APPLICATION NOTE

AN435
I^{2}C byte oriented system driver

Author: Joe Brandolino, Philips Semiconductors FAE

1994 Oct 25
Supersedes data of 1993 Jun 01
IC20 data handbook

Philips
Semiconductors

PHILIPS

I^{2}C byte oriented system driver AN435

Author: Joe Brandolino, Philips Semiconductors FAE, Canada

DESCRIPTION

IIC_OS2.ASM contains a complete multi–master I^{2}C driver for the byte oriented Philips microcontrollers. To date, the list of byte oriented 80C51 derivative microcontrollers includes:

– 8XC552
– 8XC562
– 8XC652
– 8XC654
– 8XCL410
– 8XCL580
– 8XCL781

The I^{2}C Bus is a deceptively simple concept. With only two lines involved – the data line (SDA) and the clock line (SCL) – one would think that writing an I^{2}C driver is a trivial task. In reality, a complete multimaster capable I^{2}C driver would be a complex state machine (the I^{2}C hardware state machine could assume one of 28 possible states).

The idea behind IIC_OS2 is to make the state machine workings of the I^{2}C mechanism transparent to the user. When this driver is incorporated into a program, the user communicates with I^{2}C peripherals (and other masters) by executing a command file which contains simple macro directives. For example, to send a byte of data (stored in 'the_data') to a PCF8574 I/O expandor (having I^{2}C address 'PCF8574_adrs') using this driver, the main program would look like this:

```
;PROGRAM:
;       MOV     DPTR,#COMMAND_FILE               ;load address of command file
;       CALL    DO_IIC                                 ;call ”IIC_OS2” routine
;       JBC     IIC_failure, PROGRAM                      ;check for a failure
;               .
;               .
;               .
;COMMAND_FILE:
;       DB      PCF8574_adrs OR iic_write_mask     ;address of slave + R/W bit
;       DB      ioD_                            ;define where to get data from
;       DB      1                              ;define number of bytes to send
;       DB      the_data                    ;define address of data to be sent
;       DB      iic_end_                      ;’end of command file’ directive
;
```

This example describes one option out of many which can be used to send the data byte to a slave. Without a driver like "IIC_OS2", the user would have to interact with the IIC SFR registers, and take into account all hardware state possibilities.

The comments in this listing assume that the reader has a basic knowledge of the 80C51 family, and is familiar with IIC basics. This program has been tested as thoroughly as time permitted; however, Philips cannot guarantee that this I^{2}C driver is flawless in all applications.

The comment text fields in this file use a consistent method of highlighting the various parameters of the software. All constants (EQUates), registers, bits and other bytes are surrounded by ' ' in the comment text. All routines, labels, procedures and file names are surrounded by ” ” in the comment text. Generally speaking, all 8051 mnemonics are in UPPERCASE, all variable names and labels are in LOWERCASE or MiXeD case. All EQUates are named such that the last character in the EQUate name is an underscore (e.g., ’iic_end_’). The terms IIC and I^{2}C are used interchangably, and both mean Inter–Integrated Circuit.

NOTE:
To incorporate this program into your main program, place it somewhere in your source file by including the following text:

```
;         $include(mod552)      ;include the desired processor descriptor file
;         $include(IIC_OS2.asm)                          ;include this program
```

Since this program has a ’CSEG AT... definition for the IIC interrupt vector, it is probably best to place it in your program where all the other interrupt vector directives reside so that assembly synchronisation errors do not occur.

One must also ensure that the data bytes used by this program do not conflict with those in your main program. Don’t forget to initialize the I^{2}C control registers and the interrupt registers, etc. For example:

```
;INIT:          .
;               .
;               .
;        MOV     IEN0,#10100000B         ;enable IIC interrupt (and any other)
;        MOV     S1CON,#ENS1_NOTSTA_NOTSTO_NOTSI_AA_CR0
;        MOV     S1ADR,#Own_adrs OR general_enable_ ;enable slave/general mode
;        MOV     IIC_status,#status_OK_              ;init system status byte
;        CLR     IIC_failure                                 ;init status bit
;               .
;               .
;               .
;
```

This driver uses DATA space bytes (approx. 16 bytes), plus several buffers which are required only for multi–master scenarios where this micro can be addressed as a slave. One bit addressable byte is used. The user must ensure the following EQUates are set appropriately for his system:

Slave address for this microcontroller – other bus masters can address this micro as a slave; this driver simply sends ‘SLVbytes_out_’ number of bytes or receives ‘SLVbytes_in_’ number of bytes in the case of being addressed as a slave. The LSBit of the address (‘Own_adrs_’) is set indicating that general calls will be responded to.

```
;
Own_adrs_         EQU     02EH    ;address of micro when addressed as a slave
general_enable_   EQU     1       ;general call recognised since LSBit is set
SLVbytes_in_      EQU     8     ;# bytes to receive when addressed as a slave
SLVbytes_out_     EQU     8         ;# bytes to transmit when    ”     ”    ”
IIC_buffer_size_  EQU     8           ;# bytes reserved for ’IIC_data_buffer’
;
```

Change the following equates to suit your system – they define where the start of DATA and BIT ADDRESSABLE DATA for the required bytes in ”IIC_OS2”.

```
;
IIC_OS2_DATA             EQU     48       ;change location to suit your system
IIC_OS2_BITADRS_DATA     EQU     32                           ;bit addressable!
;
```

To interface to this I^{2}C driver, the user need not understand all the details of the program – only the following registers must be understood:

```
;       ’IIC_data_buffer’                – used with the ’ioBuffer_’ command
;       ’Slave_in’ buffer (if required)  – used only in multi–master systems
;       ’Slave_out’ buffer (if required) – used only in multi–master systems
;       ’aux_adrs’                       – used with ’use_aux_adrs_’ command
;       ’indirect_adrs’                  – used with the ’indirect_’ command
;       ’indirect_count’                 – used with the ’indirect_’ command
;       ’IIC_failure’ (BIT)              – set if command file was kaput
;       ’IIC_status’                     – holds dynamic status of session
;       ’IIC_final_status’               – holds the final status of session
;
```

Additionally, there is a command file structure (the command file is a list of commands that ”IIC_OS2” will execute) which the user must conform to. The list of command file directives includes:

```
;       ’ioD_’          – target DATA space for data transfers
;       ’ioC_’          – target CODE space for data transfers
;       ’ioX_’          – target XDATA space for data transfers
;       ’ioBuffer_’     – target ’IIC_data_buffer’ for data transfers
;       ’immediate_’    – used to output bytes from command file stream
;       ’call_’         – used to call a subroutine between reapeated starts
;       ’indirect_’     – gets I/O address and count from ’indirect_ registers
;       ’use_aux_adrs_’ – gets slave address from ’aux_adrs’
;
;       ’iic_end_’      – last byte of a command file
;     ’iic_write_mask_’ – OR with slave address to indicate a write operation
;     ’iic_read_mask_’  – OR with slave address to indicate a read operation
;
```

The command file structure is explained in detail below.

NOTE:
Multi–master systems are very specific to the system design, and therefore, very difficult to make generic. Every multi–master system will have a different protocol for how many (and which) bytes to send/receive when the master is addressed as a Slave Receiver or Slave Transmitter. For this reason, this program implements the multi–master scenario very simply – if the micro running this program is addressed as a slave, it will read up to ’SLVbytes_in_’ number of data bytes or write ’SLVbytes_out_’ number of data bytes (depending on what the calling master requests). The target data buffer in these cases are the ’Slave_in’ buffer and the ’Slave_out’ buffer.

This program also treats the general call scenario as a Slave Receiver mode. If the general call is received, and the ‘S1ADRS’ has been set to accept the general call, up to eight bytes can be received into the ‘Slave_in’ buffer. The IIC specification has defined how the general call should be handled, and the user can write his own code to conform to this specification, or simply use the general call as a means of sending common information to all the masters on the bus which use this driver. Since the bytes sent from the general call master to the other masters which use this driver end up in the ‘Slave_in’ buffer (and the ‘IIC_status’ is set to indicate a general call scenario), the user can write code in his mainline routine to conform to the I^{2}C specification if he wishes.

The user can make the size of these slave input/output buffers (by altering the corresponding equates ‘SLVbytes_in_’ and ‘SLVbytes_out_’) as large as required. The calling Master can terminate the slave session at any number of data bytes sent or received by providing a stop or a not acknowledge.

IIC_OS2, when integrated into the user’s system, will require 16 DATA bytes (mapped anywhere in the internal DATA memory space), and one bit–addressable byte. About 600 bytes of code–space memory are used.

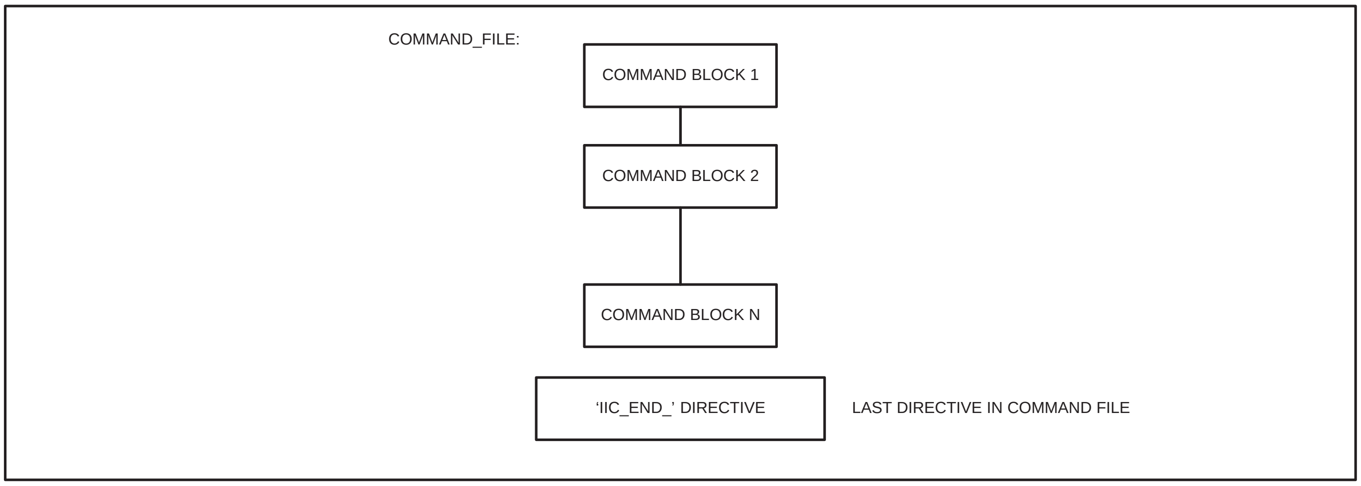
The user of this program need not concern himself with the bit or byte level operation of the I^{2}C harware – this program takes care of all I^{2}C registers, and checks for all collisions, arbitration lost scenarios, bus errors, etc. A command list consistiing of a limited number of simple macro commands is set–up by the user, and this driver uses that list of commands to perform the desired I^{2}C operations.

The user loads the DPTR register with the address of the sequence of I^{2}C operations desired. Once this register is loaded, the “DO_IIC” routine is called. “DO_IIC” starts the I^{2}C process by forcing a START condition from which point the interrupt service routine “IIC_VECTOR” will execute the command file and interact with the IIC SFRs. “DO_IIC” acts as a timeout watchdog while the command file is running. Once the command file is completely executed, “DO_IIC” will return to the calling routine with the ‘IIC_failure’ bit and ‘IIC_final_status’ byte set according to the results of the command file execution. The ‘IIC_failure’ bit will be set if an error occurs so the calling program could try again or interrogate the ‘IIC_final_status’ byte to determine exactly what kind of error took place.

The I^{2}C operations to be performmed are stored sequentially starting at the address specified by the DPTR (‘IIC_Command_File_adrs’) and in the memory space designated by ’Data?adrs?space’. I^{2}C operations include:

1. sending or receiving any number of bytes from 1 to 255 into any valid address space
2. repeated start automatically performmed so multiple slaves can be communicated with in one call
3. call subroutines between repeated start conditions directly from the I^{2}C command file list (i.e., transparent to the calling routine).

The I^2C Command File must be constructed so that it conforms to the I^2C driver system format. This format is very simple and can be pictorially viewed as:



Each Command Block consists of the following bytes: (note '+' means logical OR)

1. slave address + direction mask	<u>slave address</u> = hex number or 'use_aux_adrs_' directive <u>direction mask</u> = 'iic_read_mask_' or 'iic_write_mask_'
2. memory space directive + directive options + sub-directive options	<u>memory space directive</u> = none or 'ioD_' or 'ioX_' or 'ioC_' or 'immediate_; (targets CODE) or 'ioBuffer_; (targets DATA) <u>directive options</u> = none or 'indirect_' ('indirect_' used only with 'ioD_', 'ioX', or 'ioC' memory space directives) <u>sub-directive option</u> = none or 'call_'
3. number of data bytes	for Options 1 to 3 only
4. low byte address of data	for Options 1 to 3 only
5. high byte address of data	for Options 1 to 3 only (and only if target is CODE or XDATA).
6. low byte of address of subroutine	only if 'call_' sub-directive used
7. high byte of address of subroutine	only if 'call_' sub-directive used

The I^{2}C Command File is built from 1 to any number of blocks. Each block is from 2 to 7 bytes long, depending on what functions must be performmed. There are only four types of blocks, indicated by the options in the format below and breifly explained here:

Option 1 = GENERAL DATA TRANSFER into DATA or XDATA space, (or from DATA, XDATA or CODE space)

FUNCTION:	send/receive bytes to/from slave from/to any memory space
# BYTES IN THIS COMMAND FILE BLOCK:	5
NUMBER OF BYTES TO SEND/RECEIVE:	get number from command file
ADDRESS FOR DATA DERIVED FROM:	command file
OTHER FUNCTIONS:	none
COMMENTS:	Option 1 is useful for sending or receiving any specified number of data bytes to/from the specified slave. Every required piece of information is stored in the command file – that is, the address of the slave + read/write bit, the number of bytes to send or receive, and the address to send from or receive to. Recall that the address space is always specified in the command file.

Option 2 = 'immediate_' DIRECTIVE

FUNCTION:	send bytes to slave from command file
# BYTES IN THIS COMMAND FILE BLOCK:	4
NUMBER OF BYTES TO SEND/RECEIVE:	get number from command file
ADDRESS FOR DATA DERIVED FROM:	data read directly from command file
OTHER FUNCTIONS:	none
COMMENTS:	Option 2 is used to send up to 255 bytes to an addressed slave. The bytes are fixed and are stored in the command file itself; this method reduces command file bytes since no specification for data address is necessary. Option 2 provides a simple means of setting the sub–address in I^{2}C memory devices.

Option 3 = 'ioBuffer_' DIRECTIVE

FUNCTION:	send/receive bytes to/from slave from/to 'IIC_data_buffer'
# BYTES IN THIS COMMAND FILE BLOCK:	3
NUMBER OF BYTES TO SEND/RECEIVE:	get number from command file
ADDRESS FOR DATA DERIVED FROM:	always targets the 'IIC_data_buffer'
OTHER FUNCTIONS:	none
COMMENTS:	Option 3 is the data equivalent of 'immediate_'. That is this option tells IIC_OS2 to always target the DATA space 'IIC_data_buffer' for input and output. The user must pre–load the data into the buffer for outputs. The user must ensure that the number of data bytes specified for input must not exceed 'IIC_buffer_size_'.

Option 4 = 'indirect_' DIRECTIVE

FUNCTION:	send/receive bytes to/from slave from/to any memory space
# BYTES IN THIS COMMAND FILE BLOCK:	2
NUMBER OF BYTES TO SEND/RECEIVE:	get number from 'indirect_count'
ADDRESS FOR DATA DERIVED FROM:	get address from 'indirect_address'
OTHER FUNCTIONS:	none
COMMENTS:	Option 4 assumes that the calling program has set–up the 'indirect_count' register with the number of bytes to be sent or received, and the 'indirect_address' with the address of the bytes to be sent or received. The data space targetted is specified in the command file as usual.

Additionally, the ‘call_’ and ‘use_aux_adrs_’ subdirectives allow for additional flexibility:

SUBDIRECTIVE ‘call_’

The ‘call_’ subdirective can be ORed with any of the memory space directives (the second byte of all command file blocks). If this is done, a subroutine whose address is specified at the end of the command file block will be called once the block is completed execution.

ALTERNATE ADDRESS OPTION ‘use_aux_adrs_’

The ‘use_aux_adrs_’ command is used in place of the slave address in the command file (the first byte of the command file block). It tells the system to get the slave address from the ‘aux_adrs’ register. This function is useful when many slave devices are communicated with in a very similar fashion – instead of having a separate block for each slave, each block being identical except for the slave address, the user could have one block and use the ‘use_aux_adrs_’ feature.

To efficiently use this system, several of these blocks can be put together. In fact, there is no limit on the number of blocks allowed. This IIC_OS2 lends itself very nicely to complex I^{2}C requirements. The following examples will illustrate the usefullness of this program.

EXAMPLES

The following examples are samples for each option. The code fragment “PROGRAM” is simply the part of the code that sets up and calls the “DO_IIC” program which waits for the execution of the command file. The “Optionx_file” code fragments are the code space command files.

It should be noted that “IIC_OS2” will process a command file until an ‘iic_end_’ character is encountered – after which, a STOP condition will be implemented. This means that the master can keep possesion of the bus for as long as it has to.

It is assumed that all the slave addresses and other EQUates have been defined in the program previously.

EXAMPLE Option 1 – General Data Transfer

```
;PROGRAM:
;        MOV     DPTR,#Option1_file              ;load address of command file
;        CALL    DO_IIC                ;call program to wait for IIC execution
;        JBC     IIC_failure,PROGRAM            ;check for any type of failure
;        JMP     MORE_PROGRAM
;        ;
;        ;Notice the block structure of the command file.  Each block has
;        ;been spaced to accentuate this structure.
;        ;’Option1_file’ tells the IIC_OS2 (called through ”DO_IIC”) to
;        ;read 5 bytes of data in from ’slave1’ and store the bytes in the
;        ;DATA space starting at location ’d_iic_data’; after this input
;        ;is done, ’slave2’ has 1 byte written to it from the XDATA space
;        ;starting at address ’x_iic_data’.  Notice that all XDATA memory
;        ;space references require 2 address bytes.
;        ;Both blocks below are option 1 types, but the first is a read
;        ;and the second is a write.
;        ;
; Option1_file:
;        DB      slave1_address OR iic_read_mask_      ;slave1 address + read bit
;        DB      ioD_                              ;indicate DATA space target
;        DB      5                                   ;indicate number of bytes
;        DB      d_iic_data                     ;start address of target bytes
;
;        DB      slave2_address OR iic_write_mask_    ;slave2 address + write bit
;        DB      ioX_                              ;indicate DATA space target
;        DB      1                                   ;indicate number of bytes
;        DB      LOW(x_iic_data)                ;start address of target bytes
;        DB      HIGH(x_iic_data)
;
;        DB      iic_end_                                   ;end of iic session
;
;MORE_PROGRAM:
;        continue with program
;
```

EXAMPLE Option 2 – ‘immediate_’ Directive

```
;PROGRAM:
;        MOV     DPTR,#Option2_file              ;load address of command file
;        CALL    DO_IIC                ;call program to wait for IIC execution
;        JBC     IIC_failure,PROGRAM            ;check for any type of failure
;        JMP     MORE_PROGRAM
;        ;
;        ;Notice the block structure of the command file.  Each block has
;        ;been spaced to accentuate this structure.
;        ;’Option2_file’ tells the IIC_OS2 (called through ”DO_IIC”) to
;        ;write one byte of data to the addressed slave – the data is present
;        ;in the command file.
;        ;In this example, the address for a memory location in an IIC memory
;        ;peripheral will be set and the following block does an option 1
;        ;type of input.
;        ;
;Option2_file:
;        DB      PCF8570_adrs OR iic_write_mask_    ;slave address + write bit
;        DB      immediate_                           ;send out next byte only
;        DB      1                                       ;number of data bytes
;        DB      1                                       ;data byte to be sent
;                                                    ;(end of option 2 block)
;        DB      PCF8570_adrs OR iic_read_mask_      ;slave address + read bit
;        DB      ioD_                          ;memory space where bytes go to
;        DB      6                                   ;number of bytes to be read
;        DB      iic_input_data                         ;address of input target
;
;        DB      iic_end_
;
;MORE_PROGRAM:
;        continue with program
;
```

EXAMPLE Option 3 – ‘ioBuffer_’ Directive

```
;PROGRAM:
;        MOV     DPTR,#Option3_file              ;load address of command file
;        CALL    DO_IIC                ;call program to wait for IIC execution
;        JBC     IIC_failure,PROGRAM            ;check for any type of failure
;        JMP     MORE_PROGRAM
;        ;
;        ;Notice the block structure of the command file.  Each block has
;        ;been spaced to accentuate this structure.
;        ;’Option3_file’ tells the IIC_OS2 (called through ”DO_IIC”) to
;        ;read 5 bytes of data in from ’slave1’ and store the bytes in the
;        ;DATA space starting at location ’IIC_data_buffer’; after this input
;        ;is done, ’slave2’ has 3 bytes written to it from the DATA space
;        ;starting at address ’IIC_data_buffer’.  The ’IIC_data_buffer’ is
;        ;targetted because the ’ioBuffer_’ directive was specified.
;        ;Both blocks below are option 1 types, but the first is a read
;        ;and the second is a write.
;        ;
; Option1_file:
;        DB      slave1_address OR iic_read_mask_       ;slave1 address + read bit
;        DB      ioBuffer_                        ;indicate ’ioBuffer’ target
;        DB      5                                  ;indicate number of bytes
;
;        DB      slave2_address OR iic_write_mask_     ;slave2 address + write bit
;        DB      ioBuffer_                        ;indicate ’ioBuffer’ target
;        DB      3                                  ;indicate number of bytes
;
;        DB      iic_end_                                  ;end of iic session
;
```

EXAMPLE Option 4 – ‘indirect_’ Directive

```
;PROGRAM:
;        MOV     indirect_adrs,#LOW(input_data1)
;        MOV     indirect_adrs+1,#HIGH(input_data1)
;        MOV     indirect_count,#6
;        MOV     A,decision
;        JZ      PROGRAM_10
;        MOV     indirect_adrs,#LOW(input_data2)
;        MOV     indirect_adrs+1,#HIGH(input_data2)
;        MOV     indirect_count,#3
;
;PROGRAM_10:
;        MOV     DPTR,#Option4_file              ;load address of command file
;        CALL    DO_IIC                ;call program to wait for IIC execution
;        JBC     IIC_failure,PROGRAM_10         ;check for any type of failure
;        JMP     MORE_PROGRAM
;        ;
;        ;
```

Notice the block structure of the command file. Each block has been spaced to accentuate this structure. ‘Option4_file’ tells the IIC_OS2 (called through “DO_IIC”) to read ‘indirect_count’ number of bytes into external ram space starting at address ‘indirect_adrs’. In the body of “PROGRAM” the ‘indirect_ registers are loaded based on a decision. In this case, if the data byte ‘decision’ is zero, 6 bytes of data are read from the slave and placed in external ram starting at the address ‘input_data1’; if ‘decision’ is not zero, three bytes of data are read from the slave and placed in external ram starting at address ‘input_data2’.

```
Option4_file:
;        DB      slave_address OR iic_read_mask_     ;slave address + read bit
;        DB      ioX_ OR indirect_              ;read to external ram area and
;                                                      ;use indirect registers
;        DB      iic_end_
;
;MORE_PROGRAM:
;        continue with program
;
```

EXAMPLE using subdirective ‘call_’

The ‘call_’ subdirective can be used with any of the four options outlined above. The ‘call_’ subdirective is ORed with the memory space directive (the second byte of any command block).

```
;PROGRAM:
;        MOV    DPTR,#Option1_file_with_call
;        CALL   DO_IIC
;        JBC    IIC_failure,PROGRAM
;               .
;               .
;               .
;
;       ;
;       ;This example has 5 bytes read from ’slave1’ and placed into DATA
;       ;space ’d_iic_data’.  A subroutine is then called to sum the 5 bytes
;       ;and the result is placed into ’d_iic_data’.  The second block writes
;       ;the sum to ’slave2’.
;       ;
;Option1_file_with_call:
;        DB      slave1_address OR iic_read_mask_   ;slave1 address + read bit
;        DB      ioD_ OR call_             ;indicate DATA target + subroutine
;        DB      5                                  ;indicate number of bytes
;        DB      d_iic_data       ;address of data (DATA are 1 byte addresses)
;        DB      LOW(Option1_sub)                       ;address of subroutine
;        DB      HIGH(Option1_sub)
;
;        DB      slave2_address OR iic_write_mask_ ;slave2 address + write bit
;        DB      ioD_                                   ;indicate DATA target
;        DB      1                                  ;indicate number of bytes
;        DB      d_iic_data                                  ;address of data
;
;        DB      iic_end_                                    ;end of iic session
;
;       ;
;       ;Subroutine sums five bytes read in from 'slave1'.
;       ;
;Option1_sub:
;        MOV    R0,#d_iic_data
;        MOV    R1,#5
;        CLR    C
;        CLR    A
;Op1_10:
;        ADDC   A,@R0
;        INC    R0
;        DJNZ   R1,Op1_10
;        MOV    d_iic_data,A
;        RET
;
```

EXAMPLE using alternate address option 'use_aux_adrs_'

The alternative address option is used to tell "IIC_OS2" to get the slave address from the register 'aux_adrs'. This is useful for writing generic Command File Blocks to deal with a family of I^{2}C peripherals like I/O expandors, RAMs etc.

In the following example, one implementation of a generic command file to read and write to PCF8570 RAMs is shown. To use these generic command files, the user loads the address of the PCF8570 into 'aux_adrs' and the address of the byte in the RAM into 'IIC_data_buffer', then calls the command file "Read_RAM". To write to RAM, repeat the same as above, and load the data to be sent into 'IIC_data_buffer + 1'.

```
;PROGRAM:
;        MOV    aux_adrs,#PCF8570_1_adrs                 ;slave address of RAM
;        MOV    IIC_data_buffer,#12                ;address of target location
;        MOV    DPTR,#Read_RAM                  ;generic RAM read command file
;        CALL   DO_IIC
;        JBC    IIC_failure,PROGRAM
;
;        MOV    aux_adrs,#PCF8570_2_adrs                 ;slave address of RAM
;        MOV    IIC_data_buffer,#36                ;address of target location
;        MOV    IIC_data_buffer+1,#7                              ;data to send
;        MOV    DPTR,#Write_RAM
;        CALL   DO_IIC
;        JBC    IIC_failure,PROGRAM
;
;              .
;              .
;              .
;       ;
```

"Read_RAM" is a generic read PCF8570 command file. It expects the slave address of the PCF8570 to be loaded into 'aux_adrs'; the address of the byte to read should be in 'IIC_data_buffer'.

"Write_RAM" is a generic write PCF8570 command file. It expects the slave address of the PCF8570 to be loaded into 'aux_adrs'; the address of the byte to read should be in 'IIC_data_buffer'; the data to be sent should be loaded into 'IIC_data_buffer + 1'.

```
;       ;
;Read_RAM:
;        DB      use_aux_adrs_ OR iic_write_mask_   ;slave address + write bit
;        DB      ioBuffer_                  ;indicate 'IIC_data_buffer' target
;        DB      1                                   ;indicate number of bytes
;
;        DB      use_aux_adrs_ OR iic_read_mask_     ;slave_address + read bit
;        DB      ioBuffer_                  ;indicate 'IIC_data_buffer' target
;        DB      1                                            ;number of bytes
;
;        DB      iic_end_                                 ;end of iic session
```

```
;
;Write_RAM:
;        DB      use_aux_adrs_ OR iic_write_mask_   ;slave address + write bit
;        DB      ioBuffer_                 ;indicate 'IIC_data_buffer' target
;        DB      2                                  ;indicate number of bytes
;
;        DB      iic_end_                                 ;end of iic session
;
```

SYSTEM REQUIREMENTS

Data Bytes Used:	16
Bit Addressable Bytes Used:	1
Stack Penetration:	Approximately 17 bytes worst case
Code Length:	Approximately 600 bytes of code.

"IIC_OS2" register definitions

The following data bytes are required to run the full implementation of the I^{2}C driver system. In a microcontroller as large as the 80C552 or 80C652, the requirement of these data bytes will not impose a great toll on the user's system. Note that registers, bytes, equates and bit names are surrounded by ' ' in the description – routines, subroutines and procedure names are surrounded by " ".

'IIC_status'
LOADED BY: "DO_IIC" and "IIC_VECTOR"

DESCRIPTION: holds the status of the requested IIC operations. This data byte is loaded with 'status_DO_IIC' by "DO_IIC", which then monitors this byte, and determines if the I^{2}C command file has been completed. Completion of the I^{2}C command file is known if the 'IIC_status' is equal to one of the following in its lower nibble:

'status_OK_'	– operation complete, no problems
'status_arb_lost_'	– arbitration lost to another master
'status_attempt_data_'	– tried to send data 'max_data_attempts_' times
'status_attempt_adrs_'	– tried to find slave 'max_adrs_attempt_' times
'status_timeout_'	– waited 'max_wait_' time for activity
'status_buss_err_'	– a bus error (illegal start/stop)

The upper nibble of the 'IIC_status' will be one of the following:

'status_master_'	– started&ended as a master
'status_slave_'	– addressed as slave (own address)
'status_arb_lost_slave_r_'	– arbitration lost to another master, this one addressed as a slave to be read from
'status_arb_lost_slave_w_'	– arbitration lost to another master, this one addressed as a slave to be written to
'status_general_slave_'	– addressed as slave (general call)
'status_arb_lost_general_'	– arbitration lost to a general call

The upper nibble is useful since one could have called the I^{2}C routine intending to be a master transmitter/receiver, but could be changed to a slave in mid–stream. This information may be useful in some applications.

The values 'max_data_attempts_', 'max_adrs_attempts_' and 'max_wait_' are equated in the main body below – these numbers define how many attempts should be made to send/receive data, locate a slave or wait for a response.

'IIC_final_status'
This byte holds a copy of the 'IIC_status' at the end of the command file execution. The calling routine should interrogate this byte if an 'IIC_failure' is detected.

'IO_buffer_adrs'
LOADED BY: "IIC_OS2"

DESCRIPTION: is loaded by the "IIC_OS2" operation and holds the address of the data to be trasmitted to a slave, or received from a slave. The bit-addressable register 'Data?adrs?space' determines which memory space the 'IO_buffer_adrs' targets. The initial value for this register can come from the command file itself, or may be loaded from the 'indirect_adrs' register, depending on the actions directed from the 'Data?adrs?space' register.

'IIC_Command_File_adrs'
LOADED BY: "DO_IIC" from the DPTR of the calling routine manipulated by "IIC_OS2"

DESCRIPTION: the calling routine loads the address of the command file to be executed by the "IIC_OS2" into the DPTR. "DO_IIC" loads the DPTR into this register. The "IIC_OS2" will modify this address register as the "IIC_OS2" operations proceed. If the command file resides in the DATA space, then only the LSByte of the address is used.

‘indirect_adrs’
‘indirect_count’
LOADED BY: calling routine or a command file called subroutine

DESCRIPTION: the calling routine may use the ‘indirect_’ option as loaded in the ‘Data?adrs?space’ byte. This option directs the “IIC_OS2” to get the ‘IO_buffer_adrs’ and the ‘Multiple_count’ from the calling routine (or called subroutine) loaded ‘indirect_adrs’ and ‘indirect_count’ registers.

‘Multiple_count’
LOADED BY: “IIC_OS2”

DESCRIPTION: this register is a counter for the number of bytes to be received or transmitted. Received or transmitted bytes are sent to or read from the address space indicated in ‘Data?adrs?space’ and addressed by the ‘IO_buffer_adrs’. The initial value for this register can come from the command file itself, or may be loaded from the ‘indirect_count’ register, depending on the actions directed from the ‘Data?adrs?space’ register.

‘Attempt_count’
LOADED BY: “IIC_OS2”

DESCRIPTION: counts the number of failed attempts at sending/receiving data or addressing a slave. If the number of tries in either case excedes ‘max_adrs_attempts_’ or ‘max_data_attempts_’, the error status is reflected in ‘IIC_status’ and the “IIC_OS2” quits.

’last_data’
LOADED BY: “IIC_OS2”

DESCRIPTION: holds the value of the last data byte received or transmitted. Used in “IIC_OS2” as a look–back register so failed transmissions can be repeated.

‘iic_timer’
LOADED BY: “IIC_OS2”

DESCRIPTION: used as a watchdog timer for the IIC operation. Implemented as a up–counter in “DO_IIC”, but ideally, this function should be in the hands of a real system timer.

‘Slave_in & Slave_out’
LOADED BY: mainline routine

DESCRIPTION: Buffers for Slave receiver and Transmitter modes. Main routine loads ‘Slave_out’ with the ‘SLVbytes_out_’ number of bytes to be transmitted once addressed by another master. ‘Slave_in’ is filled by the ‘SLVbytes_in_’ number of bytes from another master. If more or less bytes are to be received or sent when addressed as a slave, then the size of the buffers and the ‘SLVbytes_... EQUates must change.

IIC_OS2 does not need these 16 DATA bytes if the system is single-master. IF your system is single-master, then use the ‘Slave_in’ and ‘Slave_out’ buffers for your own general purpose buffers or registers.

IIC_data_buffer
This buffer is a general purpose buffer for the “IIC_OS2” system. It is the source or destination target when the address space directive in byte 2 of the command file is ‘ioBuffer_’.

‘aux_adrs’
Loaded by the calling routine with the address of the slave to be communicated with. This function allows the user to have a generic command block with the slave address specified by ‘aux_adrs’. The command file has ‘use_aux_adrs_’ in place of the slave address. The routine that loads ‘aux_adrs’ must ensure the read/write bit is set correctly (by ORing the ‘aux_adrs’ with ‘iic_write_mask_’ or ‘iic_read_mask_’).

```
DSEG    AT      IIC_OS2_DATA

IIC_status:             DS      1
IIC_final_status:       DS      1
IO_buffer_adrs:         DS      2
IIC_Command_File_adrs:  DS      2
indirect_adrs:          DS      2
indirect_count:         DS      1
iic_timer:              DS      2
Attempt_count:          DS      1
Multiple_count:         DS      1
last_data:              DS      1
aux_adrs:               DS      1

Slave_in:               DS      SLVbytes_in_
Slave_out:              DS      SLVbytes_out_
IIC_data_buffer:        DS      IIC_buffer_size_
```

Use ‘DATA_start’ to define where your system DATA bytes will be located.

```
DATA_start      EQU     IIC_data_buffer + IIC_buffer_size_
```

‘Data?adrs?space’ is loaded by the calling routine, and maipulated by the “IIC_OS2” routine. It indicates which memory space the command file is to be read from. It also ultimately gets the address space for the input and output data, as well as the indication for the special functions ‘indirect_’, ‘immediate_’, and ‘call_’. Generally speaking, the calling routine loads ‘Data?adrs?space’ with the code for which memory space holds the command file to be executed – then, the command file information ammends this byte as each block of the command file is executed.

```
DSEG    AT      IIC_OS2_BITADRS_DATA

Data?adrs?space:     DS      1

BSEG    AT      Data?adrs?space.0

IO_Data:                DBIT    1
IO_Code:                DBIT    1
IO_Buffer:              DBIT    1
immediate_data:         DBIT    1
call_function:          DBIT    1
indirect_xxx:           DBIT    1
```

This next bit is set whenever the microcontroller is addressed as a slave receiver or a slave transmitter. “DO_IIC” needs this bit to hold off pending calls from the main routines to the I^2C bus.

```
i_am_a_slave:           DBIT    1
```

If any I^2C session failure occurs, set the failure bit

```
IIC_failure:            DBIT    1
```

Mask bytes for the various ’Data?adrs?space’ bits. These codes are used in the command file. See examples above.

```
;
ioD_            EQU     00000001B           ;input/output data from DATA space
ioC_            EQU     00000010B                 ;output data from CODE space
ioBuffer_       EQU     00000100B    ;input/output data from ’IIC_data_buffer’
ioX_            EQU     00000000B          ;input/output data from XDATA space
immediate_      EQU     00001000B                ;next byte is data to be sent
call_           EQU     00010000B                ;call subroutine after in/out
indirect_       EQU     00100000B          ;get info from ’indirect_ registers
iospaces_       EQU     00000111B              ;mask to isolate I/O space bits
commands_       EQU     11111000B     ;mask to isolate control and status bits
slave_bit_mask  EQU     01000000B          ;mask to isolate ’i_am_a_slave’ bit
failure_bit_mask EQU    10000000B           ;mask to isolate ’IIC_failure’ bit
;
```

The following status bytes are used to indicate the present state as well as the ultimate state of the I^{2}C operations. These values are loaded into ‘IIC_status’ by the various states as well as “DO_IIC”.

```
;
status_OK_               EQU     0         ;end of session and/or bus available
status_arb_lost_         EQU     1                           ;arbitration lost
status_attempt_data_     EQU     2;’max_data_attempts_’ failed to get/send data
status_attempt_adrs_     EQU     3   ;’max_adrs_attempts_’ failed to find slave
status_timeout_          EQU     4         ;”DO_IIC” detected timeout problem
status_buss_err_         EQU     5           ;a bus error or illegal start/stop
status_bad_read_space_   EQU     6       ;command file target for read was CODE


status_master_              EQU     00H            ;master receiver/transmitter
status_slave_               EQU     10H       ;addressed as slave (own address)
status_arb_lost_slave_r_    EQU     20H   ;arbitration lost, addressed as slave
status_arb_lost_slave_w_    EQU     30H   ;arbitration lost, addressed as slave
status_general_slave_       EQU     40H      ;addressed as slave (general call)
status_arb_lost_general_    EQU     50H         ;arbitration lost, general call


status_DO_IIC_           EQU     0FH    ;all running fine (bus busy) indication
status_type1_mask_       EQU     0FH              ;mask to isolate lower nibble
status_type2_mask_       EQU    0F0H    ;mask to isolate upper nibble of status
;
;
```

IIC control characters

These characters are used in the IIC command file and various routines.

‘iic_end_’
: must be the last character in any command file sequence.

‘iic_write_mask_’ and ‘iic_read_mask_’
: are used in conjunction with the slave address to indicate whether data is a comin’ or a goin’.

‘max_data_attempts_’
: should be equated to a value indicating how many times this system should re–try to get data from/to a slave before it gives up and says an error has occured.

‘max_adrs_attempts_’
: should be equated to a value indicating how many times this system should re–try to address a slave before it gives up and says an error has occured.

‘max_wait_’
: is used in a crude loop counting timer in “DO_IIC”. This number should be equated to give roughly the timeout time required by your system (i.e., I^{2}C inactivity timeout). The count will depend on the clock speed as well as the average loop time in “DO_IIC”. The loop time will be affected by the IIC interrupt processing as well as any other interrupt service routines in your system.

‘use_aux_adrs_’
: is used in place of the slave address (the first byte in the command file block) so that the system will get the slave address from the ‘aux_adrs’ register. The routine that loads ‘aux_adrs’ must ensure the read/write bit is set correctly (by ORing the ‘aux_adrs’ with ‘iic_write_mask_’ or ‘iic_read_mask_’). This option excludes the slave address 0FEH from being used directly, but if it is required, the user must use the ‘use_aux_adrs’ option and load 0FEH into ’aux_adrs’.

```
;
iic_end_                 EQU             0FFH          ;end of IIC command file
use_aux_adrs_            EQU             0FEH  ;use ’aux_adrs’ as slave address
iic_write_mask_          EQU             00H
iic_read_mask_           EQU             01H
max_data_attempts_       EQU             3      ;maximum tries to get/send data
max_adrs_attempts_       EQU             3      ;maximum tries to address slave
max_wait_                EQU             1        ;reload value for ’iic_timer’
                                                                  ;(counts up)
```

'S1STA' relaod values.
Virtually the same as old '552 app. note.

```
;
ENS1_NOTSTA_STO_NOTSI_NOTAA_CR0              EQU        0D1H
ENS1_NOTSTA_STO_NOTSI_AA_CR0                 EQU        0D5H
ENS1_NOTSTA_NOTSTO_NOTSI_AA_CR0              EQU        0C5H
ENS1_NOTSTA_NOTSTO_NOTSI_NOTAA_CR0           EQU        0C1H
ENS1_STA_NOTSTO_NOTSI_AA_CR0                 EQU        0E5H
```

I^{2}C hardware interrupt vector definition

```
CSEG    AT      002BH
        JMP     IIC_VECTOR                 ;IIC_OS2 interrupt service routine
;
CSEG
;
```

"DO_IIC"

The calling routine has loaded the DPTR register with the address of the command file to be executed. This routine simply waits for the I^{2}C process to be completed. Completion of the I^{2}C session is indicated by 'IIC_status'= 'status_OK_', or one of the many error codes. "IIC_VECTOR" will take care of updating the status byte.

It is possible that another master has addressed this master as a slave. If "DO_IIC" is called under these circumstances, it will exit with 'IIC_failure' set and also check for a timeout for the addressed slave session. If a timeout is detected, all IIC_OS2 registers and bits are set to their default value. In either case the 'IIC_failure' bit is set.

In the addressed slave mode, the calling master determines bus timing and clock values etc. If something happens to that master, and it cannot complete its session, then this slave is left hanging! For this reason, we have a built-in timeout check feature for addressed slave mode (if and when this slave calls "DO_IIC" to do its own work).

'IIC_final_status' is a copy of the 'IIC_status' register. It is used because the 'IIC_status' can be set as 'status_arb_lost_slave_' which indicates an arbitration lost, addressed as slave condition – when this occurs, the very next interrupt into "IIC_VECTOR" may change 'IIC_status', thus the calling routine will cannot determine what happened (thus it can check the 'IIC_final_status').

INPUT:

'IIC_status' is checked to ensure an addressed slave state is not in progress DPTR loaded with address of the command file to execute.

OUTPUT:

'IIC_status' is updated to reflect completion of I^{2}C session or error.

'IIC_final_status' holds a copy of 'IIC_status'.

'IIC_failure' is updated (0 = all OK, 1 = some kind of error).

```
DO_IIC:
        JNB     i_am_a_slave,DO_10
```

If a slave receive or transmit session is in effect (as initiated by another master), this processor will only know that through the "IIC_VECTOR" routine – there is no timeout check etc. Because of this situation, "DO_IIC" will check for a slave session in progress and exit as a failure if all is OK with that session (so that the calling routine will keep trying to get hold of the bus). If the slave session has timed out or there is an error, clear everything and exit as an error as well.

```
DO_05:
        CALL    IIC_time                        ;addressed slave timeout check
        JNZ     DO_35                               ;if not, exit as a failure
DO_06:
        CLR     i_am_a_slave               ;if timeout, reset system and exit
        SJMP    DO_ERR                                           ;as a failure
        ;
        ;ready to try – this micro is not presently addressed as a slave, and
        ;all else seems to be OK.
        ;
DO_10:
        MOV     IIC_Command_File_adrs,DPL
        MOV     IIC_Command_File_adrs + 1,DPH
        MOV     IIC_status,#status_DO_IIC_                 ;indicate IIC busy
        SETB    STA               ;do an IIC interrupt (start start condition)
```

```
DO_15:
        MOV     iic_timer,#LOW(max_wait_)                 ;start timeout timer
        MOV     iic_timer + 1,#HIGH(max_wait_)
DO_20:
        CALL    IIC_time
        JZ      DO_ERR
        ;
```

As long as ‘iic_timer’ is OK, loop here and check the ‘IIC_status’. ‘IIC_status’ will remain as ‘status_DO_IIC_’ as long as the IIC session is still on. This byte will be loaded with ‘status_OK_’ if the session ends normally, or it will be loaded with some other status byte if an error or arbitration process occurs.

If this micro has been addressed as a slave, or has lost arbitration and become a slave, then ‘IIC_status’ indicates the situation, and this subroutine is terminated. The routine that calls this one must check the ‘IIC_status’ to determine if another master has won the bus so that it can wait for ‘IIC_status’ to become ‘status_OK_’, at which point, it could try again.

```
DO_25:
        MOV     A,IIC_status                      ;when = status_OK_, all done
        CJNE    A,#status_DO_IIC_,DO_30
        SJMP    DO_20
DO_30:
        CJNE    A,#status_OK_,DO_35
        CLR     IIC_failure
        CLR     i_am_a_slave
DO_X:
        MOV     A,IIC_status
        MOV     IIC_final_status,A
        RET
        ;
        ;if ’iic_timer’ overflows, have an IIC bus timeout error.
        ;
DO_ERR:
        CALL    MORE_00_SUB                               ;clear all registers
        ANL     IIC_status,#status_type2_mask_          ;preserve upper nibble
        ORL     IIC_status,#status_timeout_               ;indicate inactivity
        CALL    DO_ERRX
        ;
```

Do error recovery here – i.e., lost arbitration, bus error. Alternately, let the calling routine interrogate ‘IIC_final_status’ so that the main routines decide what to do for various errors. For development and debug purposes, this example routine ignores the errors.

```
DO_35:
        SETB    IIC_failure                     ;indicate a failed IIC session
        SJMP    DO_X
DO_ERRX:                                  ;ensure the interrupt bit is cleared
        RETI
        ;
```

“IIC_time”
Subroutine to increment I²C timeout timer. ACC returns 0 if timeout occurs. Should make this an actual timer in your system so that the amount of time for a timeout is deterministic. Note that the “IIC_VECTOR” routine resets this timer as well.

```
IIC_time:
        INC     iic_timer
        MOV     A,iic_timer
        JNZ     time_X
        INC     iic_timer + 1
        MOV     A,iic_timer + 1
time_X:
        RET
;
```

Subroutine “FETCH_COMMAND”

DESCRIPTION:
This subroutine fetches a byte from the address ‘IIC_Command_File_adrs’ in code memory space. If “FETCH_COMMAND_0” is called, then the address pointer ‘IIC_Command_File_adrs’ is not incremented at exit, otherwise the address pointer is incremented.

INPUT:
‘IIC_Command_File_adrs’ holds the address of the byte to be retreived.

OUTPUT:
ACCumulator holds the retreived byte. ‘IIC_Command_File_adrs’ is incremented (not if “FETCH_COMMAND_0” is called).

```
FETCH_COMMAND_0:
        CLR     C                   ;carry indicates whether pointer inc or not
        SJMP    FC_10

FETCH_COMMAND:
        SETB    C

FC_10:
        MOV     DPL,IIC_Command_File_adrs           ;no, must be XDATA or CODE
        MOV     DPH,IIC_Command_File_adrs+1
FC_Code:
        CLR     A
        MOVC    A,@A+DPTR
FC_exit:
        JNC     FCX_10                     ;don’t increment pointer if no carry
        INC     DPTR                           ;if carry set, increment pointer
FCX_10:
        MOV     IIC_Command_File_adrs,DPL                        ;restore pointer
        MOV     IIC_Command_File_adrs+1,DPH
        RET
;
```

Subroutine “FETCH_DATA”

DESCRIPTION:
This subroutine is used by all the IIC_OS2 states to get the next data byte from the address ‘IO_address’ in the memory space indicated in ‘Data?adrs?space’. This routine also saves the fetched data in the byte ‘last_data’ so error recovery can be easiy done. Before this routine exits, the pointer ‘IO_buffer_adrs’ is incremented. For the ‘immediate_’ directive, simply get the next data byte from the command file stream. Fetched data returned in ACCumulator.

INPUT:
‘IO_address’ has address where data is to be fetched from. If the target space is DATA, then the LSByte of ’IO_address’ is the full address and the MSByte is ignored. ‘Data?adrs?space’ holds the information for which address space is to be targetted for fetching the data.

OUTPUT:
ACCumulator is loaded with the fetched byte. ‘last_data’ gets the fetched byte as well. ‘IO_address’ is incremented (unless in immediate option in which case the ‘IO_address’ is meaningless).

```
FETCH_DATA:
        JNB     immediate_data,FD_10
        CALL    FETCH_COMMAND              ;if immediate option, get data from
        JMP     FD_x2                                     ;command file stream
FD_10:
        JB      IO_Data,FD_Data                         ;is data in DATA space?
        MOV     DPL,IO_buffer_adrs       ;no, then must be XDATA or CODE space
        MOV     DPH,IO_buffer_adrs+1
        JB      IO_Code,FD_Code                           ;is it in CODE space?

FD_Xdata:
        MOVX    A,@DPTR                        ;no, it must be in XDATA space

FD_exit:
        INC     DPTR                                             ;bump pointer
        MOV     IO_buffer_adrs,DPL                            ;restore pointer
        MOV     IO_buffer_adrs + 1,DPH
FD_x2:
        MOV     last_data,A                                     ;store data
        RET
        ;
        ;enter here if data is in CODE space
        ;
FD_Code:
        CLR     A
        MOVC    A,@A+DPTR
        SJMP    FD_exit
        ;
        ;enter here if data is in DATA space
        ;
FD_Data:
        MOV     R0,IO_buffer_adrs
        MOV     A,@R0
        MOV     last_data,A
        INC     R0
        MOV     IO_buffer_adrs,R0
        RET
;
```

Subroutine “STORE_DATA”

DESCRIPTION:
This subroutine stores incoming data in the address ‘IO_address’ in the data space indicated in ‘Data?adrs?space’. Only XDATA and DATA spaces are valid since we cannot write into the CODE space.

INPUT:
ACCumulator has data to be stored. This data is not corrupted. ‘IO_address’ holds address for where data is to be stored ‘Data?adrs?space’ (bit addressable) describes which data space is to be targetted.

OUTPUT:
ACCumulator contents not corrupted by subroutine. ACCumulator contents are stored in address as described above. ‘last_data’ holds a copy of the data. ‘IO_address’ is incremented.

```
;
STORE_DATA:
        JB      IO_Data,SD_Data                          ;is target area DATA?

SD_Xdata:
        MOV     DPL,IO_buffer_adrs             ;no, then must be XDATA so load
        MOV     DPH,IO_buffer_adrs+1                      ;address into DPTR
        MOVX    @DPTR,A                                      ;store the data
        MOV     last_data,A                                     ;and copy it
        INC     DPTR                                ;bump the address pointer
        MOV     IO_buffer_adrs,DPL                           ;and restore it
        MOV     IO_buffer_adrs+1,DPH
        RET
        ;
        ;enter here if the target space is DATA
        ;
SD_Data:
        MOV     R0,IO_buffer_adrs                    ;get the address into R0
        MOV     @R0,A                                        ;store the data
        MOV     last_data,A                                     ;and copy it
        INC     R0                                  ;bump the address pointer
        MOV     IO_buffer_adrs,R0                            ;and restore it
        RET
;
```

Subroutine “make_space”

DESCRIPTION:
“make_space” is used to update the ‘Data?adrs?space’ byte which holds the information for which memory space is to be targetted for data and command bytes. This subroutine is usually called by a state that has just read–in the command file byte indicating address space information.

INPUT:
ACCumulator has the data address space indication read–in from command file.

OUTPUT:
‘Data?adrs?space’ is updated with ACCumulator contents.

```
make_space:
        XCH     A,Data?adrs?space              ;get present address space byte
        ANL     A,#commands_    ;clear all bits except command and status bits
        ORL     A,Data?adrs?space              ;mask in new address space info
        XCH     A,Data?adrs?space                                      ;restore
ms_X:
        RET
$eject
;
```

IIC interrupt vector

Everytime a significant event occurs on the I^{2}C bus (a start, stop, error, etc.), this interrupt routine is entered. This routine reads the I^{2}C hardware SFR called ‘S1STA’ to determine what state the I^{2}C hardware is in.

Each state has its own processing routine as shown below. The multi–master rotuines shown are very simple in this module – multi–master functions are very dependent on the system being serviced. This module simply relinquishes control of the bus if another master wins arbitration; it will receive or send bytes if it is addressed as a slave.

```
;============================================================================
;
IIC_VECTOR:
        PUSH    PSW               ;save all registers used in interrupt vector
        PUSH    ACC
        PUSH    DPL
        PUSH    DPH
        PUSH    AR0
;
```

‘S1STA’, the SFR indicating I^{2}C hardware status for the ’552 takes on a limited range of values, namely 00H to 0C8H in steps of 08H. The following manipulation changes the ‘S1STA’ value to a number from 0 to 25. This number is then multiplied by 2 so a jump can be done from an ‘AJMP’ table.

```
;
        MOV     A,S1STA            ;get SFR which holds hardware status of bus
        SWAP    A
        RLC     A
        JNC     IICV_10
        INC     A
IICV_10:
        RL      A
        MOV     DPTR,#S1STA_00
        JMP     @A+DPTR
;
;all sections exit here.
;The timeout timer ’iic_timer’ is restarted everytime around, it is assumed
;that if an interrupt occurs, that more than likely, everything is OK.
;
IIC_EXIT:
        MOV     iic_timer,#LOW(max_wait_)                ;reload timeout timer
        MOV     iic_timer + 1,#HIGH(max_wait_)

        POP     AR0
        POP     DPH
        POP     DPL
        POP     ACC
        POP     PSW
        RETI
;
;---------------------------------------------------------------------------
;Jump table for interrupt routine entry above.
;---------------------------------------------------------------------------
;
S1STA_00:
        AJMP     MORE_00                                       ;Bus Error mode
        AJMP     MORE_08                     ;Master Receiver/Transmitter Mode
        AJMP     MORE_10                     ;Master Receiver/Transmitter Mode
        AJMP     MORE_18                              ;Master Transmitter Mode
        AJMP     MORE_20                              ;Master Transmitter Mode
        AJMP     MORE_28                              ;Master Transmitter Mode
        AJMP     MORE_30                              ;Master Transmitter Mode
        AJMP     MORE_38                     ;Master Receiver/Transmitter Mode

        AJMP     MORE_40                                 ;Master Receiver Mode
        AJMP     MORE_48                                 ;Master Receiver Mode
        AJMP     MORE_50                                 ;Master Receiver Mode
        AJMP     MORE_58                                 ;Master Receiver Mode

        AJMP     MORE_60                                  ;Slave Receiver Mode
        AJMP     MORE_68                                  ;Slave Receiver Mode
        AJMP     MORE_70                                  ;Slave Receiver Mode
        AJMP     MORE_78                                  ;Slave Receiver Mode
        AJMP     MORE_80                                  ;Slave Receiver Mode
        AJMP     MORE_88                                  ;Slave Receiver Mode
        AJMP     MORE_90                                  ;Slave Receiver Mode
        AJMP     MORE_98                                  ;Slave Receiver Mode
        AJMP     MORE_A0                                  ;Slave Receiver Mode

        AJMP     MORE_A8                               ;Slave Transmitter Mode
        AJMP     MORE_B0                               ;Slave Transmitter Mode
        AJMP     MORE_B8                               ;Slave Transmitter Mode
        AJMP     MORE_C0                               ;Slave Transmitter Mode
        AJMP     MORE_C8                               ;Slave Transmitter Mode
;
;---------------------------------------------------------------------------
;State 00 = Bus error due to an illegal START or STOP condition.  This state
;can also occur if the SIO1 enters an undefined state.
;---------------------------------------------------------------------------
;
MORE_00:
        ANL     IIC_status,#status_type2_mask_;preserve upper nibble of status
        ORL     IIC_status,#status_buss_err_              ;indicate bus error
        CLR     SCL                                          ;unstick the bus
        CLR     SDA
        SETB    SCL
        SETB    SDA
M00_10:
        CALL    MORE_00_SUB           ;clear all ”IIC_OS2” status counters etc.
        JMP     IIC_EXIT
;
;This portion of State 00 was made into a subroutine so that the ”DO_IIC”
;routine could call it when a timeout error occurs.
;This routine sets all counters and other ”IIC_OS2” registers to 0.
;
MORE_00_SUB:
        CLR     A
        MOV     Data?adrs?space,A
        MOV     Attempt_count,A
        MOV     Multiple_count,A
        MOV     last_data,A
        MOV     S1CON,#ENS1_NOTSTA_STO_NOTSI_AA_CR0                      ;STOP
        RET
;
;----------------------------------------------------------------------------
;State 08 indicates that a start condition has been transmitted.
;MASTER RECEIVER/TRANSMITTER MODE.
;In this case, the ”IIC_OS2” possibilities are one – the next byte in the
;’IIC_Command_FIle’ must be the slave address (and read/write bit).
;----------------------------------------------------------------------------
;
MORE_08:
        CALL    FETCH_COMMAND                     ;get next byte in command file
        CJNE    A,#use_aux_adrs_,M08_05
        MOV     A,aux_adrs
M08_05:
        MOV     S1DAT,A                                                ;transmit it
M08_10:
        MOV     S1CON,#ENS1_NOTSTA_NOTSTO_NOTSI_NOTAA_CR0
        JMP     IIC_EXIT
;
;----------------------------------------------------------------------------
;State 10H = a repeated start condition has been transmitted.
;MASTER RECEIVER/TRANSMITTER MODE.
;This state is handled just like State 08H.  The ”IIC_OS2” definition ensures
;that the next byte in the ’IIC_Command_File’ will be a slave address (and
;read/write bit).
;----------------------------------------------------------------------------
;
MORE_10:
        JMP     MORE_08
;
;----------------------------------------------------------------------------
;State 18H = (Slave address + Write bit) has been transmitted, and an ACK has
;            been returned.
;MASTER TRANSMITTER MODE.
;Once a slave address has been transmitted, several possibilities exist,
;namely: set–up to send/receive n bytes with count and address info coming
;        from the command file stream
;                                      OR
;        set–up to send/receive n bytes with count and address info coming
;        from the ’indirect_count’ and ’indirect_adrs’ registers.  The
;        mainline routine must set these registers before initiating an IIC
;        session.
;                                      OR
;        set–up to send  data (’immediate_’) from the command file stream.
;                                      OR
;        set–up to send/receive data from the ’IIC_data_buffer’ data space
;        as indicated by the ’ioBuffer_’ memory space dirctive.
;----------------------------------------------------------------------------
;
MORE_18:
        MOV     Attempt_count,#0                     ;clear failed attempt count
M18_15:
        CALL    FETCH_COMMAND                     ;get next byte in command file
        CALL    make_space                             ;update ’Data?adrs?space’
        CALL    M18_SUB                           ;call subroutine to load count
M18_20:                                                  ;and address of data
        CALL    FETCH_DATA                           ;now ready to get data byte
```

```
M18_25:
        MOV     S1DAT,A                                            ;send as data
M18_X:
        MOV     S1CON,#ENS1_NOTSTA_NOTSTO_NOTSI_AA_CR0
        JMP     IIC_EXIT
        ;
        ;"M18_SUB" subroutine checks for 'indirect_'  command.
        ;State 18H and State 40H use this subroutine.
        ;If an indirect feature is requested, load address and count
        ;information from the 'indirect_count' and 'indirect_adrs' registers
        ;if the 'indirect_' feature is not requested, then the count and
        ;address information are contained in the next bytes of the command.
        ;file.
        ;
M18_SUB:
        JBC     IO_Buffer,M18S_10       ;is data target the 'IIC_data_buffer'?
        JBC     indirect_xxx,M18S_20       ;if indirect, clear bit and service
        JB      immediate_data,M18S_30  ;if immediate option, do not clear bit
        ;
        ;enter here if the count and address for the data to be read/written
        ;is in the command file itself (i.e. no special commands).  NOTE that
        ;if a command file has the number of data bytes to read/write set to
        ;0, then this routine interprets it as 255 (0FFH).
        ;
        CALL    FETCH_COMMAND                   ;get next byte in command file
        DEC     A                                              ;decrement and
        MOV     Multiple_count,A                         ;store as byte count
        CALL    FETCH_COMMAND                   ;get next byte in command file
        MOV     IO_buffer_adrs,A              ;store as LSByte of data address
        JB      IO_Data,M18S_X             ;if DATA space, only 1 address byte
        CALL    FETCH_COMMAND                   ;get next byte in command file
        MOV     IO_buffer_adrs+1,A            ;store as MSByte of data address
M18S_X:
        RET
        ;
        ;enter here if we are to use the 'IIC_data_buffer' as a source or
        ;destination for data transfers (as indicated by the 'ioBuffer_'
        ;directive in the command file).
        ;
M18S_10:
        CALL    FETCH_COMMAND
        DEC     A
        MOV     Multiple_count,A
        MOV     A,#ioD_                       ;transmit bytes from DATA space
        CALL    make_space                          ;update 'Data?adrs?space'
        MOV     IO_buffer_adrs,#IIC_data_buffer
        RET
        ;
        ;enter here if indirect requested.  The number of bytes to be
        ;written or read is contained in the 'indirect_count' register,
        ;the address of the bytes to be read or written is contained in
        ;the 'indirect_address' register(s).
        ;
M18S_20:
        DEC     indirect_count
        MOV     Multiple_count,indirect_count
        MOV     IO_buffer_adrs,indirect_adrs
        MOV     IO_buffer_adrs + 1,indirect_adrs + 1
        RET
        ;
        ;enter here if the immediate option is active.  In this case, the
        ;next byte in the data stream is the number of bytes to be sent.
        ;The bytes to be sent are also in the command file stream and is
        ;handled by the ”FETCH_DATA” routine.
        ;
M18S_30:
        CALL    FETCH_COMMAND
        DEC     A
        MOV     Multiple_count,A
        RET
;
;–––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––
;State 20H = (Slave address + Write bit) has been transmitted, no ACK from
;slave.
;MASTER TRANSMITTER MODE.
;This state counts the number of failures for a transmitted address – if
;’max_adrs_attempts_’ failures occur in–a–row, then abort session.
;–––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––
;
MORE_20:
        INC     Attempt_count                              ;bump attempt count
        MOV     A,Attempt_count
        CJNE    A,#max_adrs_attempts_,M20_10            ;if too many failures,
        ANL     IIC_status,#status_type2_mask_
        ORL     IIC_status,#status_attempt_adrs_       ;indicate attempt error
        JMP     M00_10
        ;
        ;if less than ’max_attempts’ failures, then set command file pointer
        ;back one, and try sending address again.
        ;
M20_10:
        MOV     R0,#IIC_Command_File_adrs
        MOV     A,@R0
        JNZ     M20_20
        INC     R0
        DEC     @R0
        DEC     R0
M20_20:
        DEC     @R0

        JMP     M28_10
;
;–––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––
;State 28H = Data byte has been transmitteed, ACK has been received.
;MASTER TRANSMITTER MODE.
;This section is entered when a data byte has been successfully transmitted.
;Now the system has to check if more data bytes are to be sent, and if not,
;should a subroutine be called before going on to next IIC block in the
;IIC command file.
;–––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––
;
MORE_28:
        MOV     Attempt_count,#0             ;clear attempt count since all OK
        MOV     A,Multiple_count              ;check for end of data bytes out
        JZ      M28_03                                ;last byte has been sent
        DEC     Multiple_count             ;more bytes to be sent so decrement
        JMP     M18_20                               ;count and send next byte
        ;
        ;enter here when all data bytes sent.
        ;
M28_03:
        JBC     call_function,M28_20     ;check for request to call subroutine
M28_05:
        CALL    FETCH_COMMAND_0                      ;check for end–of–session
        CJNE    A,#iic_end_,M28_10
M28_X:
        ANL     IIC_status,#status_type2_mask_
        ORL     IIC_status,#status_OK_
        JMP     M00_10
        ;
        ;if not end–of–session, do another start
        ;
M28_10:
        MOV     S1CON,#ENS1_STA_NOTSTO_NOTSI_AA_CR0   ;impose a repeated start
        JMP     IIC_EXIT
        ;
        ;enter here if a ’call_’ to a subroutine is requested.  First push
        ;the return address (above) onto stack, then get the address of the
        ;subroutine to call from the IIC command file.  Push the call address
        ;onto the stack (low address first), then call subroutine.
        ;
        ;The called subroutine could be used to modify the contents of the
        ;’IIC_Command_File_adrs’ registers.  In doing so, IF–THEN–ELSE
        ;control flow could be done (i.e. based on some  IIC read information,
        ;the subroutine may decide to run one of several other IIC blocks,
        ;or end the session altogether).  More likely, the subroutine will be
        ;used to manipulate some data before it is transmitted.
        ;
M28_20:
        MOV     A,#LOW(M28_05)                  ;put return address onto stack
        PUSH    ACC
        MOV     A,#HIGH(M28_05)
        PUSH    ACC
        CALL    FETCH_COMMAND     ;get address of subroutine from command file
        PUSH    ACC                     ;and put it onto the stack (LSB first)
        CALL    FETCH_COMMAND
        PUSH    ACC
        RET                                               ;CALL the subroutine
;
;–––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––
;State 30H = Data byte has been transmitted, NO ACK received.
;MASTER TRANSMITTER MODE.
;This state is similar to state 20H, except that data has been transmitted,
;not an address.
;The routine ’FETCH_DATA’ always stores the data fetched as ’last_data’ so
;that in the case of a NO ACK, it can be re–transmitted.
;–––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––
;
MORE_30:
        INC     Attempt_count                              ;bump attempt count
        MOV     A,Attempt_count                     ;if too many attempt, error
        CJNE    A,#max_data_attempts_,M30_10
        ANL     IIC_status,#status_type2_mask_
        ORL     IIC_status,#status_attempt_data_         ;error status update
        JMP     M00_10

M30_10:
        MOV     A,last_data                         ;get data not received and
        JMP     M18_25                                            ;re–send it
;
;–––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––
;State 38H = Arbitration lost to another master.
;MASTER TRANSMITTER MODE.
;If this state is entered, simply let the other Master have the run of the
;bus.  The mainline routine that started the IIC session can check
```

```
;the 'IIC_status' register for this state and re-try later.
;-----------------------------------------------------------------------------
;
MORE_38:
        ANL     IIC_status,#status_type2_mask_
        ORL     IIC_status,#status_arb_lost_              ;error status update
        JMP     M40_30
;
;-----------------------------------------------------------------------------
;State 40H = (Slave Address + Read bit) has been transmitted, ACK received.
;MASTER RECEIVER MODE.
;This state is very similar to State 18H and shares a subroutine with that
;state.  The byte retrieved from the command file here is the memory space
;directive byte.  Since this is a read operation, one cannot read into CODE
;space.
;------------------------------------------------------------------------------
;
MORE_40:
        MOV     Attempt_count,#0
        CALL    FETCH_COMMAND
        CALL    make_space
        JNB     immediate_data,M40_10
        ANL     IIC_status,#status_type2_mask_
        ORL     IIC_status,#status_bad_read_space_ ;can't write to CODE space
        JMP     M00_10
M40_10:
        CALL    M18_SUB
M40_20:
        MOV     A,Multiple_count
        JNZ     M40_30
        MOV     S1CON,#ENS1_NOTSTA_NOTSTO_NOTSI_NOTAA_CR0
        JMP     IIC_EXIT
M40_30:
        MOV     S1CON,#ENS1_NOTSTA_NOTSTO_NOTSI_AA_CR0
        JMP     IIC_EXIT
;
;-----------------------------------------------------------------------------
;State 48H = (Slave address + Read bit) transmitted, NOT ACK received.
;MASTER RECEIVER MODE.
;See State 20H.
;-----------------------------------------------------------------------------
;
MORE_48:
        JMP     MORE_20
;
;-----------------------------------------------------------------------------
;State 50H = Data byte has been received, ACK returned.
;MASTER RECEIVER MODE.
;This state stores the received data byte and determines whether more data is
;required or not.  If more data required (i.e. 'Multiple_count' > 1), then
;send back an ACK, if no more data to be received ('Multiple_count" = 1), then
;set-up to return a NOT ACK on next data byte reception.
;-----------------------------------------------------------------------------
;
MORE_50:
        MOV     Attempt_count,#0                              ;indicate all OK
        MOV     A,S1DAT                                ;get received data byte
        CALL    STORE_DATA                ;store the data in appropriate space
        MOV     A,Multiple_count          ;check for more bytes to be received
        CJNE    A,#1,M50_10
        ;
        ;If next byte to be received is last, make sure a NOT ACK is sent
        ;with next reception.
        ;
M50_05:
        MOV     S1CON,#ENS1_NOTSTA_NOTSTO_NOTSI_NOTAA_CR0
        SJMP    M50_15
M50_10:
        MOV     S1CON,#ENS1_NOTSTA_NOTSTO_NOTSI_AA_CR0
M50_15:
        DEC     Multiple_count
        JMP     IIC_EXIT
;
;-----------------------------------------------------------------------------
;State 58H = Data byte has been received, NOT ACK has been returned.
;MASTER RECEIVER MODE.
;This state is entered when the last byte required has been received by the
;Master.  In this case, the byte must be stored, then a check must be done
;for the calling of a subroutine, and/or the end of the entire IIC session.
;See State 28H for more details.
;-----------------------------------------------------------------------------
;
MORE_58:
        MOV     A,S1DAT                                     ;get received byte
        CALL    STORE_DATA                                          ;store it
        MOV     Attempt_count,#0                             ;clear error flag
        JMP     M28_03                  ;check for end–of–session or 'call_'
;
;-----------------------------------------------------------------------------
;State 60H = Own Slave Address (+ Write bit) has been received,
;            ACK has been returned.
;SLAVE RECEIVER MODE.
;When own address found, this system will receive 'SLVbytes_in_' bytes of data
;into 'Slave_in' data space.
;The calling master must produce the stop or repeated start conditions.  This
;micro was not in an active IIC mode when the other master addressed it, so
;the "DO_IIC" subroutine is not active, thus timeout problems will not be
;checked for unless "DO_IIC" is called.  "DO_IIC" will do only a timeout
;check if called from the main program since it will wait for the 'IIC_status'
;to become 'status_OK_'.
;-----------------------------------------------------------------------------
;
MORE_60:
        MOV     IIC_status,#status_slave_
M60_10:
        MOV     Multiple_count,#(SLVbytes_in_ – 1)   ;set for # bytes received
        MOV     Data?adrs?space,#ioD_           ;receive bytes into DATA space
        SETB    i_am_a_slave         ;set slave indicator in 'Data?adrs?space'
        MOV     IO_buffer_adrs,#Slave_in         ;address of DATA space target
        SJMP    M40_30
;
;-----------------------------------------------------------------------------
;State 68H = Arbitration lost while addressing a slave; Own slave address and
;            write bit has been received.
;SLAVE RECEIVER MODE.
;Indicate that arbitration is lost so that the "DO_IIC" routine is aborted
;and the interrupt from the IIC hardware runs the system.
;"DO_IIC" is active if this state is entered since state 68H is entered
;upon lost arbitration for the bus.
;"DO_IIC" will terminate in this case since the 'IIC_status' will show that
;another master has won the bus.
;-----------------------------------------------------------------------------
```

```
;
MORE_68:
        MOV     IIC_status,#status_arb_lost_slave_w_
        SJMP    M60_10
;
;-----------------------------------------------------------------------------
;State 70H = General call address (00H) has been received, ACK has been
;            returned (by this micro).
;SLAVE RECEIVER MODE.
;Indicates that a general call has been received – 'SLVbytes_in_' bytes will
;be received into 'Slave_in' as if this slave were addressed.
;-----------------------------------------------------------------------------
;
MORE_70:
        MOV     IIC_status,#status_general_slave_
        SJMP    M60_10
;
;-----------------------------------------------------------------------------
;State 78H = Arbitration lost while addressing a slave – General call address
;            (00H) has been received, ACK has been returned (by this micro).
;SLAVE RECEIVER MODE.
;Indicates that a general call has been received. 'SLVbytes_in_' bytes will be
;received into 'Slave_in' as if this slave were addressed.
;-----------------------------------------------------------------------------
;
MORE_78:
        MOV     IIC_status,#status_arb_lost_general_
        SJMP    M60_10
;
;-----------------------------------------------------------------------------
;State 80H = Previously addressed with own slave address; data has been
;            received, ACK has been returned (by this micro).
;SLAVE RECEIVER MODE.
;Data byte received in 'S1DAT', ACK returned.
;-----------------------------------------------------------------------------
;
MORE_80:
        SJMP    MORE_50
;
;-----------------------------------------------------------------------------
;State 88H = Previously addressed with own slave address; data byte has been
;            received, NOT ACK has been returned (by this micro).
;SLAVE RECEIVER MODE.
;Last byte to be received is in 'S1DAT'.  A NACK has been returned.
;-----------------------------------------------------------------------------
;
MORE_88:
        MOV     A,S1DAT                                     ;get received byte
        CALL    STORE_DATA                                  ;store it
M88_05:
        CALL    Slave_receive_done
M88_10:
        ANL     IIC_status,#status_type2_mask_
        ORL     IIC_status,#status_OK_
        CLR     i_am_a_slave
        SJMP    M40_30
;
;-----------------------------------------------------------------------------
;State 90H = Previously addressed with general call; data byte has been
;            received, ACK has been returned (by this micro).
;SLAVE RECEIVER MODE.
```

```
;---------------------------------------------------------------------------
;
MORE_90:
        SJMP    MORE_80
;
;---------------------------------------------------------------------------
;State 98H = Previously addressed with general call; data byte has been
;            received, NOT ACK has been returned (by this micro).
;SLAVE RECEIVER MODE.
;---------------------------------------------------------------------------
;
MORE_98:
        SJMP    MORE_88
;
;---------------------------------------------------------------------------
;State A0H = a STOP or repeated START has been received while still in the
;            addressed slave receiver or transmitter mode.
;SLAVE RECEIVER MODE.
;---------------------------------------------------------------------------
;
MORE_A0:
        SJMP    M88_05
;
;---------------------------------------------------------------------------
;State A8H = Own slave address + read byte has been received; ACK has been
;            returned (by this micro).
;SLAVE TRANSMITTER MODE.
;This micro has been addressed by another master, and has been told to send
;data.  This micro will respond by sending 'SLVbytes_out_' bytes of data from
;'Slave_out'.
;---------------------------------------------------------------------------
;
MORE_A8:
        MOV     IIC_status,#status_slave_
MA8_10:
        MOV     Multiple_count,#(SLVbytes_out_)    ;set for 2 bytes to be sent
        MOV     Data?adrs?space,#ioD_          ;transmit bytes from DATA space
        SETB    i_am_a_slave         ;set slave indicator in 'Data?adrs?space'
        MOV     IO_buffer_adrs,#Slave_out        ;address of DATA space target
        SJMP    MORE_B8
;
;---------------------------------------------------------------------------
;State B0H = Arbitration lost while trying to get to a slave; own slave
;            address + read has been received; ACK has been returned (by this
;            micro).
;SLAVE TRANSMITTER MODE.
;---------------------------------------------------------------------------
;
MORE_B0:
        MOV     IIC_status,#status_arb_lost_slave_r_
        SJMP    MA8_10
;
;---------------------------------------------------------------------------
;State B8H = Data byte in S1DAT has been transmitted; ACK has been received.
;SLAVE TRANSMITTER MODE.
;This section checks if any more bytes are to be transmitted.
;---------------------------------------------------------------------------
;
MORE_B8:
        MOV     A,Multiple_count
        JZ      MB8_03
        DEC     Multiple_count
MB8_03:
        CALL    FETCH_DATA                         ;now ready to get data byte
        MOV     S1DAT,A                                          ;send as data
        JMP     M40_30
;
;-----------------------------------------------------------------------------
;State C0H = Data byte in S1DAT has been transmitted; NOT ACK has been
;            received.
;SLAVE TRANSMITTER MODE.
;This is the end of the addressed slave session.  A STOP or repeated START
;will be the next state, but this addressed slave don't care unless the next
;address sent by the calling master is it's own, or the general call address.
;-----------------------------------------------------------------------------
;
MORE_C0:
        CALL    Slave_transmit_done
        JMP     M88_10
;
;-----------------------------------------------------------------------------
;State C8H = Last data byte in S1DAT has been transmitted; ACK has been
;            received.
;SLAVE TRANSMITTER MODE.
;Treated same as state C0.
;-----------------------------------------------------------------------------
;
MORE_C8:
        JMP     MORE_C0
;
;=============================================================================
;"Slave_xxxx_done" is called immediately after all bytes are received or
;transmitted in a slave receive or transmit mode.  These subroutines could
;be used to accomodate some inter-processor protocol.
;
Slave_receive_done:
        RET
Slave_transmit_done:
        RET

CODE_start      EQU     $
END
```

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

Date of release: 06-98

Document order number: 9397 750 04054

Let's make things better.

Philips
Semiconductors

PHILIPS