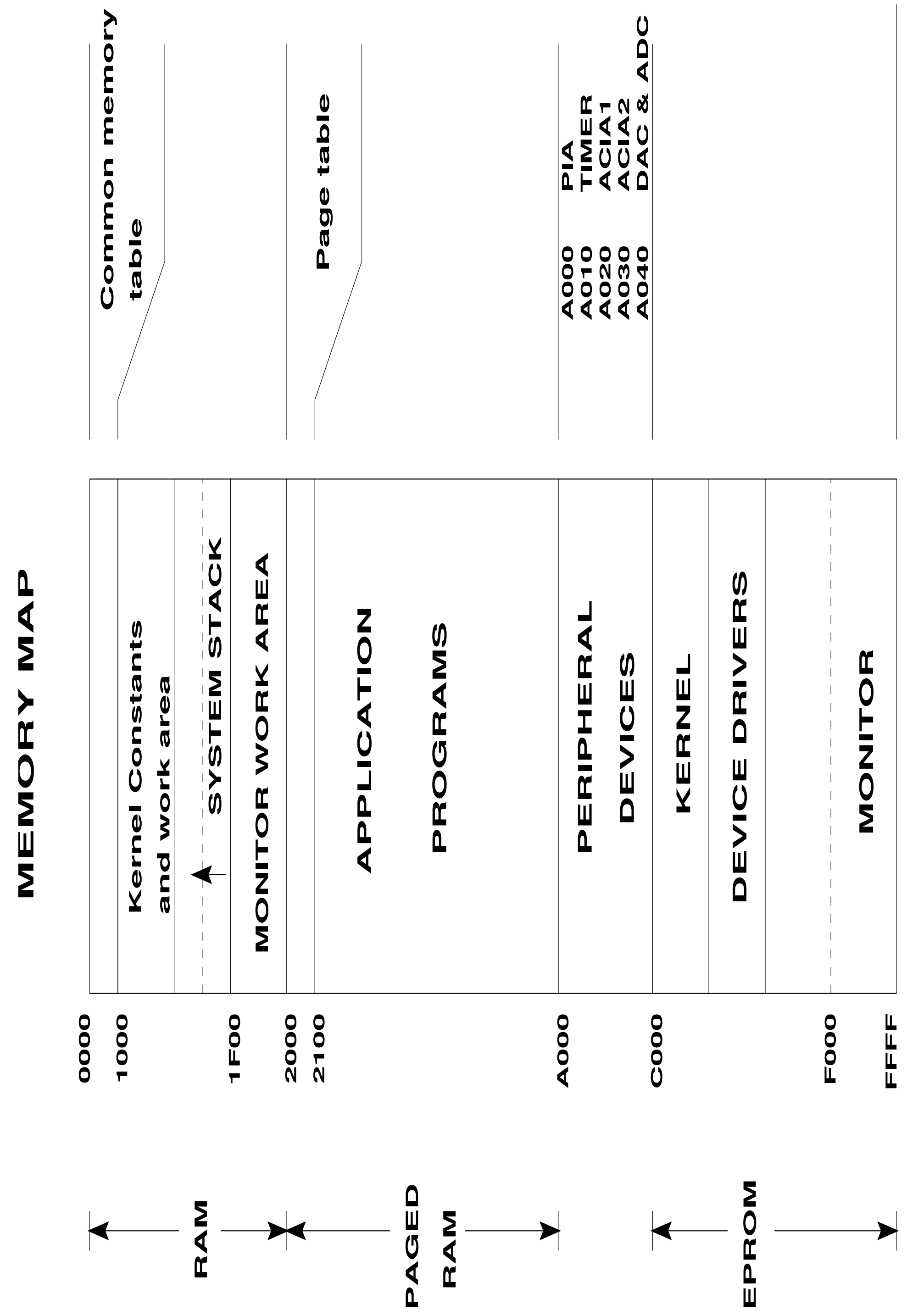

Figure 11: Memory Map of the M6809 under RInOS

request. For times longer than this, timer channel 3 can be used.

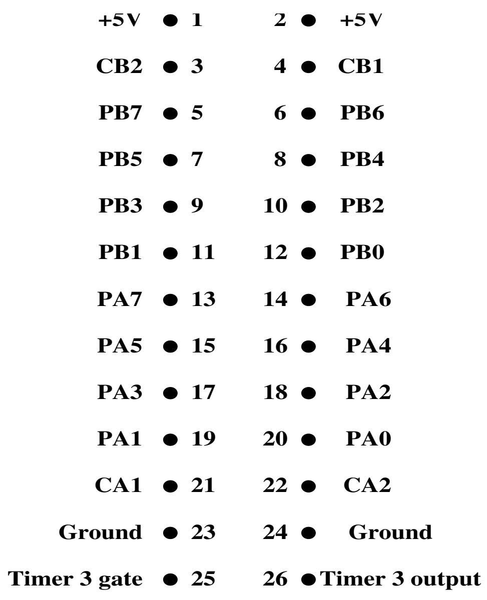

Figure 12: [Pin Definitions of the Standard ICTP Connector

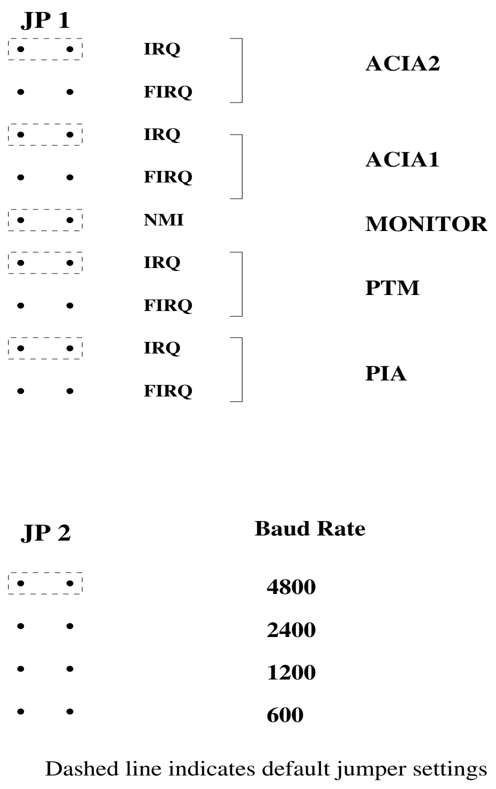

Figure 13: Jumpers Setting for the ICTOP09 Board

3 The RInOS kernel

RInOS (for **R**eal-Time **In**tegrated **O**perating **S**ystem) is a full-fledged Real-Time, multitasking kernel for embedded processors. Originally targeted at the **MC6809** and written in the assembly language of that processor, it is now being rewritten virtually entirely in C with the aim of porting it to other small embedded systems. The original design criteria were:

(i) Use software-interrupt system calls to an EPROM based kernel for interfacing to client programmes rather than linking in with a library based version at compile time.

(ii) Allow a variable number of client applications to run concurrently in RAM when the kernel is 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 can be changed after the start of the kernel and without having to re-assemble the system code.

(v) Provide a means of downloading code to a target board and running the code under the control of an external debugger.

(vi) Provide external libraries for wrapping the operating system calls and increasing functionality. These libraries are to provide compatibility with standard implementations on other systems and allow a measure of cross platform development.

The first four items in this list of design criteria belong completely to the domain of the kernel itself, whereas the remaining items are descriptions of external requirements and impose limits on the scope of the kernel design. Only those items that are essential to the operation of the kernel should be placed in it and anything that can be handled, as a general, external library function should be left out. Thus the kernel should contain a complete set of functions needed for the concurrent operation of real-time, multitasking processes, with everything else being left for external libraries.

Since C code is generally much easier to understand and considerably more concise than assembler, where possible, C has been used to illustrate the principles outlined, although the currently released versions of the kernel are written entirely in assembler and differ in detail from the some of the descriptions presented here.

4 Components of a multi-tasking kernel

In any operating system, certain requirements must be fulfilled in order to achieve the desired performance. Multi-tasking and real-time requirements each place specific demands and constraints on the system that the designer must comply with. Another important issue concerns the manner in which the target hardware interacts with the kernel and how this interaction can be generalized to allow better porting to other hardware platforms.

An operating system is basically a set of functions or system calls that provide a framework for a unit of code to execute and interact with other such units. Each process or task runs in a manner that is governed by the operating system and makes use of the system calls provided by the operating system. Thus when coding a process, the programmer can employ existing and tested functions with ease and does not usually need to be concerned with the underlying details of the platform. A basic set of functions would start with items for the creation and subsequent destruction of processes.

A process that has no interaction with its hardware platform would be of neither much interest nor use, so provision should be made for processes to use the hardware platform through special system function calls or device drivers, that allow standardized access to, for example, the input and output channels of a serial communications channel. When one considers the usage of hardware it soon becomes apparent that a process often has to wait for some event on a particular piece of hardware for a response with the desired information. For example, the serial communications port inputs a byte roughly once every 2080 μs (at a Baud rate of 4800). This means that on the scale of a microprocessor, there is a very long wait between input bytes, a time that could possibly be put to better use by processing something else. An efficient manner of achieving this is to arrange for multiple processes to divide up the work between them so that, for instance, one process could be monitoring input from the serial port and a second could be doing calculations or something similar. Then the operating system must provide mechanisms for deciding which process will be running at any one time (scheduling) and how to switch between processes (context switching). In real-time systems, a related problem is how rapidly a process can be switched when an interrupt arrives so that the time used in servicing the interrupt can be minimized. The concept of a **thread** (sometimes called a **lightweight process**) is often used. On some systems context switching involves considerable work and can take many processor cycles. Threads are part of a process and a single process can consist of several threads. If context switching is allowed between the threads, this can be a much simpler affair than a full context switch between separate processes. Thus threads are often used to improve response times. Under RInOS, threads and processes are identical.

Once multiple processes are considered, a whole host of secondary issues must be addressed. Firstly, the idea of synchronisation of resources (be they variables, sections of code, hardware etc) must be considered. A classic example of the need for synchronization is as follows: Two processes A and B attempt to write to a serial, byte-oriented printer. A tries to write the word 'Hello', and B 'Goodbye'. Let us assume that they write slowly so that the operating system performs several context switches during the time they both write to the printer. What will be the output? Assuming A starts first it will send the 'H' character maybe followed by an'e'. If at this stage we get a context switch B will then send a 'G'. It is not difficult to imagine the effects of several more context switches producing output something like "HeGololdboye" which presumably is the intention of neither A nor B. What is needed here is a lock that the first process can claim and once claimed will prevent access to other processes. When the process finishes its business it releases the lock, which then becomes available for another process to claim. In this way, access to resources can be synchronised between processes. A generalization of this principle is the semaphore with the associated operations Down and Up corresponding to lock and unlock. A second need when using multiple processes is the ability to communicate information between two or more processes in a synchronised manner. Several mechanisms for inter process communication (IPC) are often implemented in an operating system such as shared memory, pipes, messages and signals with each mechanism having its own advantages and disadvantages. Thirdly, some form of control over the running of processes is also needed. For example, each process should have a separate priority, which the scheduler uses to determine when the process should run, and it must be possible to change this priority. It must also be possible

Table 1: Basic function calls of a multi-tasking operating kernel

Process functions	1	Create a process
	2	Kill a process
	3	Set process priority
	4	Put a process to sleep
	5	Wake a sleeping process
Semaphore operations	6	Up semaphore
	7	Down semaphore
IPC functions	8	Send message
	9	Receive message
	10	Signal a process
	11	Open a pipe
	12	Close a pipe
	13	Write to a pipe
	14	Read from a pipe
Memory management functions	15	Allocate memory block
	16	Deallocate memory block

to prevent a process from running or put it to sleep and subsequently wake it up again.

Finally, an operating system generally exerts control over the allocation of memory resources to processes. Whilst this is not an essential item of an operating system it is a common feature especially in larger systems.

These then are a minimal set of function calls that most multi-tasking operating systems implement in one way or another. Most large systems implement considerably more than this number. RInos actually implements, 43 calls. Please refer to reference [1] for a complete listing.

5 Software tools and libraries

In order to use the ICTP09 board and RInOS kernel effectively, a number of other tools have had to be developed. This section presents an overview of these tools and the reader is referred to the 6809 Manual [1] for further details.

5.1 The command interpreter shell

In order to interact with the kernel, code must first be placed in the RAM of the target board. A small and simple command interpreter shell has been written that allows commands to be input over a serial line from a PC terminal. The shell is loaded following initialistion of the kernel as a child process and intercepts the input stream to the serial driver. If the ESC (0x1b) character is detected in the stream, the shell is invoked and the user can enter an interactive mode in which code can be downloaded to the board, run and debugged using a set of basic commands. Remote debugging is possible using these commands.

5.2 The gcc compiler

A complete set of compilation and dubugging tools have been provided mainly through the efforts of Rinus Verkerk. The GNU C compiler has been adapted to produce position independent M6809 assembler code, which together with a suitable assembler and linker gives relocatable code in ELF format. The standard C libraries, and a startup module crt0, that performs initialisation and setting up command line arguments are also linked into the ELF output.Embedded within the ELF output file is sufficient information for high level debugging of the code. The gcc compiler can be invoked using the following command:

```
cc09 [-Wall] [-v] [-o output] prog.c
```

Here, `prog.c` is the input file(s) and the first two options inform the compiler that you want to view any warnings generated and to see all steps involved. The third option allows the name of the output file to be changed (in this case to output).

5.3 The Java translator

Carlos Kavka has put together a system for running java programmes on the ICTP09 board. The **j09** script first calls the standard Java compiler provide with the JDK from Sun Microsystems and then translates the resulting Java byte code into 6809 assembler code. The assembler code is passed through the same assembler programme used by gcc and finally linked together with the startup module crt0 to produce an ELF executable file. The command:

```
j09 Test.java
```

will cause the java file Test.java to be compiled, assembled and linked to produce the executable file, Test, which can be run on the ICTP09 board.

5.4 The XGUM symbolic debugger

A symbolic debugger running under XWindows has been developed to ease the process of debugging the ICTP09 board. It is invoked by:

```
xgum
```

XGUM consists of a client front end that connects via a pipe to a server specific to the requirements of the debugging session. Presently two servers are available for debugging the ICTP09 board directly and an ICTP09 simulator that allows debugging on the pc itself. Both servers load the RInOS kernel. XGUM is independent of the source code language and can handle both C and Java.

6 Libraries available under RInOS

Libraries are available for provision of a number of functions under RInOS. First, support for the C language is made available via the **libc.a** library. This library contains the standard functions such as printf, putchar, malloc etc demanded by the ANSI C standard. The library is automatically linked in as part of the compilation chain and its functions can be referenced by use of the standard headers

```
<stdio.h>,
<stdlib.h>,
<string.h> etc.
```

The second group of library routines generally consists of wrapper functions for RInOS system calls and allow the user basic level access to the kernel and device driver services. These routines are found in **libcreal.a** and are automatically linked in as required. The routines are accessed by inclusion of the <¡syscalls.h> header file which also includes all basic information on structures and types defined and used by functions making system calls. This header is also automatically called by the standard header files (stdio.h) etc and only if none of these headers are included is it necessary explicitly to declare <syscalls.h> in a file. A second library, **libIOreal.a** performs a similar function by bridging the low level I/O services of the kernel to C level functions. These functions are in turn used by the standard I/O functions of the C library. If low level output is required, the header file <ICTP_IO.h> should be included.

The two members of the third group of library functions, **libgcc.a** and **libmath09.a**, are used by the compiler during code generation. **libgcc.a** is used by the compiler to perform integer arithmetic operations and to convert between the various integer types. **libmath09** performs a similar function with floating point functions. Neither library should be called explicitly in a user-defined function.

Finally, an implementation of the POSIX 1003.1c (pthreads) library is made available to simplify the mechanics of preparing multi-threaded programmes [5]. The implementation is reasonably complete within the confines of the 8 bit microprocessor platform and its use is encouraged as the functions for thread creation and mutex usage in particular offer greatly simplified functionality over the native RInOS functions. As extensions to this library, the following functions allow simple manipulation of (counting) semaphores and events

```
int event_init(event_t *event, const eventattr_t *attr)
int event_destroy(event_t *event)
int event_signal(event_t *event
int event_wait(event_t *event)
int semaphore_down(semaphore_t *sema)
int semaphore_init(semaphore_t *sema, const semaphoreattr_t *attr)
int semaphore_up(semaphore_t *sema)
int semaphore_destroy(semaphore *sema)
int semaphore_init(semaphore *sema, const semaphoreattr_t *attr)
int semaphore_destroy(semaphore_t  *sema)
```

All the definitions and types used in these function definitions can be found in the header file <pthread.h> which should be included when any reference is made to any function member of the library.

7 Programming examples

7.1 Introduction

Although programming in C under RInOS is quite straightforward, there are several points that should be noted. Code is presented that illustrate some of these points and demonstrate the use of several of the available libraries.

7.2 Creating threads and mutexes

This example uses the pthreads library to create a child thread and a single mutex. First a thread attribute is created and the desired priority of 20 is set using a variable of type struct sched_param.

Before the child is created, a static mutex is claimed using the down_user_sem() function. This is actually a RInOS wrapper function rather than the pthreads equivalent showing that the two libraries can be mixed at will. The child thread is then created using the pthread_create() function. Finally the mutex is released and the parent exits, at which point the child gains the mutex and is able to run. Note the use of the static initialiser for the mutex. Static initialisation is a convenvient method for all types of semaphore creation. In this example the thread owning the mutex will experience a priority boost if its priority is lower than any thread waiting on the mutex. Please refer to the `sycalls.h` and `pthread.h` header files for further semaphore types.

```
#include <pthread.h>
// Child prototype
void* child_thread(void *arg);
// Static mutex construction
struct semaphore  mutex = {MUTEX | SMBOOST ,1, 0,0,0};

int  main()
{
  pthread_t      child ;
  pthread_attr_t attr;

  // Child will have high priority, default is 10
  struct sched_param priority = {20};
  pthread_attr_init(&attr);

  // Initialise thread attribute
  // Set priority of thread
  pthread_attr_setschedparam(&attr,&priority);
  // Get semaphore before anyone else can
  down_user_sem(&mutex);
  pthread_create(&child, &attr, child_thread, NULL);

  // Finally release mutex
  up_user_sem(&mutex);
  return NULL;
}

void* child_thread(void *arg)
{
  // Try to get mutex
  down_user_sem(&mutex);
  return NULL;
}
```

7.3 Mutex, semaphore and event handling

The previous example showed how a mutex can be created using a static initialiser. The base type for all three semaphore types is the **semaphore** structure defined in syscalls.h which is redefined in pthread.h as **pthread_mutex_t**, **semaphore_t** and **event_t** for mutexes, counting semaphores and events respectively. Under pthreads, a mutex would defined as:

```
  pthread_mutex_t mutex =  PTHREAD_MUTEX_INITIALIZER;
```

This would initialise the mutex to a value of 1, and allow priority boosting by default. The priority boosting uses the prority inheritance protocol and _POSIX_THREAD_PRIO_PROTECT is defined by the implementation. The functions

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

are defined as operations on mutexes under pthreads which perform the down and up operations respectively.

Events and semaphores do not form part of the standard pthreads implementation. Extension functions have been added to allow use of the objects in a manner consistent with mutexes. Static initialisation of all semaphore types is similar and follows the outline of the following fragment in which a semaphore and several types of event are defined:

```
#include <pthread.h>    // For function prototypes
#define INITIAL_VALUE 10
semaphore_t count = {COUNT,INITIAL_VALUE,0,0,0}
event_t pevent = {EVENT,0};  // Persistent event
event_t revent = {REVENT,0}; // Resetable event
event_t sevent = {SEVENT,0;  // Single event, freed
```

```
                             // after signal
main()
{
   // Create a child
   ....
   event_wait(&revent);  // Wait for an event to occur
}

void* child(void* artg)
/*
  Child thread created by main
*/
{
  ....
  // Signal any threads waiting for the event
  event_signal(&revent)
}
```

The same frament using basic RInOS functions would be:

```
#include <syscalls.h>   // For function prototypes
#define INITIAL_VALUE 10
semaphore count = {COUNT,INITIAL_VALUE,0,0,0}
semaphore pevent = {EVENT,0};  // Persistent event
semaphore revent = {REVENT,0}; // Resetable event
semaphore sevent = {SEVENT,0;  // Single event, freed
                               // after signal
main()
{
   // Create a child
   ...
   down_user_sem(&revent);     // Wait for an event
                               // to occur
}

void* child(void* artg)
/*
  Child thread created by main
*/
{
 ....
  // Signal any threads waiting for the event
  up_user_sem(&revent)
}
```

Note again the use of an initial value for the counting semaphore.

As an alternative to static initialisation, all semaphore types can be created dynamically and initialised separately. Under RInOS however, this is wasteful of memory and is not recomended.

7.4 Memory allocation under RInOS

As the following examples shows, memory allocation follows standard ANSI C practise using `malloc()` and `free()`. In this example 5 blocks of (paged) memory are allocated and then freed.

```
#include "stdio.h"
#define NBLOCKS 5
void main(void)
{
  void* memptr[NBLOCKS];
  int index;
  // Allocate blocks of memory
  for (index = 0; index < NBLOCKS; index++){
    memptr[index]  = malloc(256);
  }
  // Free them again
  for (index = 0; index < NBLOCKS; index++){
    free(memptr[index]);
  }
}
```

Global or shared memory can similarly be accessed using the pair of non-standard functions:

```
  void* globalalloc(int size)
  void  globalfree(void *p)
```

7.5 Accessing system variables

When downloading a programme to RInOS, an optional, run-time argument list can be specified. The startup module, crt0 makes this list available to the programme via the standard argument passing mechanism of argc and argv. As usual, argc is the number of arguments passed to the programme and argv is an array of strings containing the individual arguments. The startup file crt0 is also responsible for making available several other global resources.

In earlier versions of RInOS, input and output functions acted on file descriptors rather than file pointers. In the current release it is expected to use the file pointers in the following list:

File	Filename	Description	Type
stdin		Standard input	read only
stdout		Standard output	write only
stderr		Standard error	write only
fpia	“lcd”	LCD file	read/write
facia1	“com1”	ACIA1 file	read/write
facia2	“com2”	ACIA2 file	read/write
fadc	“adc”	ADC file	read only
fdac	“dac”	DAC file	write only

With the exception of stdin, stdout and stderr, all the preceding files should be opened prior to use and closed when finished in the standard manner. Thus the following code fragement will open and later close the the first serial port for writing:

```
  #include <stdio.h>
 FILE* f;       // File pointer definition
  ...
  if ((f = fopen(‘‘com1’’,’’w’’)) != NULL){
        fprintf(f,...);
  }
  ...
  ...
 fclose(f);
```

Finally, crt0 provides access to the pushbutton on the LCD board. During kernel initialsation, a resetable event semaphore is reserved and initialised. On each press of the pushbutton, any threads waiting for this event will be woken. The startup module stores a pointer to the event semaphore in the global variable **pshbttn** and all any thread wishing to wait on this event has to do, is to perform a down using one other of the two functions:

```
  down_user_sem(pshbttn);
  event_wait(pshbttn);
```

The second of these is illustrated in the following code fragment:

```
/*
  Push button test
*/

#include <pthread.h>
extern semaphore_t* pshbttn;    // pushbutton semaphore
                              // pointer
void main(void)
{
    event_wait(pshbttn);      // wait for pushbutton to
                              // be pressed
    ....                      // Got event, now do
                              // something ...
}
```

References

[1] C. Verkerk, A.J.Wetherilt, *Software for the 6809 Microprocessor board.* 4, 5

[2] Andrew Tannebaum (1987) *Modern Operating Systems*, Prentice–Hall International.

[3] (1983) *The ASSIST09 Monitor*, Motorola

[4] Jean J. Lebrosse,*The μ/COS The Real–Time Kernel*,R & D Publications.

[5] Bradford Nichols, Dick Buttlar, Jacqueline Proulx Farrell (1996), *PThreads Programming*, O'Reilly & Associates, Inc. 6