INTERNATIONAL ATOMIC ENERGY AGENCY
UNITED NATIONS EDUCATIONAL, SCIENTIFIC AND CULTURAL ORGANIZATION
INTERNATIONAL CENTRE FOR THEORETICAL PHYSICS
I.C.T.P., P.O. BOX 586, 34100 TRIESTE, ITALY, CABLE CENTRATOM TRIESTE

UNITED NATIONS INDUSTRIAL DEVELOPMENT ORGANIZATION
INTERNATIONAL CENTRE FOR SCIENCE AND HIGH TECHNOLOGY

SMR/643 - 19

SECOND COLLEGE ON
MICROPROCESSOR-BASED REAL-TIME CONTROL -
PRINCIPLES AND APPLICATIONS IN PHYSICS
5 - 30 October 1992

A GENERAL PURPOSE PC BASED DATA ACQUISITION SYSTEM

J. WETHERILT
Scientific and Technical Research Council of Turkey
Marmara Scientific and Industrial Research Centre
P.O. Box 21
41470 Gebze Kocaeli
Turkey

1. Introduction

(1) Hardware elements : IEEE 488 bus

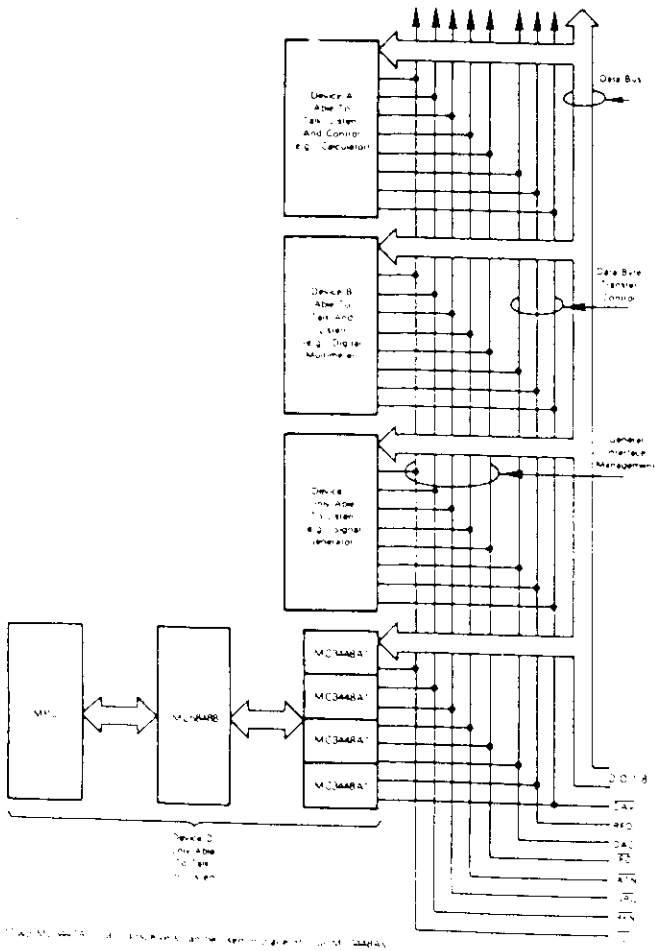
- Parallel, high speed (~ 1Mbits/sec max) data bus
- Three handshake lines DAV, NRFD, DAC
- Can broadcast or address instruments independently
- Maximum of 15 instruments can be attached to bus, each with a unique address
- Commands generally sent as ASCII strings ie "F0R0M3X"
- A status byte will be returned by a serial poll. Each bit is used to indicate the status of a different device dependent function except bit 6 which is reserved to indicate that the device has requested a service request (SRQ).

(2) Requirements of an aquisition system

- The system must be able to use an unspecified number of instruments up to a certain maximum
- It must be possible to designate experimental quantities to act as control parameters (voltage, current, temperature etc) that can be set to some initial value. When all the parameters are at their set points, a set of data will be acquired and some of the parameters will have their set points changed and the system will wait until all the parameters are again set. The process will continue until one or more parameters are outside preset limits when the system will stop. A maximum data aquisition rate of about 1 trigger/sec is expected.
- The system must be flexible enough to accomodate rapid changes in instrumentation or experimental method without the neccessity of recompilation

MC68488

FIGURE 5 — GPIB SYSTEM



2 Design of the acquisition system

- Modularity : For each instrument attached to the bus a memory resident module aims to model the intrument's behaviour. Only need to load sufficient code to drive the loaded instruments. Do not have to cater for situations that will not occur in the present experiment.

- Each module is to be configured at its time of loading with values that reflect the particular instrument and experimental conditions.

- A MASTER module is to coordinate all bus activity by communicating with each module, assessing its status and taking action accordingly.

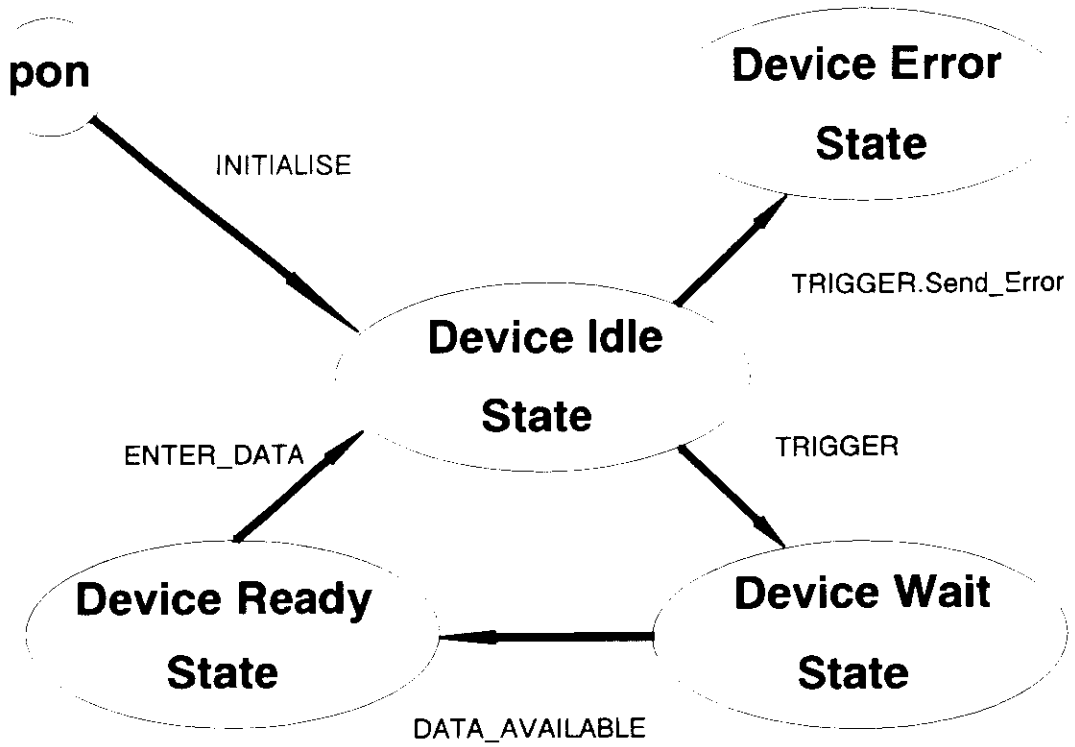
- A method by which modules can be loaded into and unloaded from memory is to be used.

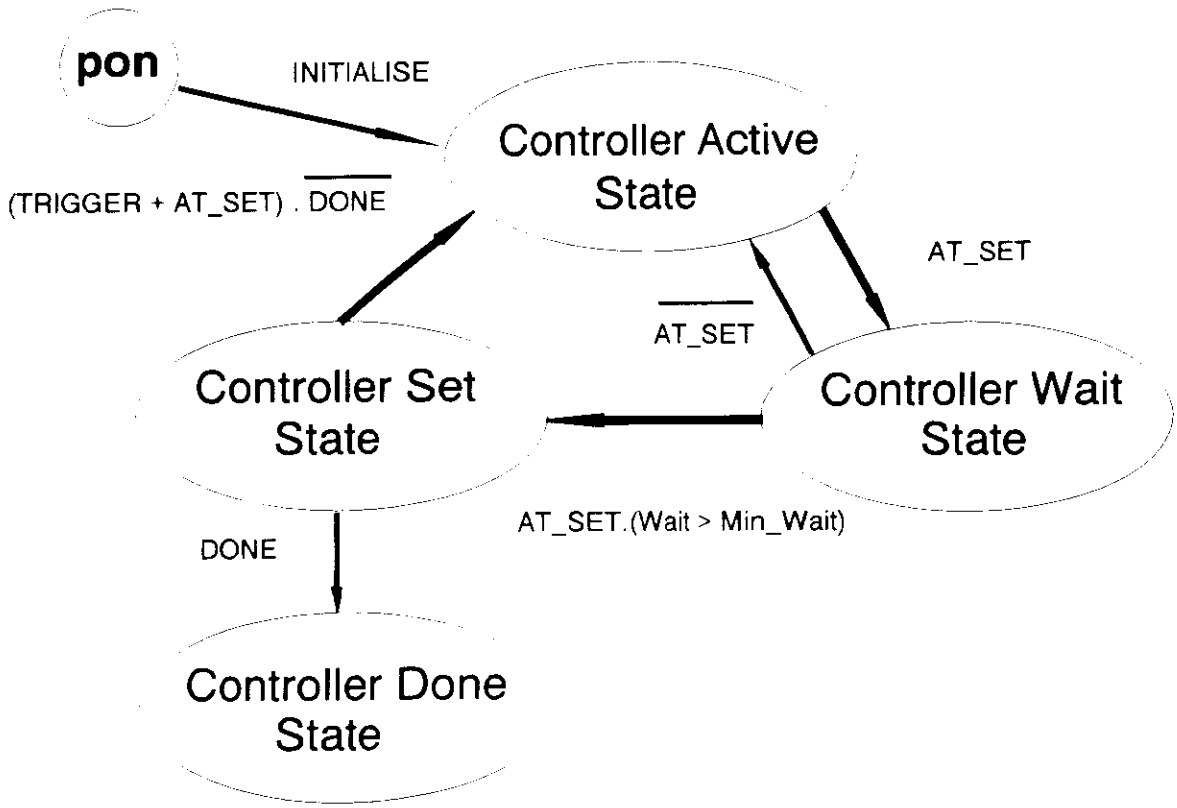
- DOS is to be used as the operating system because
  - (i) can run on small systems (no extended memory)
  - (ii) plenty of development software languages available
  - (iii) available on all PCs by default.

3 Module design

(1) Device functions :

- Initialisation : Set instrument to predefined state.
- Data Trigger : Instruct the instrument to start a measurement usually an ADC conversion)
- Status : The instrument is to return a status byte when requested that indicates (for example) whether or not a data reading is ready or the current error state
- Send Data : The instrument will send its data when so instructed
- Sensor output linearisation : Some sensors (ie thermocouples) are non linear and it is convenient to have their output linearised in some manner


pon
INITIALISE
Device Error State
TRIGGER.Send_Error
Device Idle State
ENTER_DATA
TRIGGER
Device Ready State
Device Wait State
DATA_AVAILABLE


pon
INITIALISE
Controller Active State
(TRIGGER + AT_SET) . DONE
AT_SET
AT_SET
Controller Set State
Controller Wait State
AT_SET.(Wait > Min_Wait)
DONE
Controller Done State

Controller functions :

- All device functions to be contained in the module but to be hidden from an external user. The device functions will be used by the controller to obtain data concerning its status.
- The controller will reset its set point value when so instructed by the Master and will indicate when preset limits have been exceeded
- The controller must indicate to the Master its current status and in particular when the contolled quantity is both within a given error window and stable.
- Some passive controllers need only receive an initialisation command after which they move to their set point and when stable do nothing further

Master Module functions :

- The Master module will act as a supervisor to each of the installed modules. It will periodically instruct the software associated with each instrument to perform and update of its current status. When all controllers are 'SET' a general TRIGGER will be sent to all active modules intstructing them to obtain their current values. Active controllers will reset their set points.
- An interface through which an external user can monitor and control the progress of an experiment should be implemented by the Master.
- The master module can implement many of the software functions required by the other modules. This removes the neccessity of duplicating these functions in each module and can result in considerable savings in code. Examples of such functions are : bus communication; file handling; floating point handling etc.

Timer Module

- In some cases it is neccessary to log data as a function of time. This module uses the internal PC clock to provide an interface that is compatible to the controller module (without of course any hardware instrumentation)

Sensor Module

- In order to provide linearisation of data in a general manner a module can call functions and data residing in a sensor module. This converts an input value into a conditioned output by a look up table and a binary search. The data table can be loaded into the sensor during its initialisation stage.

4 Implementation (All information here refers to DOS version 3.0 and above. Although many functions given here are undocumented in the early DOS versions, DOS 5.00 admits their existance and hence future versions will officially adhere to the present format. Versions 1 and 2 use are not compatible with all the functions give here)

Hardware Interrupt Vectors

Interrupt no	function
08h	Timer tick
09h	Keyboard
0Ah	Slave interrupt controller
0Bh	COM1
0Ch	COM2
0Dh	LPT2
0Eh	Floppy Disk
0Fh	LPT1

MSDOS interrupts and functions useful in memory resident programming

Interrupt	function	Description
1Bh	-	Ctrl-C BIOS function
21h	1Ah	Set disk transfer area (DTA)
21h	25h	Set interrupt vector
21h	2Fh	Get disk transfer area
21h	31h	Keep process
21h	33h	Get/Set Break-On flag
21h	34h	Get InDOS and DOSError flag addresses
21h	50h	Get programme segment prefix (PSP) address
21h	51h	Set programme segment prefix address
23h	-	Ctrl_Break handler
24h	-	Critical error handler (CEH)
28h	-	DOSIdle handler

ach module will install itself using Dos interrupt 21h function 31h . This enables the rogramme to terminate without deallaocating the memory reserved for it.
module can be removed from memory by using function 49h and putting the segment ddress of the Programme Segment Prefix in th ES register. Note that the environment lock also needs to be removed in a similar manner.
ny initialisation parameters are loaded from a text file at the installation time.
ignals between the modules can be passed using software interrupt handlers. At stallation each module attaches itself to a free interrupt. The signal number will be aced in the AX register and any other integer values passed in the other registers. lore sophisticted structures can be passed by the use of pairs of registers acting as ointers. Following a signal, a status word is usually returned in the AX register.
ROBLEM : The Master module must maintain a list containing (among other things) e interrupt numbers of each module installed. Therefore it must be installed first and very other module should register itself at installation by sending an appropriate gnal. However, during operation the Master must be supervising the other modules. hus the Master should also be memory resident and the software must designed ccordingly.
he Master must be activated periodically in order to coordinate bus activity. The asiest way to accomplish this is to hook onto the PC's timer which issues interrupts 3.2 times per second. This can be done in two ways :

(i) The timer interrupt provides a dummy interrupt #1Ch to allow any process to to hook on to the timer tick. The timer issues int 1Ch which usually returns immediately via an IRET. The user can redirect the interrupt vector for 1Ch to point at a new service routine.
PROBLEM : The 8259 Programmable Interrupt Controller requires a non-specific end of interrupt (20h to port 20h) to indicate that interrupts with lower priority can be serviced. The timer interrupt issues the EOI close to the end of the handler, prior to returning to the main programme. If the new handler for 1Ch is long (and generally it will be when a number of instruments are attached), the issue of the EOI will be delayed causing strange and wonderful effects (such as keyboard lockout). If the new handler itself issues the EOI the system crashes (by experience).

(ii) Replace the timer handler to int 8 by a new handler which immediately calls the old handler. This allows the EOI to be issued immediately the timer has performed its tasks and minimises any undesired effects.

(2) Hardware interrupts and DOS

- The timer is assynchronous and can therefore interrupt at any time. Especially dangerous times occur when DOS itself is interrupted as DOS is not reentrant. This is because DOS maintains several internal stacks: the IOStack used when executing functions 1-Ch (the character IO functions); the DiskStack used for most remaining functions; and the AuxilliaryStack used for exception (error) handling. Each stack is guaranteed to be large enough to hold one set of registers only. When Dos is interrupted and a second call to DOS is made the relevant stack will be overwritten by the second call which will eventually terminate normally. The first call, however, finds that its registers have been corrupted and will not terminate correctly, usually resulting in a system crash.
- Whenever a hardware interrupt occurs, DOS switches to a special stack set aside for this purpose. The size and number of such stacks can be set in the Config.Sys file with the STACKS command. However, the maximum size of any single hardware stack is 512 bytes and is totally insuffient for most high level languages. The user must therefore ensure that a stack of sufficient size is made available to any module linking itself to a hardware interrupt handler.
- Certain actions taken by the user, or errors that occur during programme execution can cause the current process to terminate (i.e. Ctrl-C, disk IO errors). If these occur during a hardware interrupt, DOS will attempt to restore control to the parent process (in general the command interpreter) . The results will be unpredicable, but a system crash often results. In any case a memory resident process should not be terminated by Ctrl-C.
- Most BIOS interrupts that control the low level PC functions are reentrant from the software aspect but because of the physical process they control, are effectively non reentrant. For example : A process is writing to the hard disk when it is interrupted by a second process that also writes to the disk. The second write moves the disk head to a new position and performs the operation. When finished it does not return the head to the position it found it and the first process continues writing blissfully unaware that the head is in the wrong position.

(3) Learning to live with DOS

- Is DOS safe ? : DOS is not reentrant and can not be made to be so. The only way around this is **NOT** to continue with a hardware interrupt if so to do would risk crashing the programme. Fortunately, DOS provides several flags that can be read to determine whether or not DOS is currently processing a system call and whether it is safe to interrupt. Function 31h returns a pointer to the "InDOS" flag which when set indicates that DOS is currently processing a call. The byte prior to this flag is the "DOSError" flag and is used to indicate whether DOS is processing an abnormal situation. Thus if either INDos or DOSError are set, DOS is not safe to interrupt and the interruption should be postponed until later. In some circumstances for instance when awaiting keyboard input, function 1 is issued, the INDos flag will always be set and the interruption will never occur. To deal with such situations, whenever a character IO function is issued and the system is waiting, interrupt 28h is repeatedly called by DOS itself. If the user wishes, the default handler can be replaced and the desired task undertaken. Any function with the exception of functions 1-Ch can be called from int 28h.

Handling Ctrl-C aborts : When Ctrl-C is pressed, BIOS interrupt 1Bh is called. This places the Ctrl-C character at the top of the keyboard buffer. The system is not immediately aborted, but when DOS is next called for any reason, the keyboard buffer is checked and the process aborted (by issuing int 23h) if the DOS function is 1-Ch or the BREAK flag is set. Abortion can be avoided by :

(i) Replacing the int 1Bh handler so that the Ctrl-C character is not recognised.

(ii) If no charcter IO functions are to be called from within the handler, the BREAK flag can be cleared for the duration of the handler using function 33H to both set and clear the flag.

(iii) Int 23h can be redirected to a new handler that clears the processor carry flag and returns. This prevents abortion but misses any Ctrl-C characters that occur whilst the handler is in effect.

Handling Hardware Errors : When a hardware error occurs the Critical Error Handler, int 24h is called (usually resulting in the famous message "Abort,Retry,Fail?"). Int 24h is expected to return in the AL register one of the following codes :

0: Ignore and return without error.
1: Retry the operation.
2: Terminate the programme.
3: Terminate and return with error code(let the application decide).

To prevent termination the int 24h handler should be replaced by a new handler that returns with the AL register cleared. It is perhaps prudent to indicate that an error has ocurred by setting an appropriate flag,as when DOS returns control, the user would be unaware that an error had ocurred.

Disk Access :

(i) The BIOS int 13h can, if neccessary be replaced by a handler that sets a semaphore and calls the old handler. Other processes wishing to use the disk services should first check the flag before accessing. This probably not neccessary as very few programmes use the disk without going through DOS (unlike the video functions int 10h) .

(ii) DOS needs the addresses of two structures maintained by each process for disk IO. These are the Programme Segment Prefix (PSP) and the Disk Transfer Area (DTA). Without these two structures essential file information will not be available. They can be obtained from DOS at installation time using functions 2Fh (Get DTA) and 51h (Get PSP). The functions 1Ah and 50h perform the inverse operations and can be used to set the values of the DTA and PSP prior to disk IO from within the interrupt handler.

Appendix H
Program Segment Prefix (PSP) Structure

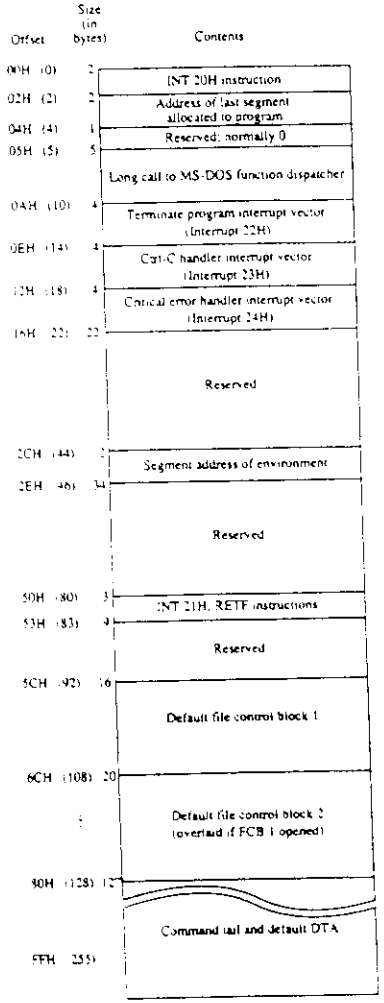


Figure H-1 (memory block diagram) illustrates the structure of the program segment prefix (PSP).

Figure H-1 Structure of the program segment prefix

(4) Coding the modules

- **Installing the device module**

```
procedure Install_Interrupt;
var DosCode : Integer;
    regs    : registers;
    Ok      : Word;
    Index   : byte;
begin
  {Set up buffer to hold data }
  for ok := 0 to BufferSize do Data_Array[index] := 0;
  {Get command line parameters and load set up data from file}
  Get_DCommandLine;
  {Save old interrupt function 35h}
  GetIntVec(IntNo,IntSave);
  {Set new handler using function 25h}
  SetIntVec(IntNo,@Int_Handler);
  {Get and set up pointers to IEEE functions etc}
  ok := Install_Device;
  State := POn;
  Err_State := No_Err;
  Con_Table := false;
  Writeln('Instrument # ',DeviceNo,' installed at int ',IntNo,' as device');
  with regs do
   begin
    AX  := $3100;
    DX := DSeg - PrefixSeg + (ofs(Data_End) div 16)+1;
    Intr($21,regs);
   end;
end;
```

```
function Install_Device:Word;
 var regs : registers;
 begin
  with regs do
   begin
   {Get pointers to IEEE functions}
    AX := 2;
    BX := 1;
    Intr(MasterInt,Regs);
    Spoll_Ptr := ptr(ES,DX);
    AX := 2;
    BX := 2;
    Intr(MasterInt,Regs);
    Xmit_Ptr := ptr(ES,DX);
    AX := 2;
    BX := 3;
    Intr(MasterInt,Regs);
    Send_Ptr := ptr(ES,DX);
    AX := 2;
    BX := 4;
    Intr(MasterInt,Regs);
    Enter_Ptr := ptr(ES,DX);
    {Register device and get DataNo}
    AX := 0;
    BX := DeviceNo;
    CX := IntNo;
    DX := Ofs(CurrentData);
    SI := ofs(Data_Array);
    ES := Seg(CurrentData);
    Intr(MasterInt,Regs);
    DataNo := ptr(ES,DX);
   end;
  Initialised := false;
  Install_Device := DeviceNo;
end;
```

- **The device interrupt handler**

```
procedure   Int_Handler(Flags,CS,IP,AX,BX,CX,DX,DI,SI,DS,ES,BP:Word);
Interrupt;
  begin
    Inline($FB);
    Case AX of
      $0  : AX := Return_Status;
      $1  : AX := Serial_Poll(Status);
      $3  : AX := Trigger_Device;
      $4  : AX := Initialise_Device;
      $7  : AX := DataNo^;
      $8  : begin
              ES := Seg(Data_Array);
              DX := Ofs(Data_Array);
            end;
      $20 : AX := Install_Table(BX);
      $21 : AX := Return_ErrState(ES,DX);
      $E0 : AX := Install_Device;
      $FF : AX := Kill;
    end{Case};
  end;
```

- **The status function**

```
function Return_Status:Word;
 begin
  case State of
    Idle,
    Waiting      : Return_Status :=  Get_Status(State);
    Data_Ready : Return_Status :=  Get_Data(State);
    POn          : Return_Status :=  Power_On(State);
    else           Return_Status :=  Get_Status(State);
  end{Case};
 end;


function Get_Status(var State:State_Type):Word;
 var Status_Bit:Word;
     ok         :Word;
 begin
   {Poll instrument and get staus byte}
   Poll := Serial_Poll(Status);
   if status <> 0  then
      begin
        State := Error;
        Err_State := Poll_Err;
        Status_Bit := ErVal;
      end
   else case State of
        Idle         : Status_Bit := DIVal;
        Waiting      : begin if ((poll and DAMask) = DA)
                       then begin
                              Status_Bit := DRVal;
                              State := Data_Ready;
                            end
                       else Status_Bit := DWVal;
                     end;
        Data_Ready   : Status_Bit := DRVal;
        POn          : Status_Bit := Initialise_Device;
        Error        : begin
                         State := Idle;
                         Status_Bit := ErVal;
                       end;
     end{Case};
  Get_Status := Status_Bit;
end;
```

- **The Master module**

```
procedure Install_Master;
 var DosCode : Integer;
    Ok     : Word;
    Dummy   : Word;
 begin
  Stack_Seg := SSeg;
  {Get parameters}
  Get_CommandLine;
  {Initialise device registry}
  Ok := Register_Device(0,0,Dummy,Dummy,Dummy);
  ok := Reset_Time;
  {Initialise variables}
  Data_Available       := false;
  Bus_Ready      := false;
  Bus_Error            := false;
  Bus_Done       := false;
  Busy                 := false;
  Stop                 := true;
  Service_Wanted       := false;
  Err_State            := No_Err;

  {Dissable interrupts while setting up module}
  Inline($FA);
  {Get INDOS and DOSERROR Flag addresses}
  with regs do
   begin
    AX := $3400;
    Intr($21,regs);
    InDos := ptr(ES,BX);
    DosError := ptr(ES,BX-1);
   end;
  {Install various hardware and software interrupt handlers}
  GetIntVec($8,TimerIntSave);
  GetIntVec(IntNo,CommandIntSave);
  GetIntVec($28,DosIdleSave);
  SetIntVec($8,@Timer_Handler);
  SetIntVec(IntNo,@Command_Handler);
  SetIntVec($28,@Service_Handler);

  Inline($FB);
  Writeln('AMASS Master module installed at int # ',IntNo);
  SwapVectors;
  Keep(DosCode);
end;
```

- **Interrupt handlers**

```
procedure Command_Handler(Flags,CS,IP,AX,BX,CX,DX,SI,DI,DS,ES,BP:Word);
Interrupt;
 Begin
  {Re-enable interrupts}
  InLine($FB);
  Case AX of
    0  : AX := Register_Device(BX,CX,DX,SI,ES);
    1  : AX := Start_Bus(BX,CX);
    2  : AX := IEEE_Funcs(BX,DX,ES);
    3  : Bus_Done := true;
    4  : AX := Return_Device(BX,CX,DX,ES);
    5  : AX := Return_Time(DX,ES);
    6  : AX := Reset_Time;
    7  : Initialise_Bus;
    8  : AX := Device_List.D_No[BX];
    9  : AX := Return_DataNo(DX,ES);
    $A : AX := Return_State(BX,CX,SI,DX,ES);
    $B : AX := Register_Table(BX);
    $20 : AX := StringToFloat(BX,CX,DX,ES);
    $21 : AX := FloatAdd(CX,DX,ES);
    $22 : AX := FloatCompare(BX,CX,DX,ES);
    $25 : AX := Set_CntrlVals(BX,CX,DX,SI,DI,ES);
    $26 : AX := GetValStr(BX,CX,DX,ES);
    $30 : AX := OpenFile(CX,DX,ES);
    $31 : AX := AppendData(BX,CX,DX,ES);
    $42 : Hold  := true;
    $43 : Hold  := false;
    $50 : AX := Return_ErrorPtr(ES,DX);
    $51 : AX := Reset_Error;
    $E0 : AX := Install_Devices;
    $FE : AX := Kill_Device(BX);
    $FF : AX := Kill_All;
  end{Case};
end;
```

```
procedure Master_Function;
 var X,Y:word;
 begin
  {stop interrupts}
  Inline($FA);
  if DosError^ <> 0 then Exit;
  Busy := true;
  Set_Stack;
  SwapVectors;
  SetCtrl_C;
  Service_Wanted :=  false;
  Count := 0;
  if ((DataNo >= MaxData) or Bus_Done) then
   begin
     State := Done;
     Stop  := true;
   end;
  {Allow interrupts}
  Inline($FB);
  State_Action;
  ResetCtrl_C;
  SwapVectors;
  InLine($FA);
  Reset_Stack;
  busy := false;
end;
```

```
procedure Timer_Handler;
Interrupt;
 var Index : Word;
    regs  : registers;
 begin
   {Call old timer interrupt and reset hardware}
   InLine($9C/$FF/$1E/TimerIntSave/$FB);
   Timer;
   if busy then exit;
   if Hold then Exit;
   if stop  then Exit;
   if ((InDos^ = 0) and (DosError^=0) )
     then Master_Function
     else Service_Wanted := true;
 end;

{Hander for DOS int 28h}
 procedure Service_Handler;
 Interrupt;
 begin
   if Service_Wanted then Master_Function;
 end;
```

```
procedure State_Action;
 var ok : word;
 begin
  if State <> Error then LastState := State;
  Case State of
   Idle      : begin
                if Bus_Ready then
                 begin
                   State := Waiting;
                   Status := Trigger_Bus;
                   Exit;
                 end;
                Status := Get_Status;
                If Bus_Error then State := Error;
               end;
   Waiting   : begin
                Status := Get_Status;
                if Bus_Error then State := Error else
                if Data_Available then State := Data_Ready;
               end;
   Data_Ready : begin
                 if not Pause then
                  begin
                    if Sync_Trigger <> 0  then
                     begin
                       State := Error;
                       Err_State := Device_Err;
                       Exit;
                     end;
                    State := Idle;
                    Bus_Ready := false;
                    Data_Available := false;
                    Bus_Error := false;
                   end;
                 end;

    Error     : begin
                    If Err_State = Disc_Err then Exit;
                    Err_State := Device_Err;
                    Status := Get_Status;
                   if not Bus_Error then State := Idle;
                 end;
    Done      : Status := Start_Bus(0,0);
   end{Case};
  end;
```

(5) **Testing the software**

- Warning : Memory resident programming can cause premature aging. Most normal software debuggers can not enter interrupt service routines and therefore the tools available for normal programmes do not generally exist
- Whenever possible try out the functions from within a normal programme first and then move them function by function (in some cases line by line) into the memory resident module.
- To check whether a programme is loading properly and to find out how much space it occupies use software utilities like MI or Chkdsk
- For intermodule communication, load one module to memory and the second module as a normal programme. Step through the second using a debugger paying special attention to the codes returned by each module call
- Develop a purging programme that removes a module from memory as early as possble
- Finally when you can not find any more bugs, give the sytem to someone who does not understnd memory resident programming. New bugs will rapidly appear!

(6) **Installing and monitoring the system**

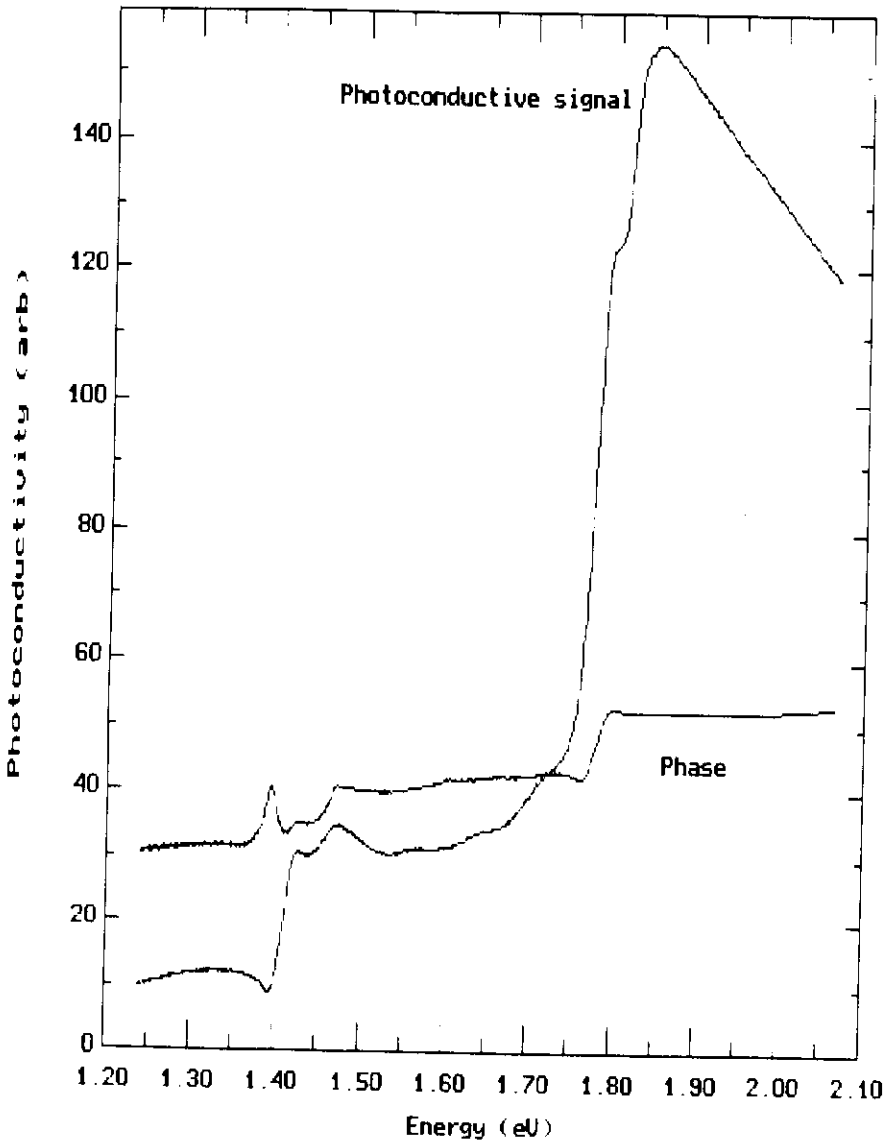
- Software that helps the user to provide the necessary run time parameters (such as which modules to load, the values of the particular settings needed for the given experiment etc) and writes them to a file, can facilitate the loading process
- Use a foreground process to start and stop the acquisition of data and to monitor the progress of the experiment. This can be a simple programme that waits for a one of a set of keys to be pressed and sends a signal to the Master to perform the required action. A more sophisticated monitor could display the progress of an experiment graphically. In the present implementation the Master module returns pointers to the data buffers of each installed module together with a pointer to the current data number. The foreground process plots each new point by monitoring any change in the data number. When a change occurs a new ponit is plotted.
- As the complete acquisition package is memory resident (with teh exception of the monitor), once it has been started it is not necessary to run the monitor and any normal programme can be substituted for the foreground task. This can sometimes be of use.

Performance

- Using the techniques outlined here, a data acquisition system can be constructed. Data obtained in an experiment to measure the modulated photoconductive current in a GaAs/GaAlAs multiple quantum well as a function of the exciting light wavelength is presented in the figure. This experiment used the following instruments :

  Controller 1 Home made interface to monochromator
  Controller 2 Oxford Instruments 3120 Temperature controller
  Controller 3 Keithley Instruments Programmable Voltage source
  Device 1 Stanford Research SR530 LIA
  Device 2 Stanford Research SR530 LIA

  The experiment was run on an IBM XT



Useful Bibliography

Dos Programmers' Reference Manual, Microsoft Press, 1990.

The MS-DOS Encyclopedia, Microsoft Press, 1988

J. Prosise, PC Magazine, p 313, April 1987.

D. Rollins, PC Tech Journal, p 130, April 1987.