APPLICATION NOTE

AN454
Interfacing the 83C576/87C576 to the ISA bus

December 21, 1994

Philips Semiconductors

PHILIPS

Interfacing the 83C576/87C576 to the ISA bus AN454

INTRODUCTION

The most interesting feature of the Philips 83C576, and the principal subject of this application note is its Universal Peripheral Interface (UPI). The UPI is a microprocessor slave interface which allows the ’576 to communicate with a microprocessor or microcontroller host with minimal support logic. The UPI acts as a ”bus gasket” between the 8051 core inside the ’576 and the host. Commands and Data can easily be exchanged over this interface. Along with the hardware interface, a simple, effective bidirectional software protocol can be implemented for reliable data transfer. Each device can pace the exchange of data using a double bi–directional data register implemented as part of the UPI inside the ’576. As part of the UPI definition, hardware flow control is supported so that both the host and 8051 core can each establish whether the other has written or read any data to the UPI. This flow control scheme is the heart of a synchronous interface that is independent of the performance of the host or 8051 core.

The UPI is ideal for the PC environment and provides an almost seamless interface to the PC host via the ISA bus. This application note demonstrates in both hardware and software terms how to interface the ’576 to a PC host. The UPI interface occupies two locations in the memory or IO space of each device. The first location is the Data register and the second is the Status register. The Data register is simply a pair of registers one directed to the host and the other directed to the 8051 core. This allows both the host and 8051 core to write to the data register simultaneously without effecting each–others data. The write cycle events are recorded as ‘buffer flags’ (IBF and OBE) in the Status register. One further unique feature of this interface is the concept of ‘Commands and Data’ in that any data written to the Status register address is directed to the Data register and a flag is set in the Status register recording this cycle as a Command. This flag is known as the ‘AF’ flag and indicates that a write cycle has been performed on the Status register.

From a software perspective the host device is the ‘master’ of the UPI. Even though from a hardware sense, each device can pace the cycles, the provision of a single AF flag precludes the implementation of a multi–master protocol. This limitation, however, is not severe, in that this provides a simple scheme for exchanging data between the devices. This interface provides the mechanisms for many protocol schemes, the most effective one is used in PCs over the keyboard interface and operates as follows:

HOST (Master)	SLAVE (’576 core)
Write a command to the Status register.	
	AF flag set, the byte in the data register is a Command
Poll the IBF flag in the Status register; wait for it to be 0.	
	Read the command.
Write the Information byte to the Data register.	
	AF=0, read the information byte from the Data register; Execute the Command.
Poll OBE, wait for it to be cleared in the Status register; this means that a response is available in the data register.	
	Write a Response byte to the data register.
Read the Response byte.	

DESCRIPTION

The purpose of this Application Note is to demonstrate the use of the ’576 in a PC environment. This example shows the use of a ’576 as a peripheral to the PC/AT. The example shows the implementation of a data acquisition card with digital IO and analog IO. Some software is described to show how to exchange data and commands between the PC and the ’576. In this example, the PC is the host of the transfers and the ’576 is the slave. An example circuit is also described which details the electrical interface to the PC/AT ISA bus.

Figure 1 shows the basic configuration of the acquisition system.

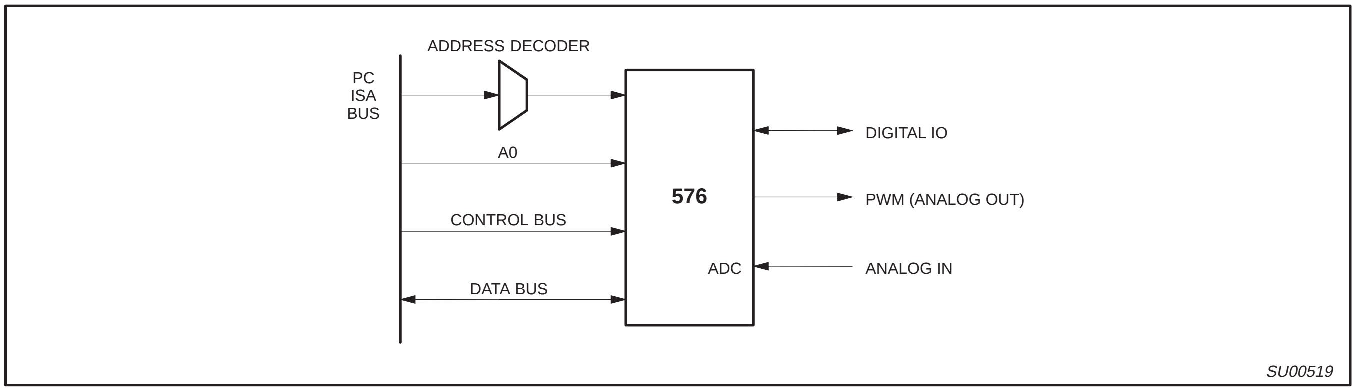


Figure 1. Block Diagram

UPI DEFINITION

The Universal Peripheral Interface (UPI) functions as a microprocessor slave interface. This allows the ’576 to interface directly to a microprocessor bus as a slave or peripheral device.

The UPI is an 8–bit bidirectional data register, P0, with an associated status register, UCS. The data buffer is comprised of two registers; the input register and, the output register. The input register can be written by the host and read by the ’576. The output register can be written by the ’576 and read by the host. The status register may be read or written by the ’576 core but may only be read by the host. The Status register bits can also be affected by hardware events, for example, a host write cycle to the data register will set the IBF flag.

The host control interface for these registers is comprised of four signals, –RD, –WR, an address line A0, and a chip select –CS. Data transfer is directed to the UPI as shown below:

–CS	A0	–RD	–WR	CONDITION
0	0	0	1	Read Output Data Register
0	1	0	1	Read Status Register
0	0	1	0	Write Input Data Register, AF=0
0	1	1	0	Write Input Data Register, AF=1
1	X	X	X	Disable IO

Write cycles to the data register with A0 = 1 cause the AF flag to be set in the status register. These cycles are generally interpreted as commands. Write cycles to the data register with A0 = 0 cause the AF flag to be cleared in the status register. These cycles are usually interpreted as data.

The status register has 4 control bits and 4 user defined status bits. They are defined as follows:

UCS

7	6	5	4	3	2	1	0
ST7	ST6	ST5	ST4	UE	AF	IBF	OBE

UCS.7	ST7	User defined status bit
UCS.6	ST6	User defined status bit
UCS.5	ST5	User defined status bit
UCS.4	ST4	User defined status bit
UCS.3	UE	UPI enable bit. 0 = Disabled, 1 = Enabled.
UCS.2	AF	Address Flag – contains the state of the A0 (Address) pin on the last write cycle. 0 = write cycle with A0 cleared 1 = write cycle with A0 set
UCS.1	IBF	Input Buffer Full Flag – set by hardware on the rising edge of a write command to the Input Data Register. Cleared by hardware on the completion of a read cycle of the Input Data Register by the ’576.
UCS.0	OBE*	Output Buffer Empty Flag – Cleared by hardware on the completion of a write cycle to the Output Data Register by the ’576. Set by hardware on the rising edge of the read command of the Output Data Register by the host.

* **NOTE:** This flag is OBE when read by the MCU, but is inverted or OBF (Output Buffer Full) when read by an external host.

General Handshaking Protocol

Host write cycles with A0 set are directed to the Input Data Register. The ’576 can distinguish the cycles by monitoring the AF flag (Address Flag) in the status register.

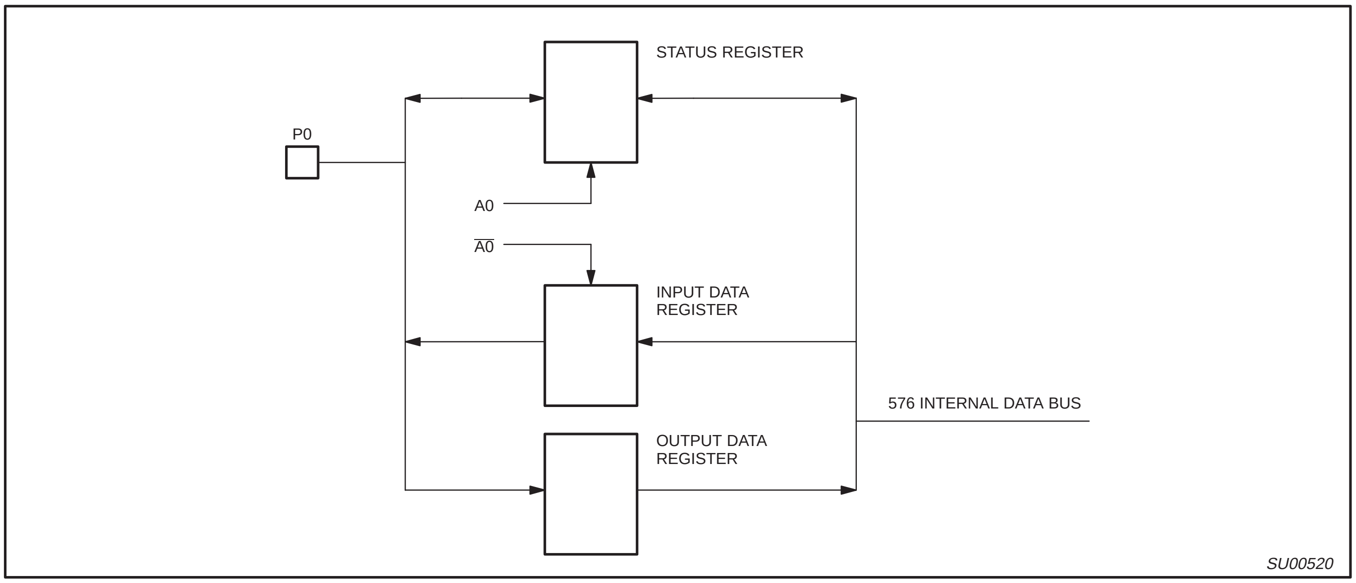


Figure 2. Internal Structure of the UPI Registers

ISA BUS INTERFACE

The data acquisition system is comprised of the ’576 microcontroller and an address decoder. The ’576 provides an almost seamless interface to the PC/AT IO channel. The interface to the PC is known as the ISA (Industry Standard Architecture) bus. This bus is an asynchronous, command driven bus. Most notebook and PDA chipsets provide programmable or fixed address decoders assigned to an output pin. This pin is used to control the –CS input of the ’576 thus providing a completely seamless interface to the ISA bus.

Address Path

In this application the ’576 is interfaced as an IO device. The IO space on a PC is 0 to 03FFH (1Kbytes), therefore SA0 to SA9 must be decoded to locate the ’576 in a single region. SA0, however, is used to select between the Data and Status registers, thus the ’576 occupies two locations in IO space. For the acquisition card we selected 0300H for the Data register and 0301H for the Status register. 300H and 301H have been chosen because they are assigned to a ”prototyping area” in the PC ISA IO space map. The address decoder must reject the address during DMA cycles. For this purpose, AEN is used in the decoder and qualified for its low, inactive state (AEN, when active indicates that the current cycle is a DMA cycle). Output Y0 of U2 is asserted (low) when SA5 to SA7 are low, SA8 is high and AEN is low. U3 further decodes the address and output Y0 is asserted (low) when Y0 of U2 is low together with SA1 to SA4 low and SA9 high. Output Y0 of U3 is fed to the –CS input of the ’576, it is asserted (low) when the address is 300H with AEN low or 301H with AEN low.

Data Path

The ’576 interfaces to the least significant 8 bits of the data bus, SD0 to SD7. When the ’576 is configured in UPI mode, PORT0 is configured as push–pull outputs for host read cycles. ISA requires 24mA I_{OL} for this interface. The ’576 can only handle 15mA. The 24mA requirement goes back to the XT–TTL days and with current CMOS motherboards 12mA is probably sufficient with a fully loaded system.

Interrupt

An interrupt is employed to completely demonstrate the integration of the ’576 to the PC. The first free interrupt is IRQ10. For this reason alone, we need to connect to the extended AT ISA slot where this interrupt line is available. Interrupts on PCs are edge sensitive, sometimes shared and are usually pulled–up on the motherboard just to make life more complicated. If we want to generate an interrupt, port pin P2.3 must be driven low then high. The 8259 interrupt controller in the PC will see the interrupt on the rising edge. Once the host acknowledges the interrupt by say, reading the Data register (detected by the assertion of OBE) the ’576 must drive P2.3 high, returning IRQ10 to a high impedance state.

Timing

Another aspect of the design is to consider the timing implications of the UPI. The diagrams below show the relationships of the control, address and data signals.

SYMBOL	PARAMETER	LIMITS
t_{AS}	Address Setup time	21ns MIN
t_{PW}	Command pulse width	600ns MIN (assumes 8.33MHz bus)
t_{DS}	Data setup time	15ns MIN
t_{DH}	Data hold time	15ns MIN
t_{CR}	Cycle recovery time	55ns MIN

The address setup time is lengthened by the two 3 to 8 line decoders, U2 and U3. The ISA worst case is 9ns, we are adding 2 further delays of 6ns giving 21ns of address setup. If the ISA bus is clocked at a higher rate as in some PDAs and some notebooks, the Command pulse width will be shortened. The ISA bus can be clocked as high as 11.1MHz, yielding a minimum command pulse width of 450ns.

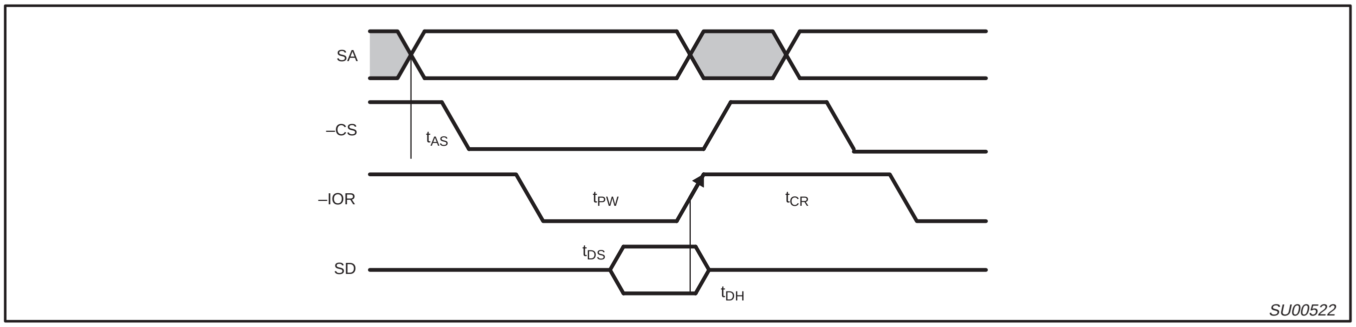


Figure 3. ISA Cycle IO Read Timing

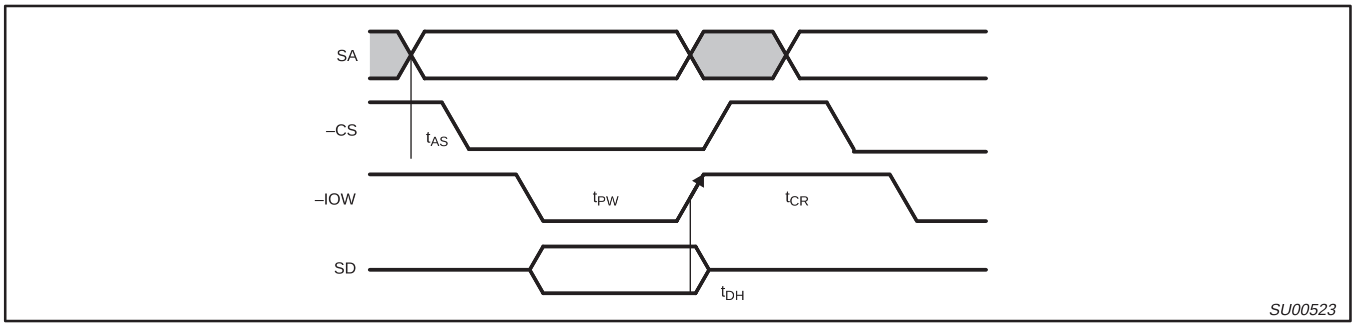


Figure 4. ISA Cycle IO Write Timing

SOFTWARE

The example below shows a typical protocol employed here to exchange data between the PC and the ’576.

The protocol that we are employing is only a subset of what can be achieved with this Command / Data, two byte protocol. To complete a transaction, the PC sends two bytes, a Command and Information byte, then the ’576 returns two bytes; a response and information byte. Sometimes the information bytes will be meaningless; that’s okay, as long as we stick to the protocol.

COMMANDS

	Command	**Information**	
01H	Get Analog Channel	Channel	[0, 1, 2, 3]
02H	Get Digital Byte	NULL	
03H	Put Digital Byte	Data Byte	
04H	Increase PWM	Channel	[0, 1]

RESPONSES

	Response	**Information**
01H	Get Analog Channel Complete	M.S. 8 bits of ADC conversion
02H	Get Digital Byte Complete	Data Byte
03H	Put Digital Byte Complete	NULL
04H	Increase PWM Complete	NULL

DRIVERS

PC Driver

The PC must write a command. This is done by making an IO cycle to port 0301H. This sets the AF flag in the status register and causes an IBF interrupt in the ’576. The PC can poll the status register to see when the IBF flag has cleared, this means that the ’576 has read the Command in the Input Data Register. The PC then sends the Information byte; this is done in the same way except that the cycle is made to 300H.

576 Driver

Communication from the ’576 to the PC is established by driving the interrupt request line, IRQ10, high. The PC first reads the status register to check if the interrupt was from the ’576 instead of some other external device. If the OBF flag is set, the PC knows that the data in the Output Data Register is valid. The PC then reads to Output Data Register which causes the OBF flag to become cleared. Once the ’576 detects this read cycle, the interrupt line, P2.3, can be returned to its low state. The AF bit in the Status register indicates whether the data is a Response byte to a previous command (AF set) or an associated Information byte (AF cleared).

```
/*
         Code to demonstrate the use of the UPI to the ’576
         in an interrupt driven mode.

         Written in Microsoft C7.0 for PC AT

*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <dos.h>
#include <conio.h>

#define UC      unsigned char
#define UL      unsigned long
#define UI      unsigned int

#define P8259A_IMR     0x21     /* ;interrupt mask register i/o address    */
#define P8259B_IMR     0xa1     /* ;interrupt mask register i/o address    */
#define EOI            0x20
#define P8259A         0x20
#define P8259B         0xa0

#define ENABLE_IRQ10            ~0x04
#define ENABLE_IRQ2             ~0x04

#define UPI_DATA_REG            0x300
#define UPI_STATUS_REG          0x301
#define IBF                     0x02
#define OBF                     0x01

unsigned char interrupt_received = 0;

/* declare a variable to store the pointer of the current interrupt service routine */
void (_interrupt _far *old_int)();

/*Module put_byte().

put_byte() has two parameters; the address of the IO port (port) and the value to be written to the IO
port (data). A dummy variable is declared for lint purposes, since the C function outp() returns an
unsigned short. First put_byte() uses the C oupt() function to write the byte to the IO port then,
put_byte() monitors the IBF flag in the status register to see if the ’576 has read the byte. When the PC
writes the byte to the input register, the ’576 automatically sets the IBF flag. When the ’576 core reads
the input register, the IBF flag is automatically cleared. The while() loop breaks when the IBF is 0. The
C function inp() is used to read the status register, the returned value is logically ANDed with the
constant IBF (0x02) and the result is complemented and evaluated by the while() statement. */

void
put_byte (
         unsigned short port,
         unsigned char data) {

         unsigned short dummy;

         dummy = outp(port, UPI_DATA_REG);        // send the byte
         while (~(inp(UPI_STATUS_REG) && IBF));   // check to see if ’576 has read it

}
```

```
/*Module get_byte()

get_byte() has 1 parameter and 1 return value. The parameter is used to pass the address of the IO port
to read. The return value is used to pass back the contents of the read port as a byte. get_byte first
waits for the byte to be written to the UPI port by the ’576. This is achieved by monitoring the OBE flag
in the status register. The C function inp() is used to read the contents of the status register. The
returned word is masked (ANDed) with the constant OBF (0x01) and the negated result is evaluated by the
while() statement. If the IBF flag is 0, the while loop continues to read and evaluate the status
register, if the IBF flag is set the while loop breaks.
When the while loop breaks, there must be something to read in the output data register, the C function
inp() is used to read the input data register whose address is past in the variable ’port’. The result is
cast as a byte and returned to the calling routine. */

unsigned char
get_byte (

         unsigned short port) {



         while (!(inp(UPI_STATUS_REG) && OBF));               // wait for the byte
         return ((byte)inp(port));
}

/*Module send_command()

send_command assembles a two byte message comprising of a command byte follwed by an information byte. If
we direct the first byte to the address of the status register, it will appear in the output data
register and the AF flag will automatically be set. The ’576 will read the status register before the
data register, the set AF flag will tag the data as a command. The information byte is directly written
to the output data register, thus the AF flag will be cleared and the ’576 will interpret the byte as
information. */

void
send_command (
         unsigned char command,
         unsigned char information ) {

         put_byte(UPI_STATUS_REG,command);
         put_byte(UPI_DATA_REG,information);
}

/* Module get_response()

This module has no parameters and 1 return value. get_response strips out the information byte from a
two–byte message from the ’576. The first byte is always the echoed command and is discarded by assigning
it to a dummy variable. The send byte is always the information and is read using the get_byte() function
and returned to the calling function. */

unsigned char
get_response (
               void ) {

         unsigned char dummy;

         dummy = get_byte(UPI_DATA_REG);         // get the response
         return(get_byte(UPI_DATA_REG));         // return the info byte
}
```

```
/* Module EnableIRQ10()

This module enables the hardware interrupt, IRQ10 via the 8259 programmable interrupt controller (PIC)
insude the PC. IRQ 10 is a cascaded interrupt driven by a sencond PIC attached to IRQ2 of the first.
Therefore both PICs need to be unmasked. The C function disable() issues a CLI instruction to the
processor disabling interrupts. The current value of the interrupt mask for PICA is read and ANDed with
the mask cleared for IRQ2. The current value of the interrupt mask for PICB is read and ANDed with the
mask cleared for IRQ10. The new masks are written to the PICs consecutively. Interrupts are re–enabled
using the C function enable() which issues an STI instruction to the processor. */

void
EnableIRQ10 (void) {

              unsigned short mask;

              _disable();
              mask = _inp(P8259A_IMR);
              mask &= ENABLE_IRQ2
              _outp   (P8259A_IMR,mask);

              mask = _inp(P8259B_IMR);
              mask &= ENABLE_IRQ10;
              _outp   (P8259B_IMR,mask);
              _enable();
}

/* Module Irq10_isr()

This module is the service routine for an IRQ10 interrupt. It simply sets a global flag which is
monitored by the main loop and then issues a ”non specific end–of–interrupt (EOI)” to each PIC. The EOI
flags are written to reset the interrupt signal that the PICs assert to each other and the processor.”

void
__cdecl __interrupt __far Irq10_isr(void) {

         interrupt_recieved = 1;

         _outp(P8259A,EOI);
         _outp(P8259B,EOI);
}
```

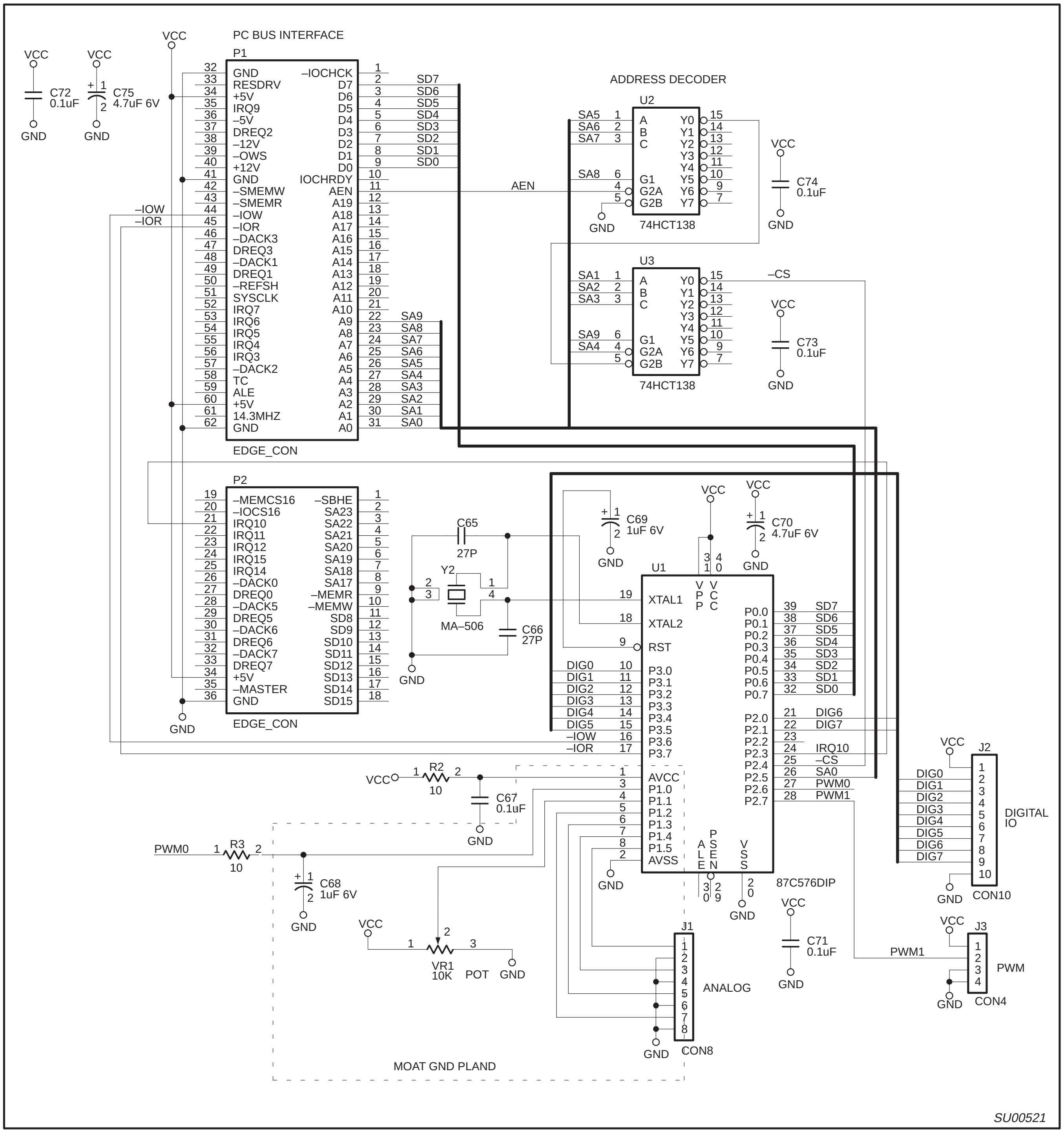
```
void
main(void) {

         printf(”\n576 Interrupt Interface Utility\n”);

/* save the pointer of the current interrupt service routine using the C function _dos_getvect() */
         old_int = _dos_getvect(0x12);

/* store the pointer of our interrupt service routine (Irq10_isr) using the C function _dos_setvect() */
         _dos_setvect(0x12, Irq10_isr);

/* clear the global flag */
         interrupt_received = 0;

/* unmask and enable interrupts */
         EnableIRQ10();


         printf(”\nGet Analog Channel      01 00 returned ”);
/* send the command */
         send_command(1,0);
/* wait for an interrupt */
         while(~interrupt_received);
/* print the result on sdtio (the screen) */
         printf(”%02x”,get_response());
/* clear the global interrupt flag */
         interrupt_received = 0;

         printf(”\nGet Digital Byte        02 00 returned ”);
         send_command(2,0);
         while(~interrupt_received);
         printf(”%02x”,get_response());
         interrupt_received = 0;

         printf(”\nPut Digital Byte        03 55 returned ”);
         send_command(3,0x55);
         while(~interrupt_received);
         printf(”%02x”,get_response());
         interrupt_received = 0;

         printf(”\nSet PWM                 04 01 returned ”);
         send_command(4,1);
         while(~interrupt_received);
         printf(”%02x”,get_response());
         interrupt_received = 0;

/* restore the old interrupt vector */
         _dos_setvect(0x12, old_int);

}
```

```
/*
              Code to demonstrate the use of the UPI
              on the ’576.

              Written in Franklin C for the ’576.
*/

#include <reg51.h>

// Constants

#define UC    unsigned char
#define UI    unsigned int

// Special Function Registers

sbit    IRQ              = P2^3;

sfr     UCS              = 0x86;

sbit    UE               = UCS^3;
sbit    AF               = UCS^2;
sbit    IBF              = UCS^1;
sbit    OBE              = UCS^0;

sfr     ADC              = 0xB1;
sbit    ADF              = ADC^7;
sbit    ADCE             = ADC^6;
sbit    AD8M             = ADC^5;
sbit    AMOD1            = ADC^4;
sbit    AMOD0            = ADC^3;
sbit    ASCA2            = ADC^2;
sbit    ASCA1            = ADC^1;
sbit    ASCA0            = ADC^0;
sfr     ADC0H            = 0xAA;
sfr     ADC1H            = 0xAB;
sfr     ADC2H            = 0xAC;
sfr     ADC3H            = 0xAD;
sfr     ADC4H            = 0xAE;
sfr     ADC5H            = 0xAF;
sfr     PWM0             = 0xBE;
sfr     PWM1             = 0xBF;
sfr     PWMP             = 0xBD;
sfr     PWCON            = 0xBC;
```

```
/* Module get_analog_channel

This module initializes the ADC for mode 0 and 8 bit conversions. This module has 1 parameter and 1
return value. The channel to be converted is passed as an unsigned char parameter and converted to 3
single bit values. The converted values are set in the ADC control register. The conversion is started by
setting the enable bit (ADCE). The conversion is monitored for completion by polling the ADF bit. The
result from the selected channel is returned to the calling function using the switch() statement */

UC
get_analog_channel(
         UC channel ) {

         AMOD0 = 0;       // Mode 0
         AMOD1 = 0;
         AD8M = 1;        // 8 bit mode
         ASCA2 = channel^2;
         ASCA1 = channel^1;
         ASCA0 = channel^0;
         ADCE = 1;        // start conversion

/* poll the ADF flag. This loop breaks when ADF = 1 */
         while(~ADF);     // wait for conversion to complete

/* return the ADC value based on the selected channel */
         switch (channel) {
                  case 0:
                           return(ADC0H);
                           break;
                  case 1:
                           return(ADC1H);
                           break;
                  case 2:
                           return(ADC2H);
                           break;
                  case 3:
                           return(ADC3H);
                           break;
                  case 4:
                           return(ADC4H);
                           break;
                  case 5:
                           return(ADC5H);
                           break;
                  default:
                           return(0xFF);
                           break;
         }
}
```

```
/* Module get_digital_byte()

This module has 1 parameter and 1 return value. The parameter is included for consistency with the other
functions and has no meaning in this case. The return value is the combined result of reading 6 bits from
P3 and 2 bits from P2. */

UC
get_digital_byte (UC dummy) {

// declare a temporary variable
         UC byte;

// read P3 and strip the 2 M.S. bits
         byte = P3 && 0x3f;

// read P2, strip 6 M.S. bits, shift result 6 places left and OR with the temporary variable
         byte |= ((P2 && 0x03) << 6);

// return the result
         return (byte);
}

/* Module put_digital_byte()

This module take the byte, passed as a parameter, masks the unused bits of the ports and ORs the masked
result with the current port values. */

UC
put_digital_byte (
         UC byte ) {

// clear the current port value
         P3 &= 0xc0;
// AND the parameter with a mask and OR the result with the cleared port value
         P3 |= (byte && 0x3f);
         P2 &= 0x3f;
         P2 |= ((byte && 0xc0) >> 6);
         return(byte);
}

/* Module increase_pwm()

This module increments the current PWM value of the selected channel. The channel is passed to this
function as a parameter */

UC
increase_pwm (
         UC channel ) {

// test the channel parameter
         if (channel == 0)
// increase PWM0
                  PWM0++;
         else
// increase PWM1
                  PWM1++;

         return(channel);
}
```

```
void
main () {
// declare temporary variables for the command and information bytes
         UC command;
         UC information;

// initialize the PWM controller
         PWMP = 0x80;
         PWCON = 0xFF;

// do forever
         while(1) {

// wait for a command from the PC ie, when the IBF flag is set
                  while (~IBF);           // wait for the command

// read the command byte. This read clears the IBF automatically
                  command = P0;

// wait for the next byte from the PC
                  while (~IBF);           // wait for the information
                  information = P0;

// echo the command
                  P0 = command;
// wait for the PC to read it
                  while (OBE);

// parse the command and call the appropriate function, passing the information as a parameter
                  switch (command) {
                           case 1 : {
                                    P0 = get_analog_channel(information);
                                    break;
                           }
                           case 2: {
                                    P0 = get_digital_byte(information);
                                    break;
                           }
                           case 3: {
                                    P0 = put_digital_byte(information);
                                    break;
                           }
                           case 4: {
                                    P0 = increase_pwm(information);
                                    break;
                           }
                           default: {
                                    P0 = 0xFF;
                                    break;
                           }
                  }
// wait for the PC to read the information.
                  while (OBE);
         }
}
```


PC BUS INTERFACE
P1
EDGE_CON
P2
ADDRESS DECODER
U2
74HCT138
U3
74HCT138
U1
87C576DIP
C72 0.1uF
C75 4.7uF 6V
C74 0.1uF
C73 0.1uF
C65 27P
C66 27P
Y2 MA–506
C69 1uF 6V
C70 4.7uF 6V
R2 10
C67 0.1uF
R3 10
C68 1uF 6V
VR1 10K POT
C71 0.1uF
J1 ANALOG CON8
J2 DIGITAL IO CON10
J3 PWM CON4
MOAT GND PLAND
AEN
SU00521

Philips Semiconductors
811 East Arques Avenue
P.O. Box 3409
Sunnyvale, California 94088–3409
Telephone 800-234-7381