I^{2}C routines for 8XC528 — AN438

Philips Semiconductors Application Note EIE/AN90015

Summary
This application note presents a set of software routines to drive the I^{2}C interface in 8xC528 type of microcontrollers. A description of the I^{2}C interface is given. Examples show how to use these routines in PL/M-51, C and assembly source code.

1.0 INTRODUCTION

This application note describes the I^{2}C interface of the 8xC528 microcontroller and gives a set of routines in application programs to drive this interface.

Chapter 2.0 gives a hardware description of the bit level I^{2}C. It gives an overview of what functions are done in hardware by the interface and the functions that should be implemented by software. The registers described are accessible with software and control the I^{2}C interface.

Chapter 3.0 gives a description of the routines that may be used by the application program. The routines are written in such a way that the I^{2}C interface becomes transparent to the user. the slave program is described in more detail, because this routine may be adapted by the user for his specific application.

Chapter 4.0 gives simple example programs that show how to use the routines in assemble, PL/M and C application programs.

References:

– The I^{2}C-bus specification	9398 358 10011
– 80C51-based 8-bit Microcontrollers	Data handbook IC20
– PLM51 I^{2}C Software Interface IIC51	ETV/AN89004

2.0 THE I^{2}C INTERFACE

2.1 Characteristics of I^{2}C Interface

The Block diagram of the bit-level I^{2}C interface is shown on page 2. P1.6/SCL and P1.7/SDA are the serial I/O pins. These two pins meet the I^{2}C specification concerning the input levels and output drive capability. Consequently, these pins have an open drain configuration. All four modes of the I^{2}C bus can be used:

– Master transmitter

– Master receiver

– Slave transmitter

– Slave receiver

The advantages of using the bit-level I^{2}C hardware compared with a full software implementation are:

– Higher bit rate

– No critical software timing requirements

– Less software overhead

– More reliable data transfer

The bit-level I^{2}C hardware can perform the following functions:

– Filtering the incoming serial data and clock signals. Glitches shorter than 4 XTAL periods are rejected.

– Recognition of a START or STOP condition.

– Generating an interrupt request after reception of a START condition.

– Setting the Bus Busy flag when a START condition is detected.

– Clearing the Bus Busy flag when a STOP condition is detected.

– Recognition of a serial clock pulse on the SCL line.

– Latching the serial data bit on the SDA line at every rising edge on the SCL line.

– Stretching the LOW period of the serial clock SCL to synchronizer with external master devices.

– Setting the Read Bit Finished (RBF) or Write Bit Finished (WBF) flag is an error free bit transfer has occurred.

– Setting a Clock LOW-to-HIGH (CLH) flag when a leading edge is detected on the SCL line.

– Generation of serial clock pulse on SCL in master mode.

The following functions must be done with software:

– Handling the I^{2}C interrupt caused by a detected START condition.

– Conversion of serial to parallel data when receiving.

– Conversion of parallel to serial data when transmitting.

– Comparing received slave address with own slave address.

– Interpretation of acknowledge information.

– Guarding the I^{2}C status if the RBF and WBF flags indicate a not regular bit transfer.

– Generating START/STOP conditions when in master mode.

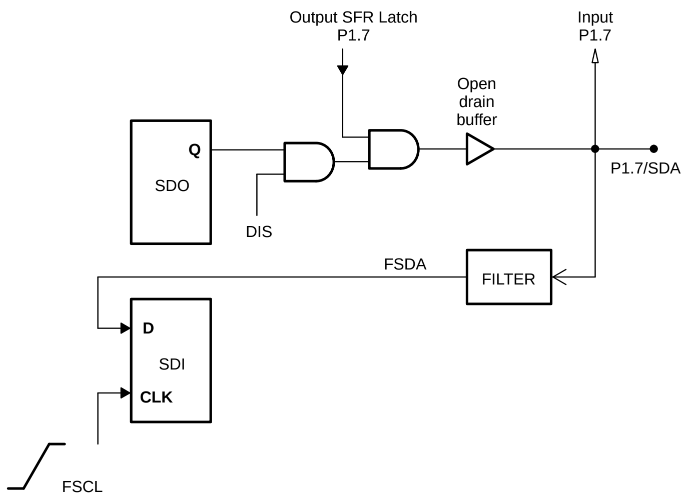
– Handling bus arbitration when in master mode.

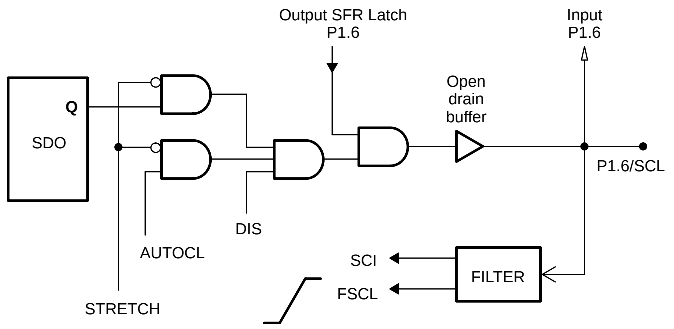
Special Function Registers

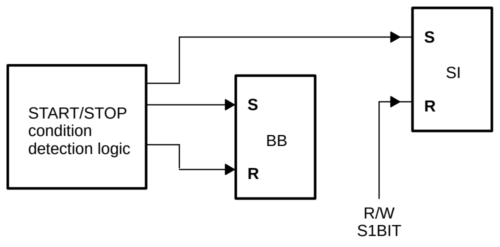
S1INT (DAh)	S1 1)	X	X	X	X	X	X	X	R/W
	7	6	5	4	3	2	1	0	
S1BIT	SDI	0	0	0	0	0	0	0	R
(D9h)	SDO	X	X	X	X	X	X	X	W
Bit Address	DF	DE	DD	DC	DB	DA	D9	D8	
S1SCS	SDI	SCI	CLH	BB	RBF	WBF	STR	ENS	R
(D8h)	SDO 2)	SCO 2)	CLH 1)	X	X	X	STR	ENS	W

1) Software can only clear this bit
2) This bit is read with read–modify–write operation

X =Undefined (R) or don't care (W)
R = Read access
W = Write access







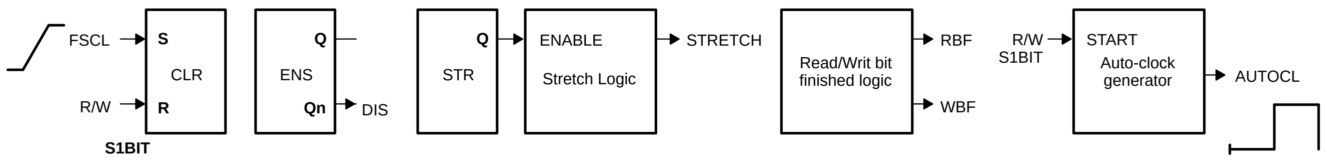


Figure 1.

2.2 Control and Status Registers

Control of the I^{2}C bus hardware is done via 3 Special Function Registers:

S1INT
This register contains the serial interrupt flag SI.

S1BIT
For read, this register contains the received bit SDI.
For write, this register contains bit SDO to be transmitted.

S1SCS
For read, this register contains status information.
For write, this register is used as control register.

2.2.1 S1INT: I^{2}C Interrupt Register

– **S1INT.7** is the Serial Interrupt request flag (SI).

If the serial I/O is enabled (ENS = 1), then a START condition will be detected and the SI flag is set on the falling edge of the filtered SCL signal.

Provided that EA (global enable) and ES1 (enable I^{2}C interrupt) are set (in the interrupt enable IE register), SI generates an interrupt that will start the slave address receive routine.

SI is cleared by accessing the S1BIT register or by writing '00H' to S1INT. SI cannot be set by software.

After reception of a START condition, the LOW period of the SCL pulse is stretched, suspending serial transfer to allow the software to take appropriate action. This clock stretching is ended by accessing the S1BIT register.

2.2.2 S1BIT: Single Bit Data Register

– **S1BIT.7** contains two physical latches: the Serial Data Output (SDO) latch for a write operation, and the filtered Serial Data Input (SDI) latch for a read operation. SDI data is latched on the rising edge of the filtered SCL pulse. S1BIT.7 accesses the same physical latches as S1SCS.7, but S1BIT.7 is not bit addressable.

Reading or writing S1BIT register starts the next additional actions:
- SI, CLH, RBF and WBF flags are cleared.
- Stretching the LOW period of the SCL clock is finished.
- Auto-clock pulse is started if enabled.

The auto-lock is an active HIGH SCL pulse that starts 28 Xtal periods after an access to S1BIT. SCL remains high for 100 Xtal periods. If the SCL line is kept LOW by any device that wants to hold up the bus transfer, the auto-clock counter still runs for 20 Xtal periods to try to make SCL high and then go into a wait-state. This will result in a minimum SCL HIGH time of 80 Xtal periods (5µs at f_{Xtal} = 16 MHz).

The auto-clock signal will be inhibited if the SCO flag in the S1SCS register is set to '1'. SCL pulses must then be generated by software. In this situation, access to S1BIT may be used to clear the SI, CLH, RBF and WBF flags.

A quick check on a successful bit transfer from/to SDO/SDI is carried out be testing only the RBF or WBF flag (see 2.2.3).

2.2.3 S1SCS: Control and Status Register

– **S1SCS.7** represents two physical latches, the Serial Data Output (SDO) latch for write operations and the Serial Data Input (SDI) latch for read operations. S1SCS.7 accesses the same physical latches as S1BIT.7, but S1SCS.7 is bit addressable. However, a read or write operation of S1SCS.7 does not start an auto-lock pulse, with not finish clock stretching, and will not clear flags.

– **S1SCS.6** represents two physical latches, the Serial Clock Output (SCO) latch for write operations and the Serial Clock Input (SCI) latch for read operations. The output of SCO is "OR-ed" with the auto-clock pulse. If SCO = '1' the auto-clock generation is disabled and its output is LOW. Internal clock stretching logic and external devices can then pull the SCL line LOW.

If the auto-clock is not used, the SCL line has to be controlled by setting SCO = '1', waiting for CLH to become '1' and setting SCO = '0' after the specified SCL HIGH time. Data access should be done via S1SCS.7.

– **S1SCS.5** is the serial Clock LOW-to-HIGH transition flag (CLH). This flag is set by a rising edge of the filtered serial clock. CLH = '1' indicates that no devices are stretching SCL LOW, and since the last CLH reset, a new valid data bit has been latched in SDI.

CLH can be cleared by writing '0' to S1SCS.5 or by a read or write operation to the S1BIT register. Clearing CLH also clears RBF and WBF. Writing a '1' to S1SCS.5 will not affect CLH.

– **S1SCS.4** is the Bus Busy flag (BB). BB is set or cleared by hardware only. If set, it indicates that a START condition has been detected on the I^{2}C bus. A STOP condition clears the BB flag.

– **S1SCS.3** is the Read Bit Finished flag (RBF). If RBF = 1, it indicates that a serial bit has been received and latched into SDI successfully. If during a bit transfer RBF is '0', the cause is indicated as follows:

SCI = '1' and CLH = '1'	The SCL pulse is not finished and still HIGH.
CLH = '0'	A bus device is delaying the transfer by stretching the LOW level on the SCL line.
BB = '0'	A STOP-condition has been detected during the bit transfer. This should be considered as a bus-error.
SI = '1'	A START-condition has been detected during the bit transfer. This should be considered as a bus-error.

RBF can be cleared by clearing CLH or by a read or write operation to the S1BIT register.

– **S1SCS.2** is the Write Bit Finished flag (WBF). If set, it indicates that a serial bit in SDO has been transmitted successfully. If during bit transfer WBF is '0', the following conditions may be the cause:

SCI = '1' and CLH = '1'	The SCL pulse is not finished and still HIGH.
CLH = '0'	A bus device is delaying the transfer by stretching the LOW level on the SCL line.
BB = '0'	A STOP-condition has been detected during the bit transfer. This should be considered as a bus-error.
SI = '1'	A START-condition has been detected during the bit transfer. This should be considered as a bus-error.

WBF can be cleared by clearing CLH or access to the S1BIT register.

– **S1SCS.1** is the STRetch control flag (STR). STR can be set or cleared by software only. Setting STR enables the stretching of SCL LOW periods. Stretching will occur after a falling edge on the filtered serial clock. This allows synchronization with the SCL clock signal of an external master device.

  If STR is cleared, no stretching of the SCL LOW period will occur after the transfer of a serial bit.

  The LOW level on the SCL line is also stretched after a START condition is received, regardless of the STR contents. The stretching of the SCL LOW period is finished by a read or write operation of the S1BIT register.

– **S1SCS.0** is the ENable Serial I/O flag (ENS).

  ENS can be set or cleared by software only.

  ENS = '0' disables the serial I/O. The I/O signals P1.6/SCL and P1.7/SDA are determined by the port latches of P1.6 and P1.7 (open drain). If P1.6 and P1.7 are connected to an I^{2}C bus, then the flags SDI, SCI, CLH, and BB still monitor the I^{2}C bus status, but will not influence the I/O lines, nor will they request an interrupt.

  ENS = '1' enables the START detection and clock stretching logic. Note that the P1.6 and P1.7 latches and the SDO and SCO control flags must be set to '1' before ENS is set to avoid SCL and/or SDA to pull the lines LOW.

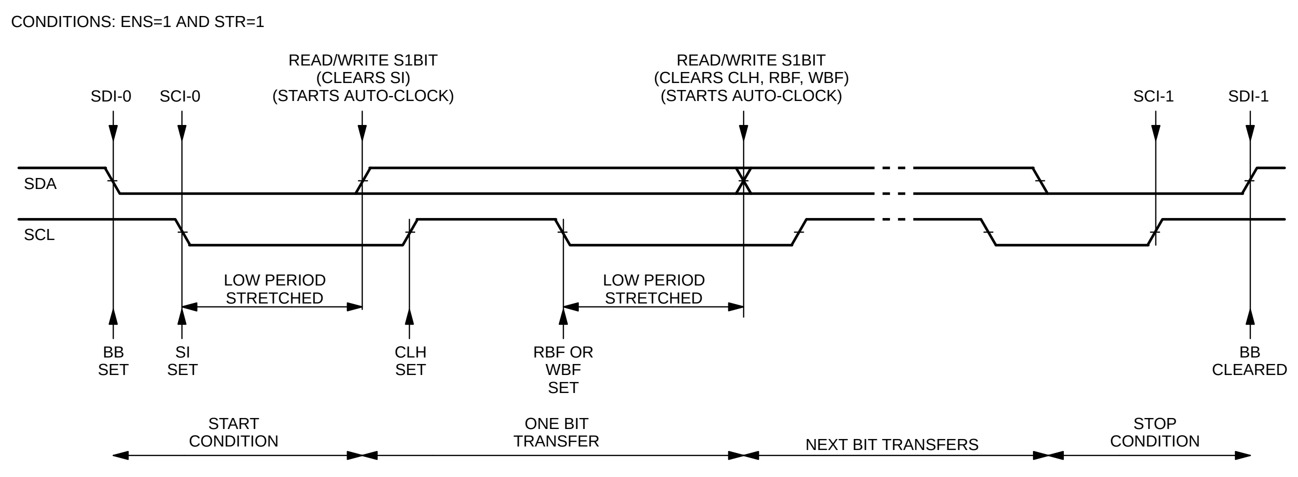


Figure 2. Example of a Serial Transfer

3.0 I^{2}C ROUTINES

3.1 Introduction

A set of routines is written for the I^{2}C interface that supports multi-master and slave operation. The routines are placed in a library I2C_DR.LIB. If I2C_DR.LIB is linked to an application program, only the needed object modules are linked in the output file.

The routines can be used as device driver for PL/M-51, C and 8051-assembly code. By using these routines the bit-level I^{2}C interface is fully transparent for the user.

The routines use the following 8xC528 resources:

– Exclusive use of Register_Bank_1. Only R7 of this register bank contains static data (Own Slave Address). R0..R6 may be used by the application program when the I^{2}C routine is finished.
– 7 bytes DATA used for parameter passing.
– 1 byte Bit-Addressable DATA for status flags.

When using routines from this library DPH, DPL, PSW (except CY) and B are not altered.

An n-bytes data buffer is used as destination or source buffer for the bytes to be received/transmitted and reside in DATA or IDATA memory space.

The code is written to generate the highest transfer rate on the I^{2}C bus. At f_{Xtal} = 16MHz this will result in a bit rate of 87.5kbit/sec.

The following software tools from BSO/Tasking are used for program development:

– OM4142 Cross Assembler 8051 for DOS: V3.0b
– OM4144 PL/M 8051 Compiler for DOS: V3.0a
– OM4136 C8051 Compiler for DOS: V1.1a
– OM4129 XRAY51 debugger: V1.4c

3.2 Functional Description

When using these routines in a PL/M application program, they must be declared EXTERNAL. In this declaration the user can specify the type returned by each procedure. All procedures (except Init_IIC and Dis_IIC) can return a BIT or BYTE, depending on the chosen EXTERNAL declaration. The BIT or BYTE returned is ‘0’ if the I^{2}C was successful. If a BYTE is returned, the following check bits are available for the user:

BYTE.0	An I^{2}C error has been detected.
BYTE.1	No ACK received.
BYTE.2	Arbitration lost.
BYTE.3	Time out error. This may be caused by an external device pulling SCL LOW.
BYTE.4	A bus error has occurred. This may be a spurious START/STOP during a bit transfer.
BYTE.5	No access to I^{2}C bus.
BYTE.6	0
BYTE.7	0

Note that typed procedures must be called using an expression. If the result of an I^{2}C procedure is to be ignored, a dummy assignment must be done for a typed procedure. The examples in the following section assume that the procedures are called from a PL/M program. Examples will be given later how to use these routines with C and assembly application programs.

3.2.1 Init_IIC

Declaration

```
Init_IIC:
    PROCEDURE (Own_Slave_Address, Slave_Sub_Address)
        EXTERNAL;
    DECLARE (Own_Slave_Address, Slave_Sub_Address) BYTE;
    END;
```

Description

Init_IIC must be called after RESET, before any procedure is called. The I^{2}C interface and I^{2}C interrupt will be enabled. The global enable interrupt flag, however, will not be affected. This should be done afterwards. Own_Slave_Address is passed to Init_IIC for use as slave. Slave_Sub_Address is the pointer to a DATA buffer that is used for data transfer in slave mode. When used as master in a single master system, these parameters are not used.

Example

```
CALL Init_IIC (54h,.Slave_Data_Buffer);
ENABLE; /* Enable Interrupts; EA=1 */
```

3.2.2 Dis_IIC

Declaration

```
Dis_IIC:
    PROCEDURE EXTERNAL;
```

Description

Dis_IIC will disable the I^{2}C-interface and the I^{2}C-interrupt. The I^{2}C interface will still monitor the bus, but will not influence the SDA and SCL lines.

Example

```
CALL Dis_IIC;
```

3.2.3 IIC_Test_Device

Declaration

```
IIC_Test_Device:
    PROCEDURE (Slave_Address) [BIT|BYTE] EXTERNAL;
    DECLARE (Slave_Address) BYTE;
    END;
```

Description

IIC_Test_Device just sends the slave address to the I^{2}C bus. It can be used to check the presence of a device on the I^{2}C bus.

I^{2}C Protocol

S-SlvW-A-P	: Device is present, IIC_Error=0
S-SlvW-N-P	: Device is not present, IIC_Error=1

Example

```
DECLARE IIC_Error BIT;
.....
IIC_Error=IIC_Test_Device(8Ch);
IF (IIC_Error) THEN
    “Device not acknowledging on slave address”
ELSE
    “Device acknowledges on slave address”
```

3.2.4 IIC_Write

Declaration

```
IIC_Write:
    PROCEDURE (Slave_Address, Count, Source_Ptr)
        [BIT|BYTE] EXTERNAL;
    DECLARE (Slave_Address, Count, Source_Ptr) BYTE;
    END;
```

Description

IIC_Write is the most basic procedure to write a message to a slave device.

I^{2}C Protocol

L	=Count
D1[0..L–1]	BASED by Source_Ptr

S-SlvW-A-D1[0]-A....A-D1[L-1]-A-P

Example

```
DECLARE Data_Buffer(4) BYTE;
.....
CALL IIC_Write(02Ch, LENGTH(Data_Buffer),.Data_Buffer);
```

3.2.5 IIC_Write_Sub

Declaration

```
IIC_Write_Sub:
    PROCEDURE (Slave_Address, Count, Source_Ptr,
        Sub_Address) [BIT|BYTE] EXTERNAL;
    DECLARE (Slave_Address, Count, Source_Ptr, Sub_Address)
        BYTE;
    END;
```

Description

IIC_Write_Sub writes a message preceded by a sub-address to a slave device.

I^{2}C Protocol

L	=Count
Sub	=Sub_Address
D1[0..L–1]	BASED by Source_Ptr

S-SlvW-A-Sub-A-D1[0]-A-D1[1]-A....A-D1[L-1]-A-P

Example

```
DECLARE Data_Buffer(8) BYTE;
.....
CALL IIC_Write_Sub (48h,LENGTH(Data_Buffer),.Data_Buffer,2);
```

3.2.6 IIC_Write_Sub_SWInc

Declaration

```
IIC_Write_Sub_SWInc:
    PROCEDURE (Slave_Address, Count, Source_Ptr,
        Sub_Address) [BIT|BYTE] EXTERNAL;
    DECLARE (Slave_Address, Count, Source_Ptr, Sub_Address)
        BYTE;
    END;
```

Description

Some I^{2}C devices addressed with a sub-address do not automatically increment the sub-address after reception of each byte. IIC_Write_Sub_SWInc can be used for such devices the same way as IIC_Write_Sub is used. IIC_Write_Sub_SWInc splits up the message in smaller messages and increments the sub-address itself.

I^{2}C Protocol

L	=Count
Sub	=Sub_Address
D1[0..L–1]	BASED by Source_Ptr

```
S-SlvW-A- (Sub+0) – A-D1[0] – A-P
S-SlvW-A- (Sub+1) – A-D1[1] – A-P
...............................
S-SlvW-A- (Sub+L–1)-A-D1[L-1]-A-P
```

Example

```
DECLARE Data_Buffer(6) BYTE;
.....
CALL IIC_Write_Sub_SWInc(80h,LENGTH
    (Data_Buffer),.Data_Buffer,2);
```

3.2.7 IIC_Write_Memory

Declaration

```
IIC_Write_Memory:
    PROCEDURE (Slave_Address, Count, Source_Ptr,
        Sub_Address) [BIT|BYTE] EXTERNAL;
    DECLARE (Slave_Address, Count, Source_Ptr, Sub_Address)
        BYTE;
    END;
```

Description

I^{2}C Non-Volatile Memory devices (such as PCF8582) need an additional delay after writing a byte to it. IIC_Write_Memory can be used to write to such devices the same way IIC_Write_Sub is used. IIC_Write_Memory splits up the message in smaller messages and increments the sub-address itself. After transmission of each message a delay of 40 milliseconds (f_{Xtal} = 16 MHz) is inserted.

I^{2}C Protocol

L	=Count
Sub	=Sub_Address
D1[0..L–1]	BASED by Source_Ptr

```
S-SlvW-A- (Sub+0) – A-D1[0] – A-P
    Delay 40ms
S-SlvW-A- (Sub+1) – A-D1[1] – A-P
    Delay 40ms
...............................
S-SlvW-A- (Sub+L–1)-A-D1[L-1]-A-P
    Delay 40ms
```

Example

```
DECLARE Data_Buffer(10) BYTE;
.....
CALL IIC_Write_Memory(0A0h,LENGTH
    (Data_Buffer),.Data_Buffer,0F0h);
```

3.2.8 IIC_Write_Sub_Write

Declaration

```
IIC_Write_Sub_Write:
   PROCEDURE (Slave_Address, Count1, Source_Ptr1,
      Sub_Address, Count2, Source_Ptr2)
      [BIT|BYTE] EXTERNAL;
   DECLARE (Slave_Address, Count1, Source_Ptr1,
      Sub_Address, Count2, Source_Ptr2) BYTE;
   END;
```

Description

IIC_Write_Sub_Write writes 2 data blocks preceded by a sub-address in one message to a slave device. This procedure can be used for devices that need an extended addressing method, without the need to put all data into one large buffer. Such a device is the ECCT (I^{2}C controlled teletext device; see example).

I^{2}C Protocol

L	=Count1
M	=Count2
Sub	=Sub_Address
D1[0..L–1]	BASED by Source_Ptr1
D2[0..M–1]	BASED by Source_Ptr2

```
S-SlvW-A-Sub-A-D1[0]-A-D1[1]-A-....
    -A-D1[L-1]-A-D2[0]-A-D2[1]-A-....
    -A-D2[M-1]-A-P
```

Example

```
PROCEDURE Write_CCT_Memory
   (Chapter, Row, Column, Data_Buf, Data_Count);
DECLARE (Chapter, Row, Column, Data_Buf, Data_Count) BYTE;

/*
   The extended address (CCT-Cursor) is formed by Chapter, Row
   and Column. These three bytes are written after the sub-address
   (=8) followed by the actual data that will be stored relative to the
   extended address.
*/

CALL IIC_Write_Sub_Write (22h, 3, .Chapter, 8, Data_Buf,
   Data_Count);
END Write_CCT_Memory;
```

3.2.9 IIC_Write_Sub_Read

Declaration

```
IIC_Write_Sub_Read:
   PROCEDURE (Slave_Address, Count1, Source_Ptr1,
      Sub_Address, Count2, Dest_Ptr2)
      [BIT|BYTE] EXTERNAL;
   DECLARE (Slave_Address, Count1, Source_Ptr1,
      Sub_Address, Count2, Dest_Ptr2) BYTE;
   END;
```

Description

IIC_Write_Sub_Read writes a data block preceded by a sub-address, generates an I^{2}C restart condition, and reads a data block. This procedure can be used for devices that need an extended addressing method. Such a device is the ECCT.

I^{2}C Protocol

L	=Count1
M	=Count2
Sub	=Sub_Address
D1[0..L–1]	BASED by Source_Ptr1
D2[0..M–1]	BASED by Source_Ptr2

```
S-SlvW-A-Sub-A-D1[0]-A-D1[1]-A-....
    -A-D1[L-1]-A-S-SlvR-A-D2[0]-A-D2[1]-A-....
    -A-D2[M-1]-N-P
```

Example

```
PROCEDURE Read_CCT_Memory
   (Chapter, Row, Column, Data_Buf, Data_Count);
DECLARE (Chapter, Row, Column, Data_Buf, Data_Count) BYTE;

/*
   The extended address (CCT-Cursor) is formed by Chapter, Row
   and Column. These three bytes are written after the sub-address
   (8). After that the actual data will be read relative to the extended
   address.
*/

CALL IIC_Write_Sub_Write (22h, 3, .Chapter, 8, Data_Buf,
   Data_Count);
END Read_CCT_Memory;
```

3.2.10 IIC_Write_Com_Write

Declaration

```
IIC_Write_Com_Write:
    PROCEDURE (Slave_Address, Count1, Source_Ptr1, Count2,
        Source_Ptr2) [BIT|BYTE] EXTERNAL;
    DECLARE (Slave_Address, Count1, Source_Ptr1, Count2,
        Source_Ptr2) BYTE;
    END;
```

Description

IIC_Write_Com_Write writes two data blocks from different data buffers in one message to a slave receiver. This procedure can be used for devices where the message consists of 2 different data blocks. Such devices are, for instance, LCD-drivers, where the first part of the message consists of addressing and control information, and the second part is the data string to be displayed.

I^2C Protocol

L	=Count1
M	=Count2
D1[0..L–1]	BASED by Source_Ptr1
D2[0..M–1]	BASED by Source_Ptr2

```
S-SlvW-A-D1[0]-A-D1[1]-A-....
     -A-D1[L-1]-A-D2[0]-A-D2[1]-A-....
     -A-D2[M-1]-A-P
```

Example

```
DECLARE Control_Buffer(2) BYTE;
DECLARE Data_Buffer(20) BYTE;
.....
CALL IIC_Write_Com_Write(74h, LENGTH(Control_Buffer),
    .Control_Buffer, LENGTH(Data_Buffer), .Data_Buffer);
```

3.2.11 IIC_Write_Rep_Write

Declaration

```
IIC_Write_Rep_Write:
    PROCEDURE (Slave_Address1, Count1, Source_Ptr1,
        Slave_Address2, Count2, Source_Ptr2)
        [BIT|BYTE] EXTERNAL;
    DECLARE (Slave_Address1, Count1, Source_Ptr1,
        Slave_Address2, Count2, Source_Ptr2) BYTE;
    END;
```

Description

Two data strings are sent to separate slave devices, separated with a repeat START condition. This has the advantage that the bus does not have to be released with a STOP condition before the transfer from the second slave.

I^2C Protocol

L	=Count1
M	=Count2
SlvW1	=Slave_Address1
SlvW2	=Slave_Address2
D1[0..L–1]	BASED by Source_Ptr1
D2[0..M–1]	BASED by Source_Ptr2

```
S-SlvW-A-D1[0]-A-D1[1]-....
     -A-D1[L-1]-A-S-SlvW-A-D2[0]-A-D2[1]-....
     -A-D2[M-1]-A-P
```

Example

```
DECLARE Data_Buffer_1(10) BYTE;
DECLARE Data_Buffer_2(4) BYTE;
.....
CALL IIC_Write_Rep_Write (48h, LENGTH(Data_Buffer_1),
    .Data_Buffer_1, 50h, LENGTH(Data_Buffer_2), .Data_Buffer_2);
```

3.2.12 IIC_Write_Rep_Read

Declaration

```
IIC_Write_Rep_Read:
    PROCEDURE (Slave_Address1, Count1, Source_Ptr1,
        Slave_Address2, Count2, Dest_Ptr2)
        [BIT|BYTE] EXTERNAL;
    DECLARE (Slave_Address1, Count1, Source_Ptr1,
        Slave_Address2, Count2, Dest_Ptr2) BYTE;
    END;
```

Description

A data string is sent and received to/from two separate slave devices, separated with a repeat START condition. This has the advantage that the bus does not have to be released with a STOP condition before the transfer from the second slave.

I^{2}C Protocol

L	=Count1
M	=Count2
SlvW1	=Slave_Address1
SlvW2	=Slave_Address2
D1[0..L–1]	BASED by Source_Ptr1
D2[0..M–1]	BASED by Dest_Ptr2

```
S-SlvW-A-D1[0]-A-D1[1]-....
    -A-D1[L-1]-A-S-SlvR-A-D2[0]-A-D2[1]-....
    -A-D2[M-1]-N-P
```

Example

```
DECLARE Data_Buffer_1(10) BYTE;
DECLARE Data_Buffer_2(4) BYTE;
.....
CALL IIC_Write_Rep_Read (48h, LENGTH(Data_Buffer_1),
    .Data_Buffer_1, 57h, LENGTH(Data_Buffer_2), .Data_Buffer_2);
```

3.2.13 IIC_Read

Declaration

```
IIC_Read:
    PROCEDURE (Slave_Address, Count, Dest_Ptr)
        [BIT|BYTE] EXTERNAL;
    DECLARE (Slave_Address, Count, Dest_Ptr) BYTE;
    END;
```

Description

IIC_Read is the most basic procedure to read a message from a slave device.

I^{2}C Protocol

M	=Count
D2[0..M–1]	BASED by Dest_Ptr

```
S-SlvR-A-D2[0]-A-D2[1]-A.....A-D2[M-1]-N-P
```

Example

```
DECLARE Data_Buffer(4) BYTE;
.....
CALL IIC_Read (0B5, LENGTH(Data_Buffer), .Data_Buffer);
```

3.2.14 IIC_Read_Status

Declaration

```
IIC_Read_Status:
    PROCEDURE (Slave_Address, Dest_Ptr)
        [BIT|BYTE] EXTERNAL;
    DECLARE (Slave_Address, Dest_Ptr) BYTE;
    END;
```

Description

Several I^{2}C devices can send a one byte status-word via the bus. IIC_Read_Status can be used for this purpose. IIC_Read_Status works the same way as IIC_Read but the user does not have to pass a count parameter.

I^{2}C Protocol

Status	BASED by Dest_Ptr

```
S-SlvR-A-Status-N-P
```

Example

```
DECLARE Status_Byte BYTE;
.....
CALL IIC_Read_Status (84h, .Status_Byte);
```

3.2.15 IIC_Read_Sub

Declaration

```
IIC_Read_Sub:
    PROCEDURE (Slave_Address, Count, Dest_Ptr, Sub_Address)
        [BIT|BYTE] EXTERNAL;
    DECLARE (Slave_Address, Count, Dest_Ptr, Sub_Address)
        BYTE;
    END;
```

Description

IIC_Read_Sub reads a message from a slave device, preceded by a write of the sub-address. Between writing the sub-address and reading the message, an I^{2}C restart condition is generated without releasing the bus. This prevents other masters from accessing the slave device in between and overwriting the sub-address.

I^{2}C Protocol

M	=Count
Sub	=Sub_Address
D2[0..M–1]	BASED by Dest_Ptr

```
S-SlvW-A-Sub-A-S-SlvR-D2[0]-A-D2[1]-A.....A-D2[M-1]-N-P
```

Example

```
DECLARE Data_Buffer(5) BYTE;
.....
CALL IIC_Read_Sub (0A3h, LENGTH(Data_Buffer), .Data_Buffer, 2);
```

3.2.16 IIC_Read_Rep_Read

Declaration

```
IIC_Read_Rep_Read:
   PROCEDURE (Slave_Address1, Count1, Dest_Ptr1,
      Slave_Address2, Count2, Dest_Ptr2)
      [BIT|BYTE] EXTERNAL;
   DECLARE (Slave_Address1, Count1, Dest_Ptr1,
      Slave_Address2, Count2, Dest_Ptr2) BYTE;
   END;
```

Description

Two data strings are read from separate slave device, separated with a repeat START condition. This has the advantage that the bus does not have to be released with a STOP condition before the transfer from the second slave.

I²C Protocol

L	=Count1
M	=Count2
SlvW1	=Slave_Address1
SlvW2	=Slave_Address2
D1[0..L–1]	BASED by Dest_Ptr1
D2[0..M–1]	BASED by Dest_Ptr2

```
S-SlvR-A-D1[0]-A-D1[1]-.....
    -A-D1[L-1]-N-S-SlvR-A-D2[0]-A-D2[1]-.....
    -A-D2[M-1]-N-P
```

Example

```
DECLARE Data_Buffer_1(10) BYTE;
DECLARE Data_Buffer_2(4) BYTE;
.....
CALL IIC_Read_Rep_Read (49h, LENGTH(Data_Buffer_1),
   .Data_Buffer_1, 51h,
LENGTH(Data_Buffer_2), .Data_Buffer_2);
```

3.2.17 IIC_Read_Rep_Write

Declaration

```
IIC_Read_Rep_Write:
   PROCEDURE (Slave_Address1, Count1, Dest_Ptr1,
   Slave_Address2, Count2, Source_Ptr2)
      [BIT|BYTE] EXTERNAL;
   DECLARE (Slave_Address1, Count1, Dest_Ptr1,
      Slave_Address2, Count2, Source_Ptr2) BYTE;
   END;
```

Description

A data string is received and sent from/to two separate slave devices, separated with a repeat START condition. This has the advantage that the bus does not have to be released with a STOP condition before the transfer from the second slave.

I²C Protocol

L	=Count1
M	=Count2
SlvW1	=Slave_Address1
SlvW2	=Slave_Address2
D1[0..L–1]	BASED by Dest_Ptr1
D2[0..M–1]	BASED by Source_Ptr2

```
S-SlvR-A-D1[0]-A-D1[1]-.....
    -A-D1[L-1]-N-S-SlvW-A-D2[0]-A-D2[1]-.....
    -A-D2[M-1]-A-P
```

Example

```
DECLARE Data_Buffer_1(10) BYTE;
DECLARE Data_Buffer_2(4) BYTE;
.....
CALL IIC_Read_Rep_Write(49h, LENGTH(Data_Buffer_1),
   .Data_Buffer_1, 58h,
LENGTH(Data_Buffer_2), .Data_Buffer_2);
```

3.2.18 Slave Mode Routines

There are two ways for the I^{2}C interface to enter the slave-mode:

– After an I^{2}C interrupt the software must enter the slave-receiver mode to receive the slave address. This address will then be compared with its own address. If there is a match either slave-transmitter or slave-receiver mode will be entered. If no match occurs, the interrupted program will be continued.

– During transmission of a slave-address in master-mode, arbitration is lost to another master. The interface must then switch to slave-receiver mode to check if this other master wants to address the 8xC528 interface.

The slave-mode protocol is very application dependent. In this note the basic slave-receive and slave-transmit routines are given and should be considered as examples. The user may for instance send NO_ACK after receiving a number of bytes to signal to the master-transmitter that a data buffer is full. A description of the code will be given later.

Slave parameters are given with the Init_IIC procedure. The passed parameters are the own-slave-address and a source/destination-pointer to a data buffer.

The slave-routine will be suspended at the following conditions:

– Interrupts with higher priority. Slave-routine will be resumed again after interrupt is handled.

– If a NO_ACKNOWLEDGE is received form a master-receiver.

– If a STOP condition is detected from a master transmitter.

Constraints for user software.

– The user must control the global enable (EA) bit.

– The user must control the priority level of the I^{2}C interrupt. If the slave routine is interrupted by a higher priority interrupt, the SCL line will be stretched to postpone bus transfer until the higher interrupt is finished.

3.3 The Slave Routine: SLAVE.ASM

The listing of the slave routine can be seen on page 12. The routine is written in such a way that stretching of SCL is minimized. Application code can be inserted in this routine and this will increase stretching time.

The routine has 2 entry points.

Entry via MST_ENTRY happens when an arbitration error has occurred when transmitting a slave address in master mode. Auto-clock generation will be disabled and SCL stretching enabled. The byte will be continued to be received and can later be compared with the own slave address.

The second entry point is via an interrupt when a START condition is detected. At _PIP0A the context of the interrupted program is stored. Next Auto-clock generation is disabled and SCL stretching enabled. Reception of the slave address can now begin by calling RCV_SL_BY. When the received slave-address is compared with the own-slave-address the R/W-bit is ignored. If there is no match between the 2 addresses, a negative ACK bit is sent and the slave routine is left via EXIT. If there was a match the R/W bit is checked to enter the slave-receiver or slave-transmitter mode.

The slave-transmitter mode starts at NXT_TRX. After getting the byte from the data buffer via BUF_POINT and initializing the bit counter BIT_CNT the transmission loop is entered. A bit is written via access to S1BIT because this will automatically reset the CLH and WBF status flags, and also SCL stretching. Now WBF must be tested until the transmission is successful. When WBF becomes true, SCL will be stretched again. When 8 bits are sent, the SDA line is released and RBF is tested until the ACK bit is received. The ACK bit is read by reading SDI instead of S1BIT to maintain SCL stretching. If ACK was false, no more bytes have to be sent and the routine is left. If another byte has to be transmitted, BUF_POINT is updated and transmission will continue.

The slave-receiver mode starts at RCV_SLAVE. A byte is received by calling RCV_SL_BY. This routine will clear the CY-flag when a STOP condition has been received. This means that the master will send no more bytes to this slave and the slave routine will be left. When no STOP condition was detected, the received byte will be stored @BUF_POINT and an ACK bit will be sent. After this, a new byte can be received.

When calling RCV_SL_BY the bit counter BIT_CNT will be initialized and the SCL stretching stopped by a dummy access to S1BIT. In the receive loop both BB and RBF will be checked. When BB is cleared, a STOP condition is detected and the routine will be left with CY=0.

The first 7 bits are received via S1BIT because this will release stretching. The 8th bit is accessed via SDI because stretching must be maintained.

If the slave routine is left via EXIT, the STR bit is cleared (to disable stretching on SCL edges when the 8xC528 is not addressed as slave) and a dummy access to S1BIT is done to finish current SCL stretching. If the slave routine was entered via an interrupt the previous context is restored.

```
TSW–ASM51 V3.0b Serial #00052252  Slave interrupt routine
PAGE   1


LOC   OBJ        LINE  SOURCE


                    1  $TITLE(Slave interrupt routine)
                    2  $DEBUG
                    3  $NOLIST
                    6  ;
                    7  ;This routine handles I2C interrupts.
                    8  ;8xC528 I2C interface enters in slave mode.
                    9  ;After testing R/W bit, 8xC528 will go in slave–transmit or
                   10  ;slave–receive mode.
                   11  ;Source or destination buffer for data uses pointer SLAVE_SUB_ADDRESS
                   12  ;Slave routine will use register bank 01
                   13  ;
                   14  ;********************************************************************
                   15  ;Interrupt entry point
                   16
––––               17                 CSEG AT 53H
                   18
0053: 020000   R   19                 LJMP __PIP0A    ;Vector to interrupt handler
                   20  ;
                   21  ;********************************************************************
                   22
                   23                 I2C_DRIVER SEGMENT CODE INBLOCK
––––               24                 RSEG I2C_DRIVER
                   25
                   26                 PUBLIC MST_ENTRY
                   27                 EXTRN  DATA(SLAVE_SUB_ADDRESS)
                   28                 EXTRN  BIT(ARB_LOST)
                   29
REG END            30                 BUF_POINT SET R0
REG END            31                 OWN_SLAVE SET R7
REG END            32                 BIT_CNT   SET R2
                   33
                   34  ;********************************************************************
                   35
0000: C0E0     R   36  __PIP0A:PUSH ACC        ;Push CPU status on stack
0002: C0D0         37          PUSH PSW
0004: 75D008       38          MOV PSW,#08H    ;Select registerbank 01
                   39
                   40  ;********************************************************************
                   41  ;Check slave address
                   42  ;********************************************************************
                   43
0007: 43D842       44          ORL S1SCS,#01000010B ;Disable SCL generation and enable SCL
                                                ;stretching stretching
000A: 1142     R   45          ACALL RCV_SL_BY ;Receive slave address, on exit SCL is
                                                ; stretched
000C: A2E0         46  PROC:   MOV C,ACC.0     ;Store R/W bit in F0
000E; 92D5         47          MOV F0,C
0010: 6F           48          XRL A,OWN_SLAVE ;Compare received slave address
```

```
TSW–ASM51 V3.0b Serial #00052252  Slave interrupt routine
PAGE   2


LOC   OBJ         LINE  SOURCE


0011: C2E0          49          CLR ACC.0       ;Ignore R/W bit
0013: 7050          50          JNZ NO_MATCH    ;Leave slave–routine if there is no match
0015: C3            51          CLR C           ;Send ACK
0016: 115C     R    52          ACALL SEND_ACK
0018: A800     R    53          MOV BUF_POINT,SLAVE_SUB_ADDRESS  ;Get buffer pointer
001A: A2D5          54          MOV C,F0        ;Restore R/W bit
001C: 5019          55          JNC RCV_SLAVE   ;Test R/W bit
                    56
                    57
                    58  ;*********************************************************************
                    59  ;Slave transmitter mode
                    60  ;*********************************************************************
                    61
                    62
001E: E6            63  NXT_TRX:MOV A,@BUF_POINT;Get byte to send
001F: 7A08          64          MOV BIT_CNT,#08 ;Init bit counter
                    65
0021:               66  NXT_TRX_BIT:
0021: F5D9          67          MOV S1BIT,A     ;Trx bit and stretch after transmission
0023: 23            68          RL A            ;Prepare next bit to send
0024: 30DAFD        69          JNB WBF,$       ;Test if bit is sent
0027: DAF8          70          DJNZ BIT_CNT,NXT_TRX_BIT ;Test if all bits are sent
                    71
0029: D2DF          72          SETB SDO        ;Release SDA line for NO_ACK/ACK reception
002B: E5D9          73          MOV A,S1BIT     ;Stop stretching
002D: 30DBFD        74          JNB RBF,$       ;Test if ACK bit is received
0030: A2DF          75          MOV C,SDI       ;Read bit, SCL remains stretched
0032: 4040          76          JC EXIT         ;NO_ACK received. Exit slave routine
0034: 08            77          INC BUF_POINT   ;ACK received. Update pointer for next byte to
                                                ;send
0035: 80E7          78          SJMP NXT_TRX
                    79
                    80  ;*********************************************************************
                    81  ;Slave receiver mode
                    82  ;*********************************************************************
                    83
0037:               84  RCV_SLAVE:               ;Entry in slave–receiver mode
0037: 1142     R    85          ACALL RCV_SL_BY  ;Receive byte
0039: 5039          86          JNC EXIT         ;If STOP is detected, then exit
003B: F6            87          MOV @BUF_POINT,A ;Store received byte
003C: C3            88          CLR C            ;Send ACK
003D: 115C     R    89          CALL SEND_ACK
003F: 08            90          INC BUF_POINT    ;Update pointer
0040: 80F5          91          SJMP RCV_SLAVE   ;Receive next byte
                    92
                    93
```

```
TSW–ASM51 V3.0b Serial #00052252  Slave interrupt routine
PAGE   3


LOC   OBJ         LINE  SOURCE


                   94  ;*******************************************************************
                   95  ;Receive byte routine
                   96  ;On exit, received byte in accu
                   97  ;On exit CY=0 if STOP is detected
                   98  ;*******************************************************************
0042:              99  RCV_SL_BY:
0042: 7A08        100          MOV BIT_CNT,#08
0044: E5D9        101          MOV A,S1BIT     ;Disable stretching from START or previous ACK
0046: E4          102          CLR A
0047:             103  RCV_BIT:
0047: 30DC10      104          JNB BB,STOP_RCV ;Test if STOP–condition is received
004A: 30DBFA      105          JNB RBF,RCV_BIT ;Wait till received bit is valid
004D: BA0105      106          CJNE BIT_CNT,#01,ASSEM_BIT ;Check if last bit is to be received
                  107
0050: A2DF        108          MOV C,SDI       ;Get last bit without stopping stretching
0052: 33          109          RLC A
0053: D3          110          SETB C          ;No STOP detected
0054: 22          111          RET
                  112
0055:             113  ASSEM_BIT:
0055: 45D9        114          ORL A,S1BIT     ;Receive bit; release RBF,CLH and SCL stretching
0057: 23          115          RL A
0058: DAED        116          DJNZ BIT_CNT,RCV_BIT
                  117
005A:             118  STOP_RCV:
005A: C3          119          CLR C           ;STOP detected
005B: 22          120          RET
                  121
                  122  ;*******************************************************************
                  123  ;Send ACK/NO_ACK. Value of ACK in Carry
                  124  ;*******************************************************************
005C:             125  SEND_ACK:
005C: 13          126          RRC A
005D: F5D9        127          MOV S1BIT,A     ;Carry to SDA line
005F: 30DAFD      128          JNB WBF,$       ;Test if ACK/NO_ACK is sent
0062: D3DF        129          SETB SDO        ;Release SDA line
0064: 22          130          RET
                  131
                  132  ;*******************************************************************
                  133  ;No match between received slave–address and own–slave–address
                  134  ;*******************************************************************
0065:             135  NO_MATCH:
0065: D3          136          SETB C          ;Send NO_ACK
0066: 115C    R   137          ACALL SEND_ACK
0068: 800A        138          SJMP EXIT
```

```
TSW–ASM51 V3.0b Serial #00052252  Slave interrupt routine
PAGE   4

LOC   OBJ        LINE  SOURCE

                  139  ;*******************************************************************
                  141  ;Entry point when an arbitration–lost condition is detected in
                       ;master–mode.
                  142  ;*******************************************************************
006A:             143  MST_ENTRY:
006A: 23          144          RL A            ;Restore slave address  sofar
006B: C2E0        145          CLR ACC.0
006D: 43D842      146          ORL S1SCS,#01000010B ;Disable SCL generation and enable SCL
                                               ;strectching
0070: 1147    R   147          ACALL RCV_BIT   ;Proceed with receiving rest of slave address
0072: 8098        148          SJMP PROC
                  149
                  150
                  151  ;*******************************************************************
                  152  ;Exit from interrupt routine
                  153  ;*******************************************************************
                  154
0074: C2D9        155  EXIT:   CLR STR         ;Disable stretching on next falling SCL edges
0076: E5D9        156          MOV A,S1BIT     ;Stop current SCL stretching
0078: 30001   R   157          JNB ARB_LOST,EX_SL
007B: 22          158          RET             ;Exit when entered from master mode
007C: D0D0        159  EX_SL:  POP PSW         ;Restore old CPU status
007E: D0E0        160          POP ACC
0080: 32          161          RETI
                  162
0081:             163          END
```

4.0 EXAMPLES

4.1 Introduction

Some examples are given how to use the I^{2}C routines in an applicatIon program. Examples are given for assembly, PL/M and C program.

The program displays time from the PCF8583P clock/calendar/RAM on an LCD display driven by the PCF8577.

The example can be executed on the OM4151 I^{2}C evaluation board.

4.2 Using the Routines with Assembly Sources

The listing of the example program is shown on page 17. The most important aspect when using the I^{2}C routines is preparing the input parameters before the sub-routine call. When, for example, the IIC_Write routine must be called, the parameters must be called in the following order:

```
MOV _IIC_READ_BYTE,#SLAVE_ADR
MOV _IIC_READ_BYTE+1,#COUNT_1
MOV _IIC_READ_BYTE+2,#SOURCE_PTR_1
CALL _IIC_READ
```

Note that the order of defining the parameters is the same as in a PL/M-call. An easier way to call the routines is making a macro that includes the initializing of the parameters. The example program makes use of macros.

IIC_Read is then called in the following way:

```
%IIC_Read(Slave_Adr,Count_1,Source_Ptr_1);
```

Note that in the listing the contents of the macro are shown, instead of the call.

The macro must be written as follows:

```
%*  DEFINE
    (IIC_Read(SLAVE_ADR,COUNT_1,SOURCE_PTR_1))
    (MOV _IIC_READ_BYTE,#%SLAVE_ADR
    MOV _IIC_READ_BYTE+1,#%COUNT_1
    MOV _IIC_READ_BYTE+2,#%SOURCE_PTR_1
    LCALL _IIC_READ)
```

Macros for the I^{2}C CALLs are found in I2C.MAC. This file should be included in all modules making use of the macros. One of the modules should also include the variable definitions needed by the I^{2}C routines. These are found in file VAR_DEF.ASM. If the program consists of more than 1 module, then these modules should also include EXT_VAR.ASM. This file contains the EXTRN- definitions of the I^{2}C routines.

When and I^{2}C routine is called, the accumulator contains status information and the CY-bit is set if an error has occurred. The contents of the accumulator are the same as the returned byte when using PL/M.

```
TSW–ASM51 V3.0b Serial #00052252  Assembly example program                       PAGE   1


LOC   OBJ        LINE  SOURCE


                    1  $TITLE(Assembly example program)
                    2  $DEBUG
                    3
                    4  ;Hours and minutes will be displayed on LCD display
                    5  ;Dot between hours and minutes will blink
                    6
                    7  $                                       ;Include I2C var. definitions
                    8  # 1 ”C:\USER\VAR_DEF.ASM”
                   74  # 8 ”DEMO_ASM.ASM”
                    8  $                                       ;Include I2C macro’s
                    9  # 1 ”C:\USER\I2C.MAC”
                   35  # 9 ”DEMO_ASM.ASM”
00A2               10  CLOCK_ADR   EQU 0A2h                    ;Address of PCF8583
0001               11  CL_SUB_ADR  EQU 01h                     ;Sub address for reading time
0074               12  LCD_ADR     EQU 74h                     ;Address of PCF8577
                   13
                   14  RAMVAR  SEGMENT DATA                    ;Segment for variables
                   15  USER    SEGMENT DATA                    ;Segment for application
                                                               ;program
                   16
––––               17          RSEC RAMVAR
0000:          R   18  STACK:      DS 10                       ;Stack area
000A:              19  TIME_BUFFER:DS 4                        ;Buffer for I2C strings
000E:              20  LCD_BUFFER: DS 5
                   21
––––               22          CSEG AT 00
0000: 020000   R   23          LJMP APL_START
                   24
                   25
––––               26          RSEG USER
                   27
0000:          R   28  APL_START:
0000: 900073   R   29          MOV DPTR,#LCD_TAB               ;Pointer to segment table
0003: 7581FF   R   30          MOV SP,#STACK–1                 ;Initialize stack
0006: 750E00   R   31          MOV LCD_BUFFER,#00              ;Control word for LCD driver
                   32
0009: 750022   R   33      MOV _Init_IIC_Byte  ,#22h
000C: 75010A   R   34      MOV _Init_IIC_Byte+1,#TIME_BUFFER
000F: 120000   R   35      LCALL _Init_IIC
                   36            ;Initialize I2C interface
0012: E4           37          CLR A                           ;Prepare buffer for clock int.
0013: F50A     R   38          MOV TIME_BUFFER,A
0015: F50B     R   39          MOV TIME_BUFFER+1,A
                   40
0017: 7500A2   R   41      MOV _IIC_Write_Byte  ,#CLOCK_ADR
001A: 750102   R   42      MOV _IIC_Write_Byte+1,#2
001D: 75020A   R   43      MOV _IIC_Write_Byte+2,#TIME_BUFFER
0020: 120000   R   44      LCALL _IIC_Write
```

```
TSW–ASM51 V3.0b Serial #00052252  Assembly example program                       PAGE   2


LOC   OBJ        LINE  SOURCE


                   45      ;Initialize clock
                   46
0023:              47  REPEAT:
0023: 7500A2   R   48      MOV _IIC_Read_Sub_Byte  ,#CLOCK_ADR
0026: 750104   R   49      MOV _IIC_Read_Sub_Byte+1,#4
0029: 75020A   R   50      MOV _IIC_Read_Sub_Byte+2,#TIME_BUFFER
002C: 750301   R   51      MOV _IIC_Read_Sub_Byte+3,#CL_SUB_ADR
002F: 120000   R   52      LCALL _IIC_Read_Sub
                   53       ;Read time
                   54
                   55  ;Time has been read. Order: hundreds of sec’s, sec’s, min’s and hr’s
0032: E50D     R   56          MOV A,TIME_BUFFER+3             ;Mask of hour counter
0034: 543F         57          ANL A,#3Fh
0036: F50D     R   58          MOV TIME_BUFFER+3,A
                   59
0038: 120054   R   60          CALL CONVERT                    ;Convert time data to LCD
                                                               ;segment data
                   61
                   62  ;Check if dot has to be switched on
003B: 431101   R   63          ORL LCD_BUFFER+3,#01h
                   64  ;If lsb of seconds is ’0’, then switch on dp
003E: E50B     R   65          MOV A,TIME_BUFFER+1              ;Get seconds
0040: 13           66          RRC A
0041: 4003         67          JC PROCEED
0043: 430F01   R   68          ORL LCD_BUFFER+1,#01             ;Switch on dp
                   69
                   70  ;Display new time
0046:              71  PROCEED:
0046: 750074   R   72      MOV _IIC_Write_Byte  ,#LCD_ADR
0049: 750105   R   73      MOV _IIC_Write_Byte+1,#5
004C: 75020E   R   74      MOV _IIC_Write_Byte+2,#LCD_BUFFER
004F: 120000   R   75      LCALL _IIC_Write
                   76
0052: 80CF         77          SJMP REPEAT                      ;Read new time
                   78
                   79
                   80  ;CONVERT converts BCD data of time to segment data
0054: 780F     R   81  CONVERT:MOV R0,#LCD_BUFFER+1             ;R0 is pointer
0056: E50D     R   82          MOV A,TIME_BUFFER+3              ;Get hours
0058: C4           83          SWAP A                           ;Swap nibbles
0059: 12006D   R   84          CALL LCD_DATA                    ;Convert 10’s of hours
005C: E50D     R   85          MOV A,TIME_BUFFER+3
005E: 12006D   R   86          CALL LCD_DATA                    ;Convert hours
0061: E50C     R   87          MOV A,TIME_BUFFER+2              ;Get minutes
0063: C4           88          SWAP A
0064: 12006D   R   89          CALL LCD_DATA                    ;Convert 10’s of minutes
0067: E50C     R   90          MOV A,TIME_BUFFER+2
0069: 12006D   R   91          CALL LCD_DATA                    ;Convert minutes
```

```
TSW–ASM51 V3.0b Serial #00052252  Assembly example program                       PAGE   3

LOC   OBJ        LINE  SOURCE

006C: 22           92           RET
                   93
                   94  ;LCD_DATA gets data from segment table and stores it in LCD_BUFFER
006D:              95  LCD_DATA:
006D: 540F         96         ANL A,#0FH                      ;Mask off LS–nibble
006F: 93           97         MOVC A,@A+DPTR                  ;Get segment data
0070: F6           98         MOV @R0,A                       ;Save segment data
0071: 08           99         INC R0
0072: 22          100         RET
                  101  ;
                  102  ;Conversion table for LCD
0073:             103  LCD_TAB:
0073: FC60DA      104         DB 0FCH,60H,0DAH                ;’0’,’1’,’2’
0076: F266B6      105         DB 0F2H,66H,0B6H                ;’3’,’4’,’5’
0079: 3EE0FE      106         DB 3EH,0E0H,0FEH                ;’6’,’7’,’8’
007C: E6          107         DB 0E6H                         ;’9’
                  108  ;
007D:             109         END
```

4.3 Using the Routines with PL/M-51 Sources

The following listing shows the listing of the clock program in PL/M-51. The procedures are untyped. The routines are used the same way as in the examples of chapter 3.2.

```
$OPTIMIZE(4)
$DEBUG
$CODE

/* Hours and minutes will be displayed on LCD display
   Dots between hours and minutes will blink */

Demo_plm: Do;

/* External declarations */

Init_IIC: Procedure(Own_Adr,Slave_Ptr) External;
          Declare (Own_Adr,Slave_Ptr) Byte Main;
End Init_IIC;

IIC_Write: Procedure(Sl_Adr,Nr_Bytes,Source_Ptr) External;
           Declare (Sl_Adr,Nr_Bytes,Source_Ptr) Byte Main;
End IIC_Write;

IIC_Read_Sub: Procedure(Sl_Adr,Nr_Bytes,Dest_Ptr,Sub_Adr) External;
              Declare(Sl_Adr,Nr_Bytes,Dest_Ptr,Sub_Adr) Byte Main;
End IIC_Read_Sub;
```

```
Clock: Do;
          /* Variable and constant declarations */

          Declare LCD_TAB(*) Byte Constant (0FCh,60H,0DAH,0F2H,66H,
                                           0B6H,3EH,0E0H,0FEH,0E6H);
          Declare Time_Buffer(4) Byte Main;
          Declare LCD_Buffer(5) Byte Main;
          Declare Tab_Point Word Main;
          Declare (LCD_Point,Time_Point) Byte Main;
          Declare Segment Based LCD_Point Byte Main;
          Declare Time Based Time_Point Byte Main;
          Declare Tab_Value Based Tab_Point Byte Constant;

          Declare clock_Adr Literally '0A2h';
          Declare LCD_Adr Literally '74h';
          Declare C1_Sub_Adr Literally '01h';

          Call Init_IIC(22h,.Time_Buffer);
          LCD_Buffer(0)=0; /* LCD control word */
          Time_Buffer(0)=0;
          Time_Buffer(1)=0;
          Call IIC_Write(Clock_Adr,2,.Time_Buffer); /* Initialize clock */

          Do While LCD_Buffer(0)=0; /* Program loop */
             Call IIC_Read_Sub(Clock_Adr,4,.Time_Buffer,Cl_Sub_Adr);
                                                       /* Get time */
             LCD_Point=.LCD_Buffer+1; /* Initialize pointers */
             Time_point=.Time_Buffer(3);
             Tab_Point=.LCD_Tab(0)+SHR(Time,4); /* 10–HR's */
             Segment=Tab_Value;

             LCD_Point=LCD_Point+1;
             Tab_Point=.LCD_Tab(0)+(Time AND 0FH); /* HR's */
             Segment=Tab_Value;
             Time_Point=Time_Point–1;
             LCD_Point=LCD_Point+1;
             Tab_Point=.LCD_Tab+SHR(Time,4); /* 10–MIN's */
             Segment=(Tab_Value OR 01H); /* dp */
             LCD_Point=LCD_Point+1;
             Tab_Point=.LCD_Tab+(Time AND 0FH); /* MIN's */
             Segment=Tab_Value;
             Time_Point=.Time_Buffer(1)+1; /*Check sec's for blinking */
             LCD_Point=.LCD_Buffer+1;
             If (Time AND 01H)>0 then Segment=(Segment OR 01H);
             Call IIC_Write(LCD_Adr,5,.LCD_Buffer); /* Display time */
          End;

       End Clock;

End Demo_plm;
```

4.4 Using the Routines with C Sources

An example of a C program using the I^{2}C routines follows. Function prototypes are found in header file “i2c.h”. In this example the function prototypes are written in such a way that not value is returned by the function. If the STATUS byte is needed, the header file may be changed to return a byte. Note that the function calls are written in upper-case. This is due to the fact that the used version of the assembler/linker is case sensitive.

```
#include <C:\USER\i2c.h>

         rom  char              LCD_Tab[]={0xFC,0x60,0xDA,0xF2,0x66,0xB6,0x3E,
                                           0xE0,0xFE,0xE6};

void main()

{

        #define Clock_Adr       0xA2
        #define LCD_Adr         0x74
        #define C1_Sub_Adr      0x01

        rom  char               * Tab_Ptr;
        data char               Time_Buffer[4];
        data char               * Time_Ptr;
        data char               LCD_Buffer[5];
        data char               * LCD_Ptr;

        INIT_IIC(0x22,&Time_Buffer);
        LCD_Buffer[0]=0; /* LCD control word */
        Time_Buffer[0]=0;
        Time_Buffer[1]=0;
        IIC_WRITE(Clock_Adr,2,&Time_Buffer); /* Initialize clock */

        while (1)      /* Program loop */
              {
               IIC_READ_SUB(Clock_Adr,4,&Time_Buffer,C1_Sub_Adr);
                                                    /* Get time */
               LCD_Ptr = &LCD_Buffer[1]; /* Initialize pointers */
               Time_Ptr = &Time_Buffer[3];
               Tab_Ptr = (LCD_Tab+(*Time_Ptr >> 4)); /* 10–HR’s */
               *(LCD_Ptr++) = *Tab_Ptr;
               Tab_Ptr = (LCD_Tab+(*(Time_Ptr––) & 0x0F)); /* HR’s */
               *(LCD_Ptr++) = *Tab_Ptr;
               Tab_Ptr = (LCD_Tab+(*Time_Ptr >> 4)); /* 10–MIN’s */
               *(LCD_Ptr++) = (*Tab_Ptr | 0x01); /* dp */
               Tab_Ptr = (LCD_Tab+(*Time_Ptr & 0x0F)); /* MIN’s */
               *LCD_Ptr = *Tab_Ptr;
               Time_Ptr = &Time_Buffer[1]; /* Check sec’s for blinking */
               LCD_Ptr = &LCD_Buffer[1];
               if ((*Time_Ptr & 0x01)>0)
                  *LCD_Ptr = (*LCD_Ptr | 0x01);
               IIC_WRITE(LCD_Adr,5,&LCD_Buffer); /* Display time */
              }
}
```

5.0 CONTENTS OF DISK

A disk contains the following 3 directories:

1: \USER

This director contains the files that may be used in the user program.

I2C_DR.LIB	Library with I^{2}C routines.
I2C.H	Header file for C applications.
I2C.MAC	Macro's for the I^{2}C routine calls in assembly programs.
VAR_DEF.ASM	Include file with variable definitions for assembly programs.
EXT_VAR.ASM	Include file with external definitions for assembly programs.
LIB.BAT	Example batch file to create I2C_DR.LIB.
ASM.BAT	Example batch file to assemble source modules for library.

2: \EXAMPLE

This directory contains the source files of the examples described in chapter 4.0.

DEMO_ASM.*	Assembly example.
DEMO_PLM.*	PL/M example.
HEAD_51.SRC	Example of environment file for PL/M example.
DEMO_C.*	C example.
CSTART.ASM	Example of environment file for C example.

3: \SOURCE

This directory contains the source files of the modules in the library.