SGS-THOMSON MICROELECTRONICS

APPLICATION NOTE

USING THE I^2C-bus PROTOCOL WITH THE ST9

Myriam Chabaud and Alan Dunworth

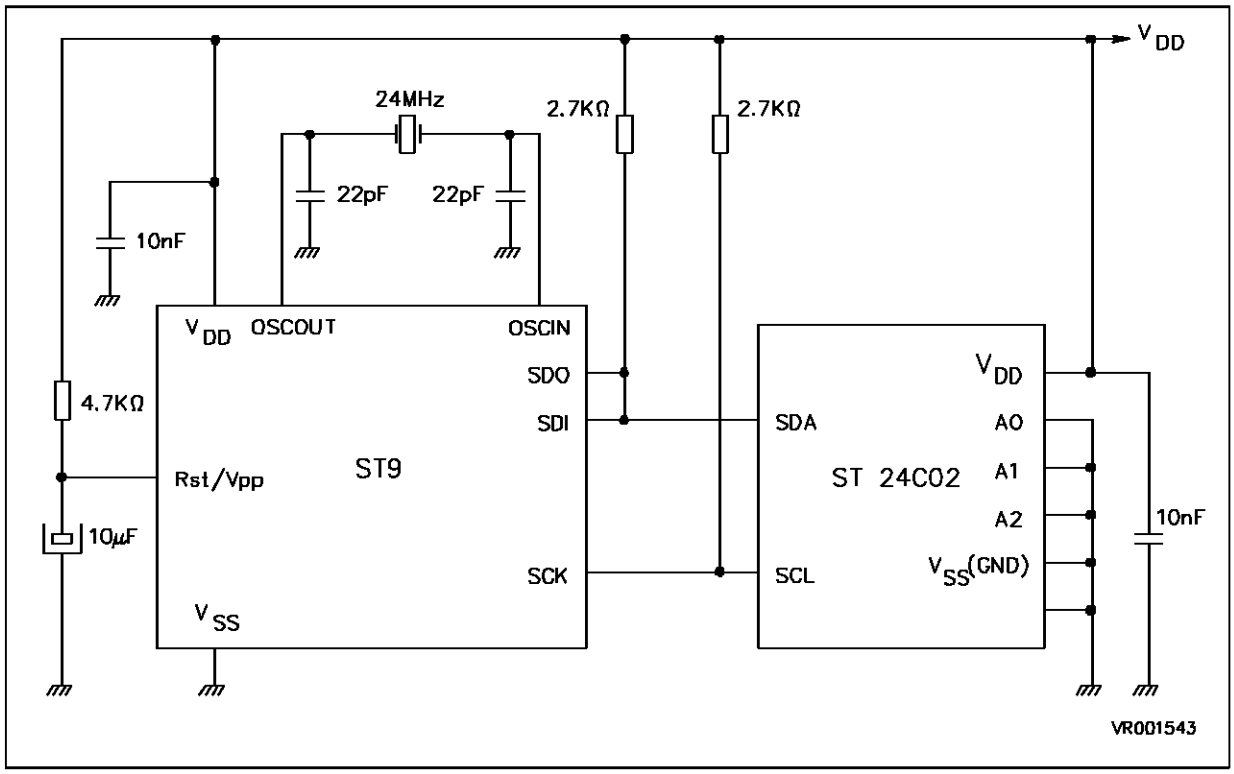
INTRODUCTION

The Serial Peripheral Interface (SPI) in the ST9 has been designed to handle a wide variety of serial bus protocols, including SBUS, IMBUS, and I^2C-bus. Certain standard I^2C-bus features have not been directly implemented in hardware, but may be realized with simple software routines, based on the SPI,contained in the standard ST9 core. This Application note gives an example of such routines, suitable for interfacing the ST9 with a serial memory device.

CHARACTERISTICS OF THE I^2C-bus

The **I^2C-bus** comprises two bidirectional lines, one for data signals (SDA) and one for clock signals (SCK). Both the SDA and the SCK lines must be connected to the positive supply via pull-up resistors (Figure 1).

Figure 1. Connection of ST24C02 and ST9 in I^2C-bus



Note: Although the ST24C02 2K bit EPROM is shown, this circuit will work with serial EEPROMS up to 16 K bit capacity (ST24C16) and all others in the ST24Cxx and ST25Cxx families.

The following basic definitions are applied:

* MASTER:

The device which initiates the transfer, generates the clock signals, and terminates the transfer is referred to as the Master. In our present application the ST9 always acts as the Master.

* SLAVE:

This is the device addressed by the Master (always the serial memory).

* TRANSMITTER:

This is the device which sends data to the bus. In our application the ST9 acts as Transmitter when it is writing data in the serial memory. Conversely, the serial memory serves as Transmitter when the ST9 is reading data from memory.

* RECEIVER:

This is the device which receives data from the bus. In our application this will be the ST9 when reading data, or the serial memory when the ST9 commands a write operation.

The following protocol has been defined:

* DATA TRANSFER

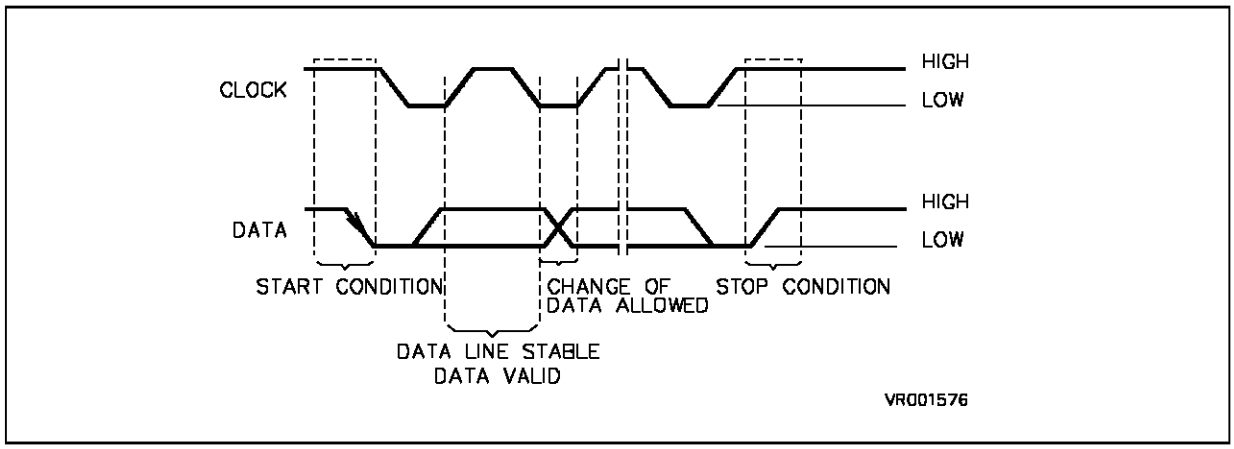
A data transfer may be initiated only when the bus is not busy.

* DATA LINE STABLE:

During data transfer, the data line must remain stable whenever the clock line is HIGH. Changes in the data line while the clock is HIGH will be interpreted as control signals.

Accordingly, the following bus conditions have been defined:

Figure 2. Data Transfer Sequence of the Serial Bus



* START DATA TRANSFER:

A change in the state of the data line from HIGH to LOW, while the clock is HIGH, defines the START condition.

* STOP DATA TRANSFER:

A change in the state of the data line from LOW to HIGH, while the clock is HIGH, defines the STOP condition.

* DATA VALID:

The state of the data line represents valid data when, after a START condition, the data line is stable for the duration of the HIGH period of the clock signal. The data on SDA may be changed during the LOW period of the clock signal. There is one clock pulse for each bit of data.

* DATA TRANSFER:

Each data transfer is initiated with a START condition and terminated with a STOP condition. The number of data bytes, transferred between the START and STOP conditions, is limited to eight bytes in the ST24C02 Memory device ERASE + WRITE mode, and is not limited in the READ mode.

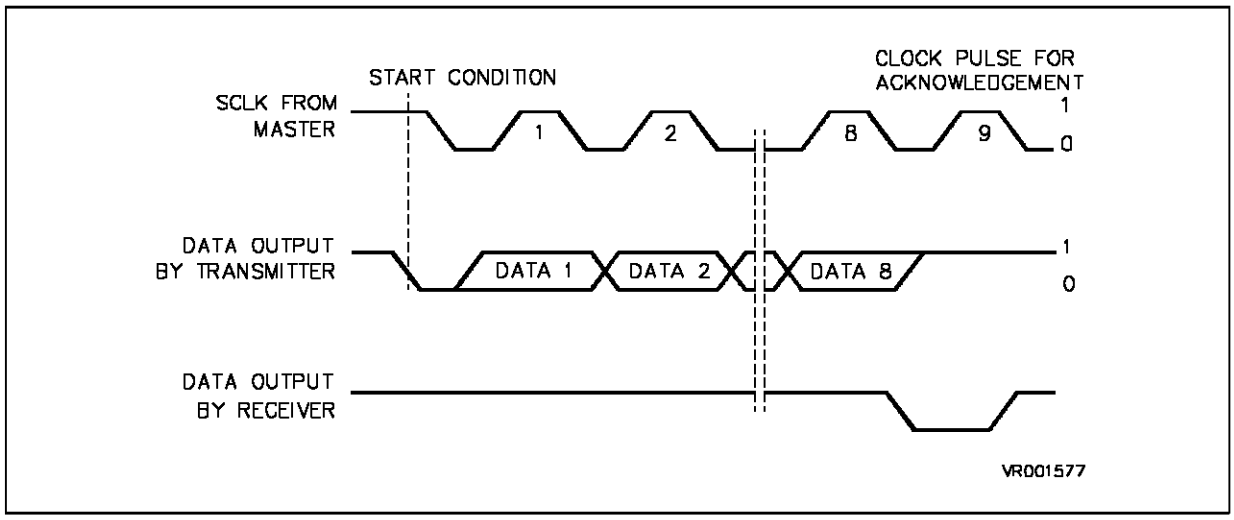
* ACKNOWLEDGE:

Each byte of eight bits is followed by an acknowledge bit. This acknowledge bit is a low level put on the SDA line by the Receiver. At the same time the Transmitter releases the SDA line to the High impedance state, and the MASTER device generates an additional 9th acknowledge-related clock pulse.

The receiving device acknowledges the receipt of the 8-bit byte by pulling the SDA down so that is stable LOW during the 9th clock pulse. Of course, set-up and hold-times must be respected.

The ST9 when acting as a Master Receive device, i.e. during serial memory READ operations, must signal an end of data by not generating an acknowledge on the last byte that has been clocked out of the slave. In this case the serial memory must leave the SDA line high to enable the Master to generate a STOP condition.

Figure 3. Acknowledgement and the 9th Clock Pulse



BASIC SOFTWARE OPERATIONS

The following aspects of the I²C-bus protocol have not been directly implemented but must be simulated in software.

* Generation of START Conditions,
* Generation of STOP Conditions,
* Generation of the Acknowledge pulse (9th clock signal),
* Generation of the Acknowledge , when the ST9 acts as a Receiver, i.e. in READ mode.
* Test of the Acknowledge from the receiver, when the ST9 acts as a Transmitter, i.e. in WRITE mode.

In order to implement these features it is necessary to drive SDA and/or SCK HIGH or LOW in the correct timing sequence.

The SDO and SCK signals are defined as Alternate Functions. These pins are configured with Open-Drain outputs and TTL inputs. The SDI signal is defined as an INPUT.

The SPI unit is enabled or disabled using the flag SPEN, bit 7 in SPICR, the SPI Control Register.

When the SPI is disabled, both SCK and SDO are released to the High impedance state. The presence of Pull-up resistors, as shown in Figure 1, effectively defines both SCK and SDA as HIGH, whenever the SPI is disabled. Note however that SDA may be driven low either by the actions of peripherals connected to the SDA line, or by appropriate action of the ST9 on the SDO line when it is defined as a normal output.

When the SPI is enabled (SPEN = “1”), it may be in either an active or passive state. The active state is entered by loading a byte of data into the SPI Data register. This automatically causes the SPI to generate a sequence of 8 clock pulses, during which data is shifted out on the SDO line, and input Data on the SDI input is clocked into the Serial input register. On completing this sequence the SPI will revert to its passive (Rest) mode.

When the SPI is in its Rest mode, the SCK clock output is in a state selected by CPOL, bit 3 of SPICR. Thus with CPOL set to a value of “1” the SCK output will be LOW. The value of SDO will be LOW (non-programmable) when the SPI is enabled but inactive.

If the SPI is enabled and in the Rest (passive) state SDO and hence SDA will be LOW.

If the SPI is disabled SDO will be released to HIGH impedance, and hence to the HIGH level by the presence of the Pull-up resistor. It may be pulled LOW by loading a Zero into the Port 2 Pin 1 output buffer and then specifying this pin as a normal Port output pin.

Having established these basic preliminaries we can proceed to discuss the provision, by software, of basic **I²C-bus** operations.

SIMULATION OF BASIC I²C-bus OPERATIONS

Using the basic operations described in the above Sections the various **I²C-bus** Protocol features may be implemented as follows.

Generation of START Conditions

The generation of a START condition is implemented in Procedure `INIT_START_I2C` (Appendix A).

a) Disable the SPI unit putting SDA and SCK in the High-impedance state.

b With the SPI disabled and SCK HIGH, pull the SDO line LOW by respecifying SDI as a normal output.

c) Hold the above condition for a period of ~5 µs by calling the `DELAI` Macro (see Appendix A).

d) Enable the SPI, specifying SCK to the rest clock state (LOW).

e) Respecify the SDO output as an Alternate Function.

Generation of STOP Conditions

The generation of a STOP condition is implemented in Procedure `GEN_STOP` (see Appendix A).

a) Pull the SDA line LOW by respecifying SDO as a normal Port output.

b) Release SCK to HIGH by disabling the SPI. Note that SDA will remain LOW.

c) Hold this condition for ~5 µs using `DELAI` Macro (see Appendix A) so as to meet the set-up Time specification

d) Respecify SDO as an Alternate Function and hence allow SDA to be pulled HIGH by the Pull-up resistor.

Generation of 9th Clock Pulse with Acknowledge Test

After the transmission of 8 Data bits a 9th Clock Pulse may be generated and the Acknowledge tested as implemented in Procedure TEST_ACK (see Appendix A).

a) Release SCK and SDA to the HIGH impedance state by disabling the SPI.

b) Wait until the SCK line goes HIGH.

c) Test for LOW on the SDA line placed by the Receiver (Slave).

d) Hold the SCK line HIGH for 5 µs using `DELAI` Macro.

e) Force SCK and SDA to LOW by enabling the SPI.

Generation of 9th Clock and Acknowledge

After the reception of 8 Data bits a 9th Clock Pulse may be generated and an Acknowledge asserted as implemented in Procedure GEN_ACK (see Appendix A).

a) Pull the SDA line LOW by respecifying SDO as a normal Port output.

b) Release SCK to HIGH by disabling the SPI. Note that SDA will remain LOW.

c) Hold the SCK line HIGH for 5 µs using `DELAI` Macro.

d) Force SCK to LOW by enabling the SPI.

e) Finally respecify the SDO Port pin as an Alternate Function.

TYPES OF TRANSFER OPERATION SUPPORTED

The ST9 supports the following three types of transfer with an electrically erasable serial memory (EEPROM) which features an **I²C-bus** protocol, e.g. ST24C02.

* Random Write (1 to 8 bytes),
* Random Read (1 to N bytes),
* Current Address Read (or Verify), (1 to N bytes.)

Random Write Mode

The serial **I²C-bus** protocol for Random Write Operations is shown in Figure 4 (single byte) or Figure 5 (for up to 8 bytes).

Figure 4. I²C-bus Protocol for Random Write Mode (1 byte)

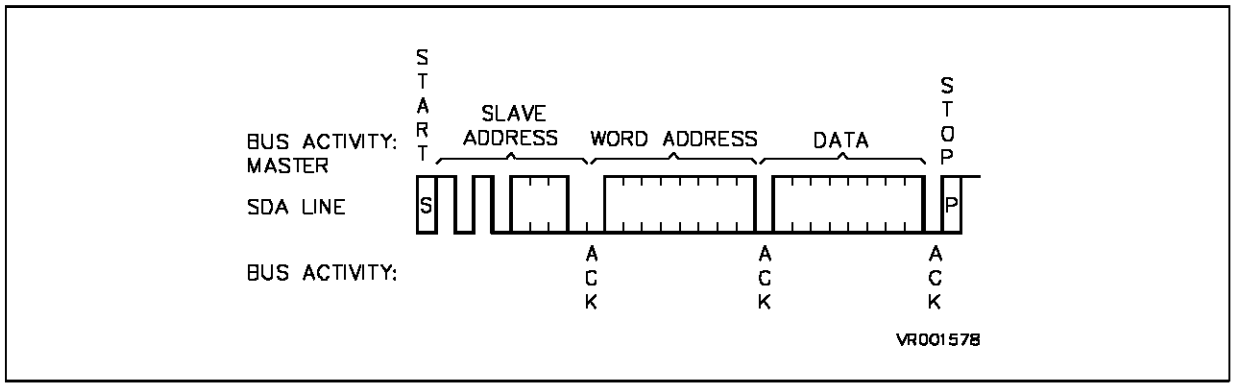
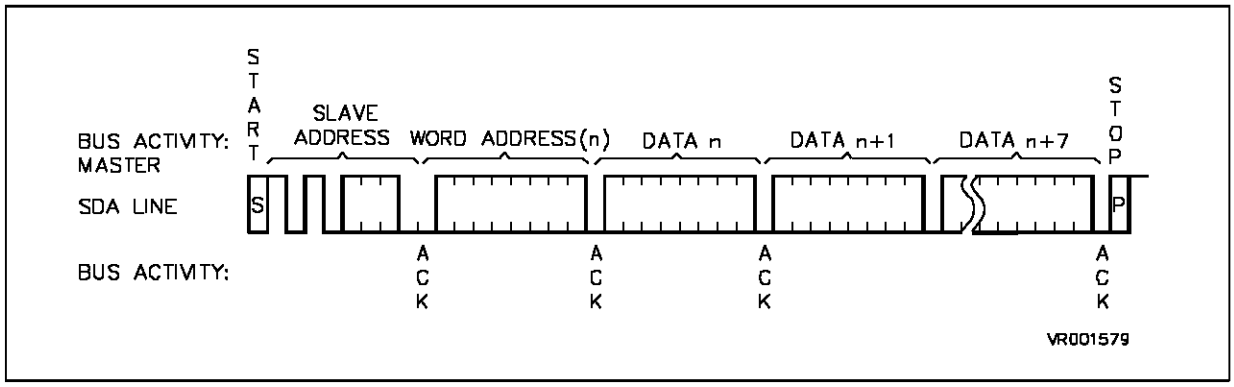


Figure 5. I²C-bus Protocol for Random Write Mode (N bytes)



To Write a single byte the Master ST9 has to transmit a sequence of 3 bytes representing successively:

a) Slave Address: 7 bits + 8th bit = “0” signifying Transmit operation.

b) Word Address: 8 bits.

c) Data value: 8 bits.

The ST9 Master generates the START condition and then transmits the sequence of 3 bytes by successively loading them into the SPI Data Register. Each such Data load generates a sequence of 8 clocks and 8

Data bits, after which the ST9 generates a 9th clock pulse and tests for an Acknowledge from the Slave. After the data pulse has been received and Acknowledged by the Slave the Master terminates the transfer by generating a STOP condition.

To Write a page of N bytes (1< N> 8) the Master ST9 has to transmit the above sequence of 3 bytes followed by the remaining N - 1 data bytes. The Slave Device contains an 8-bit address pointer, the 3 low order bits of which are incremented by 1 after each Read/Write operation with the 5 high order bits remaining constant. Thus a page of up to N = 8 bytes may be written in this way.

The Transfer sequence proceeds as described above except that the Slave continues to accept data words for writing to sequential locations until such time as the Master signals end of Transmission by sending a STOP condition.

Random Read Mode

The serial **I²C-bus** protocol for Random Read Operations is shown in Figure 6.

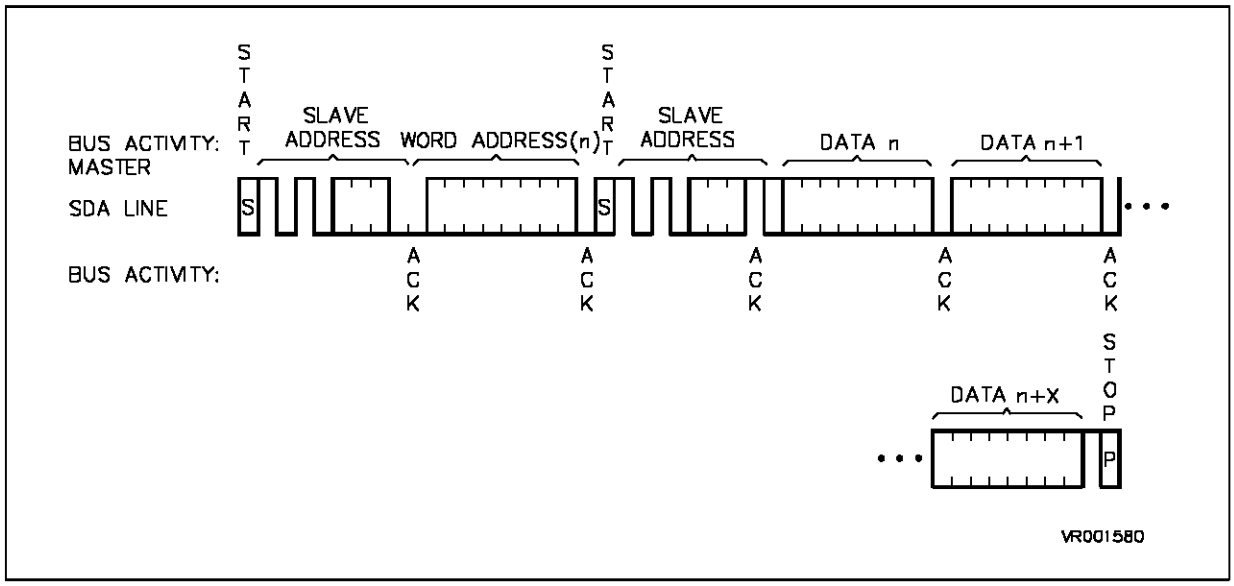
To Read a single byte the Master ST9 has to transmit a sequence of 3 bytes representing successively:

a) Slave Address: 7 bits + 8th bit = “0” signifying Transmit operation.

b) Word Address: 8 bits.

c) Slave Address: 7 bits + 8th bit = “1” signifying Receive operation.

Figure 6. I²C-bus Protocol for Random Read Mode (N bytes)



The ST9 Master generates the START condition and then transmits a dummy Write operation comprising the Slave Address byte, followed by the Word Address. Both these byte operations are followed by a 9th clock pulse and a concurrent test for Slave Acknowledge.

At this point the Master Transmitter must become the Master Receiver. This is achieved by sending another START condition, followed by the retransmission of the Slave Address with the 8th bit set now to "1" to indicate that the subsequent data transfers are from the slave to the ST9 Master.

From this point on the Slave will provide words addressed in sequence as long as the Master continues to Acknowledge receipt of data. Note that the address counter for Read operations increments over all 8 address bits, thus enabling the entire memory to be Read in one operation. The Master can terminate the transfer at any time by generating a STOP condition instead of an Acknowledgement.

Current Address Read Mode

In this alternative Read mode the Master reads from memory at the last location referenced in either Read or Write mode.

The serial **I²C-bus** protocol for Current Address Read Operations is shown in Figure 7.

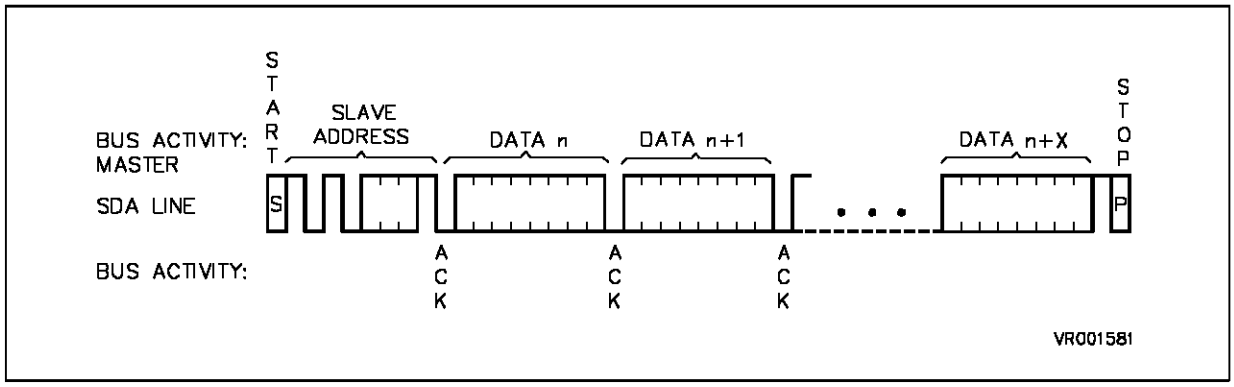
To Read any number of bytes the Master ST9 has to transmit a single byte.

Device Address: 7 bits + 8th bit = "1" signifying Receive operation.

The ST9 Master generates the START condition and then transmits the Slave Address byte. At this point the Master now issues an Acknowledge indicating that it requires additional data.

From this point on the Slave will provide words addressed in sequence as long as the Master continues to Acknowledge receipt of data. The Master can terminate the transfer at any time by issuing a STOP condition instead of an Acknowledgement.

Figure 7. I²C-bus Protocol for Current Address Read mode (N bytes)



EEP_MAN: AN I²C-bus PROTOCOL EEPROM MANAGER

Appendix A contains a detailed Assembler listing of a representative example of an EEPROM manager for a device respecting the I²C-bus serial protocol. This example is not intended to be definitive but should be taken as illustrative example of the use of the ST9 in such applications. Modifications and extensions, depending on the particular application, will readily occur to the Application Engineer, e.g. the use of the ST9 stacks as an alternative mechanism for transferring data and parameters between the Manager and the calling program. Note that Appendix A makes use of a number of Macros which are separately listed and defined in Appendix C.

The EEP_MAN/ Calling Program Interface

A calling program interfaces to EEP_MAN using four registers for calling parameters and a register-file for data.

Parameter/ Transfer-Status Registers

A call to EEP_MAN is initialized by loading parameter values into three registers, viz. EEP_FUNCT, EEP_ADD, and NB_BYTES. The status of a current transfer can be monitored by reading a fourth register, STAT_EEP, in which EEP_MAN records a value giving the status of the EEPROM device.

EEP_FUNCT Register, R3.

This register is loaded with one of the following values to specify the mode of data transfer required:

1: READ_FUNCT: Random READ mode.

2: WRITE_FUNCT: Random WRITE mode.

3: VERIFY_FUNCT: Current Address (Verify) mode.

EEP_ADD Register, R0.

This register should be loaded with the value of the EEPROM byte starting address for Random READ/WRITE operations. For a current address (Verify) operation the contents of this register is a Don't-Care value.

NB_BYTES Register, R6.

This register should be loaded with the number of bytes, #N, which should be transferred in the operation. This value may have a value from 1 to 8 for Write operations, or 1 to 256 for READ operations.

STAT_EEP Register, R4.

EEP_MAN loads this register with one of the following values to specify the current EEPROM Status.

0: EEP_OK: The EEPROM is OK.

1: LECT_ON: The EEPROM is reading a byte (random address mode).

2: VERIF_ON: The EEPROM is reading the current byte.

3: ECR_ON: The EEPROM is programming a byte (random address mode).

4: NO_ACK: The EEPROM has not Acknowledged a byte transferred from the ST9

80h: EEP_FREE_MASK: The EEPROM is available for a new operation.

Transfer of Data Values

DATA_TABLE Register File

A register-file, starting at R32 and of size #N should be reserved for READ/VERIFY operations, or loaded with data to be transferred to the EEPROM for a WRITE operation. The first byte to be transferred should be loaded into register R31+#N, and the last byte should be loaded into register R32.

EEP_MAN Data Transfer Initialization Routines

After loading the Parameter registers and setting up and, if appropriate, loading the Data table, the calling routine tests STAT_EEP to check that the EEPROM is free, and then calls Procedure EEP_MAN.

This procedure first saves the byte address counter value, NB_BYTE, specifies the Port 2 pins SDO and SCK as Alternate Functions, and SDI as an input, and then calls one of the three main initializing routines READ_EEP, VERIF_EEP, or WRITE_EEP, depending on the value transferred in register EEP_FUNCT.

These three procedures essentially carry out identical functions. After verifying that the EEPROM is not busy, they enable the SPI interrupt, generate a START condition, and transfer the EEPROM device address by loading this value into the SPI Data Register, SPIDR.

Note that the EEPROM Device Address is 7 bits long together with an eight bit which is set to “0” for READ or WRITE operations, and set to ”1” for VERIFY operations. In addition, a value of “1” is loaded into the Transaction Status Register, STAT_TRANS_SPI to indicate that the Device Address has been transferred. This register is loaded with an appropriate identifying value each time the SPI Data Register is loaded.

STAT_TRANS_SPI Register, R5.

This register serves as an internal Status register, used by EEP_MAN and its associated routines, to maintain a record of the nature of the current ST9 to EEPROM transfer.

1: T_ADD_SLAVE: The EEPROM device address has been transferred.

2: T_ADD_EEP: The EEPROM byte address has been transferred.

3: TRANS_WR_DATA: A WRITE byte has been transferred.

4: TRANS_RD_DATA: A READ byte has been transferred.

After initiating a byte transfer by loading the SPI Data Register, SPIDR, a return is made to the calling routine. At the completion of the byte transfer (8 SCK clock pulses) the SPI raises an interrupt on channel B0 (associated to external interrupt INT2).

The SPI Interrupt Service Routine

This routine is called at the termination of the transmission of each byte representing a Device Address, Word Address, READ data, or Write data. The action effected by this routine (Procedure IT_END_TRANS, see Appendix A) depends upon the values contained in the following registers:

1: STAT_TRANS_SPI Register, R5.

2: STAT_EEP Register, R4.

3: NB_BYTES Register, R6.

4: EEP_FUNCT Register, R3.

The required action depends on the nature of the previously transferred byte, indicated by the value contained in STAT_TRANS_SPI. In the case of data byte transfers the next action also depends on whether the required number of bytes has been transferred, as indicated by the value of NB_BYTES.

The organization of IT_END_TRANS is illustrated by the flow diagram of Figure 8. This will be described by considering in detail the logical flow of events associated with each of the three modes of data transfer.

Random Write Mode

Figure 5 illustrates the sequence of byte transfers involved in writing N bytes in Random Write Mode, observing the I²C-bus protocol.

(i) Transmission of Slave Device Address.

This operation is initiated by Procedure WRITE_EEP which generates a START condition, loads the Device address (with the 8th bit set to 0) in SPIDR, thus initiating the transfer, and then returns to the calling program.

In addition, this routine loads the following values into the Status Registers:

```
STAT_TRANS_SPI          <-1 (#T_ADD_SLAVE)
STAT_EEP                <-3 (#ECR_ON)
```

(ii) Transmission of Word Address.

After transmission of the 8 bits of the Device Address, an Interrupt is raised and entry made to Interrupt Procedure IT_END_TRANS. The logical flow then follows the path AI (refer to Figure 8), as a result of which the required random Word address is loaded into SPIDR, so effecting the required byte transfer.

In addition, this routine loads (or retains) the following values in the Status Registers:

```
STAT_TRANS_SPI          <-2 (#T_ADD_EEP)
STAT_EEP                <-3 (#ECR_ON)
```

(iii) Transmission of 1st Data Byte.

After transmission of the 8 bits of the Word Address, an Interrupt is raised and entry made to Interrupt Procedure IT_END_TRANS. The logical flow then follows the path ACG (refer to Figures 8, 8b), as a result of which the 1st Data Byte is loaded into SPIDR, so effecting the required byte transfer.

In addition, this routine loads (or retains) the following values in the Status Registers:

```
STAT_TRANS_SPI          <-3 (#TRANS_WR_DATA).
STAT_EEP                <-3 (#ECR_ON)
```

Figure 8. Flow Diagram of the IT_END_TRANS Interrupt Routine

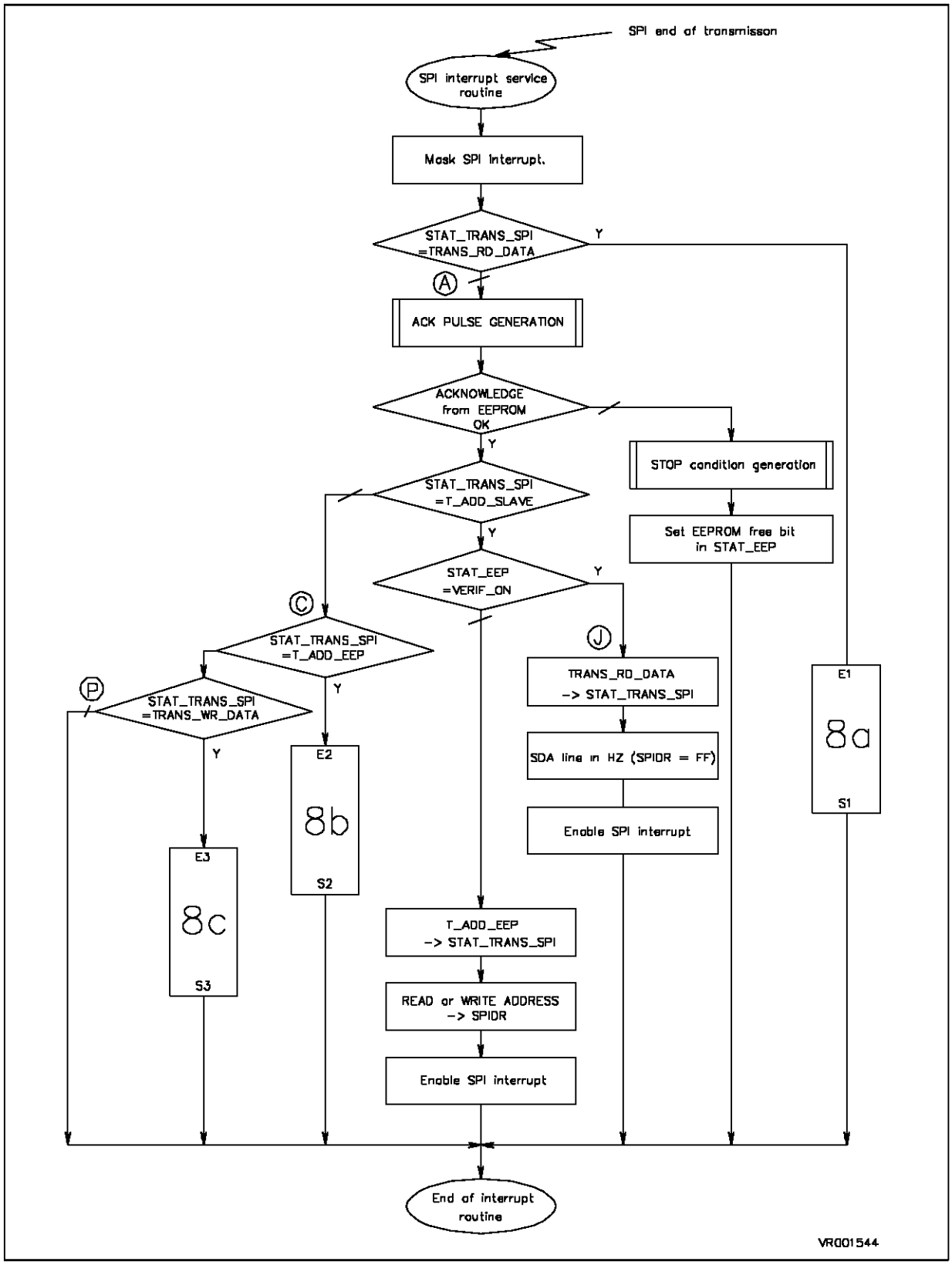


Figure 8a. Flow Diagram of the IT_END_TRANS Interrupt Routine (continued)

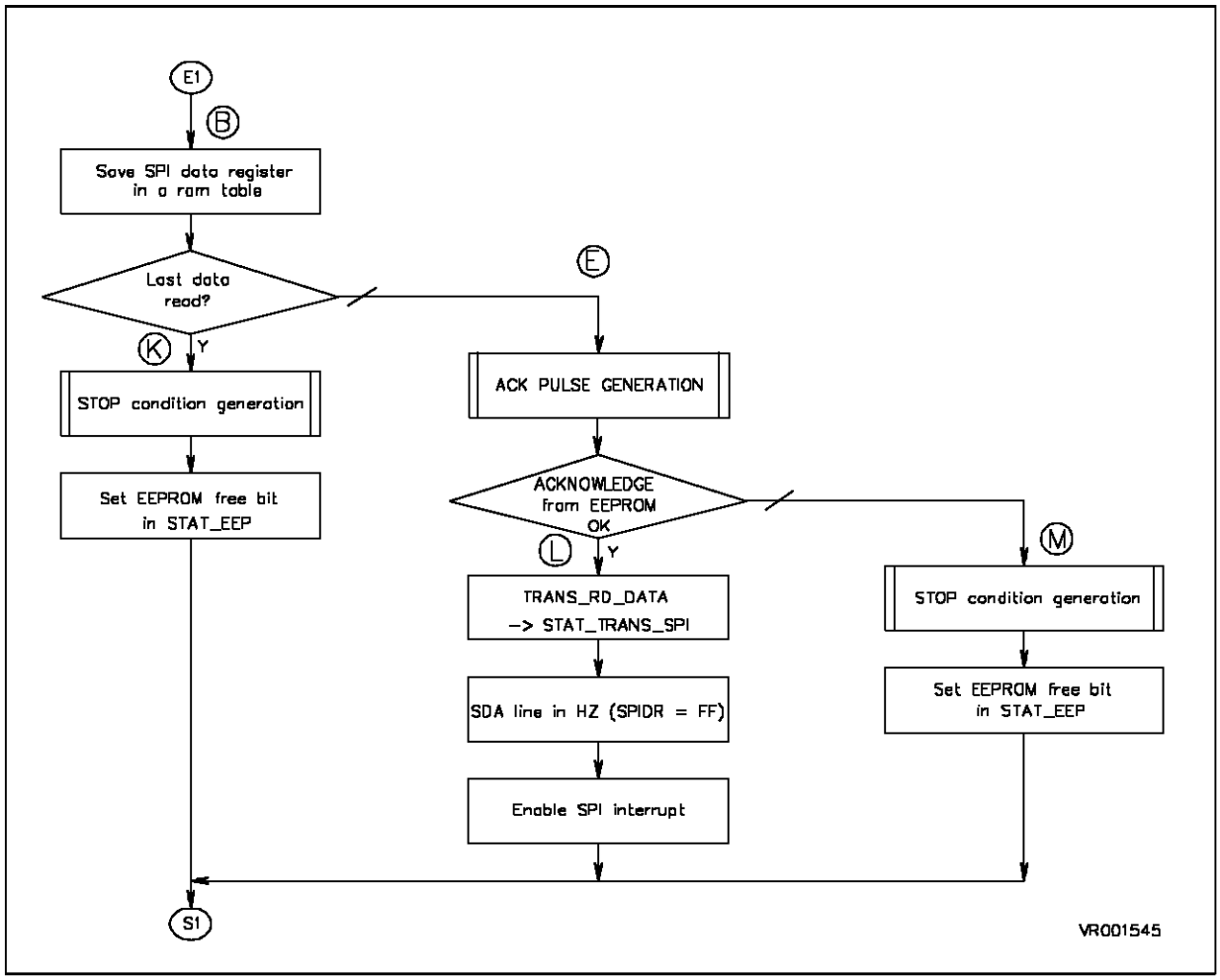


Figure 8b. Flow Diagram of the IT_END_TRANS Interrupt Routine (continued)

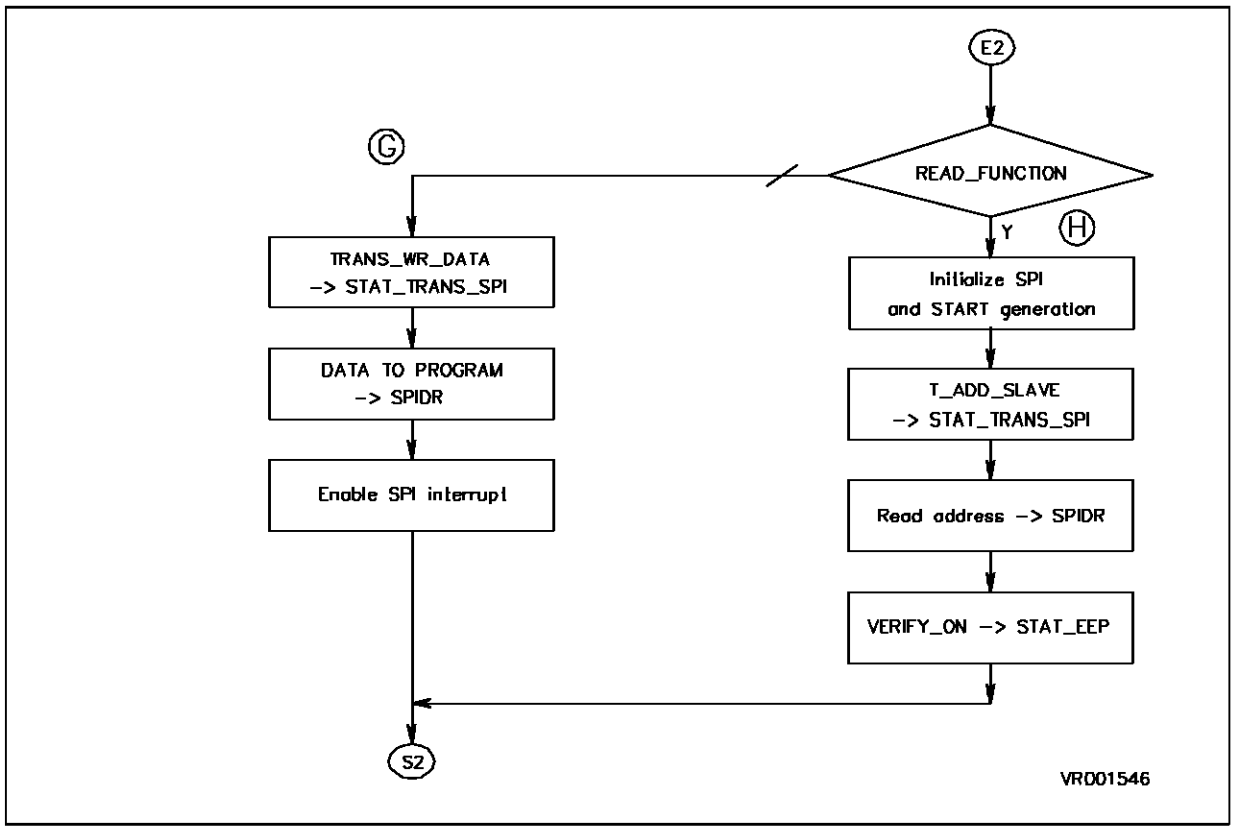
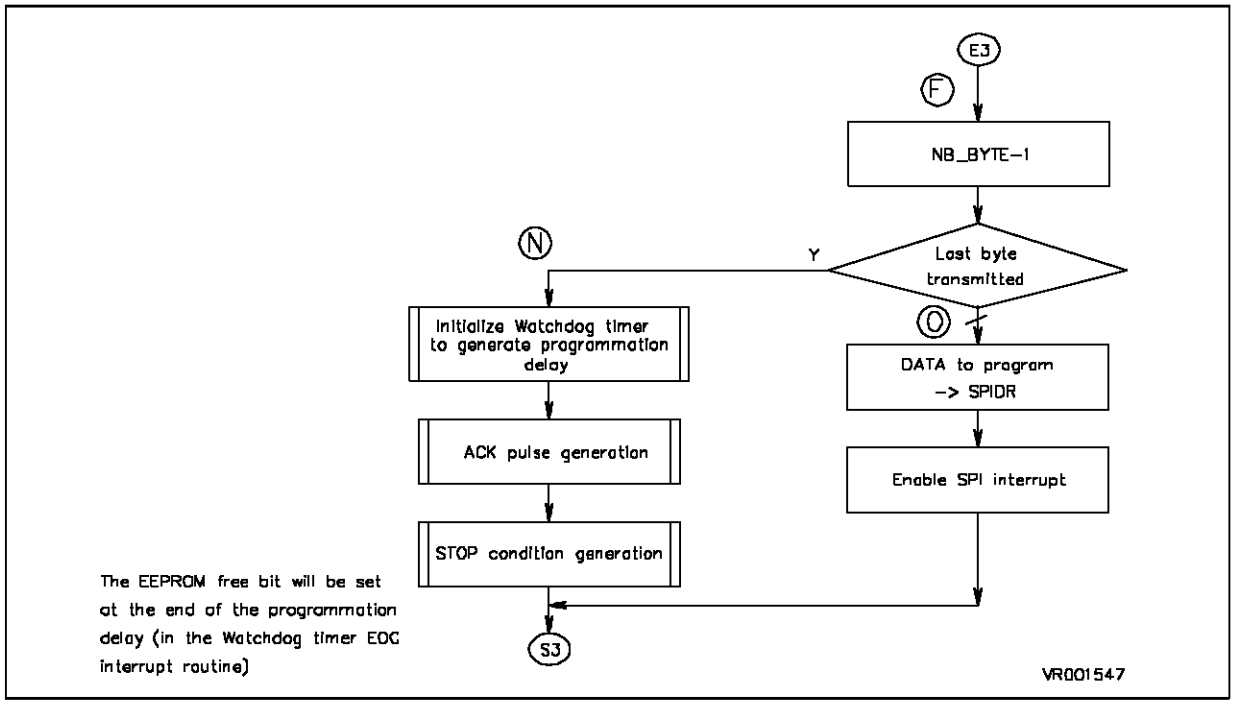


Figure 8c. Flow Diagram of the IT_END_TRANS Interrupt Routine (continued)



(iv) Transmission of Subsequent Data Bytes.

After transmission of Byte #M (1 < M < N), an Interrupt is raised and entry made to Interrupt Procedure IT_END_TRANS. The logical flow then follows the path ACFO (refer to Figures 8, 8c) as a result of which Data Byte #M + 1 is loaded into SPIDR, so effecting the required byte transfer.

The following values are retained in the Status Registers:

STAT_TRANS_SPI <-3 (#TRANS_WR_DATA).

STAT_EEP <-3 (#ECR_ON)

(v) Transmission of the final Data Byte.

After transmission of Byte #N, an Interrupt is raised and entry made to Interrupt Procedure IT_END_TRANS. The logical flow then follows the path ACFN (refer to Figures 8, 8c). On this occasion the Watch-Dog Timer routine, PROG_DELAY (see Appendix A) is entered to generate a delay equal to N x 5 milliseconds to enable the EEPROM to be programmed with the new data values.

For this purpose the Watch_Dog Timer is initialized in Single Operation, Count-down Mode, and a constant value is loaded into the counter appropriate to the required delay. An interrupt is enabled on Channel A0 for the Timer EOC event, and a return is made to the calling program.

When the Timer times out, entry is made to interrupt routine TEMP0 (see Appendix A). This routine clears the A0 interrupt pending bit, sets the EEP_FREE_MASK bit to 1, and returns to the calling program. At this point the EEPROM is available again for further data transfers.

Random READ Mode

Figure 6 illustrates the sequence of byte transfers involved in reading N bytes in Random Read Mode, observing the I²C-bus protocol.

(i) Transmission of Slave Device Address.

This operation is initiated by Procedure READ_EEP which generates a START condition, loads the Device address in SPIDR (with the 8th bit set to “0”), thus initiating the transfer, and then returns to the calling program.

In addition, this routine loads the following values into the Status Registers:

STAT_TRANS_SPI <-1 (#T_ADD_SLAVE)

STAT_EEP <-1 (#LECT_ON)

(ii) Transmission of Word Address.

After transmission of the 8 bits of the Device Address, an Interrupt is raised and entry made to Interrupt Procedure IT_END_TRANS. The logical flow then follows the path AI (refer to Figure 8), as a result of which the required random Word address is loaded into SPIDR, so effecting the required byte transfer.

In addition, this routine loads (or retains) the following values in the Status Registers:

STAT_TRANS_SPI <-2 (#T_ADD_EEP)

STAT_EEP <-1 (#LECT_ON)

(iii) Retransmission of Slave Device Address.

After transmission of the 8 bits of the Word Address, an Interrupt is raised and entry made to Interrupt Procedure IT_END_TRANS. The logical flow then follows the path ACH (refer to Figure 8), as a result of which the Device address (with the 8th bit set to "1"), loaded into SPIDR, so effecting the required byte transfer.

In addition, this routine loads the following values in the Status Registers:

```
STAT_TRANS_SPI          <-1 (#T_ADD_SLAVE)
STAT_EEP                <-2 (#VERIF_ON)
```

(iv) Read of 1st Data Byte.

After the retransmission of the 8 bits of the Device Address, an Interrupt is raised and entry made to Interrupt Procedure IT_END_TRANS. The logical flow then follows the path AJ (refer to Figure 8), as a result of which a value of 0FFh is loaded into SPIDR, so effecting the required byte transfer from the Slave Memory.

In addition, this routine loads (or retains) the following values in the Status Registers:

```
STAT_TRANS_SPI          <-4 (#TRANS_RD_DATA).
STAT_EEP                <-2 (#VERIF_ON)
```

(**v) Read of Subsequent Data Bytes.**

After transmission of Byte #M (1 < M < N), an Interrupt is raised and entry made to Interrupt Procedure IT_END_TRANS. The logical flow then follows the path BEL (refer to Figure 8), as a result of which Data Byte #M + 1 is loaded into SPIDR, so effecting the required byte transfer.

The following values are retained in the Status Registers:

```
STAT_TRANS_SPI          <-4 (#TRANS_RD_DATA).
STAT_EEP                <-2 (#VERIF_ON)
```

(vi) Read of the final Data Byte.

After transmission of Byte #N, an Interrupt is raised and entry made to Interrupt Procedure IT_END_TRANS. The logical flow then follows the path BK (refer to Figure 8), as a result of which the STOP condition is generated and the EEPROM free bit set in STAT_EEP.

Current Address READ (Verify) Mode

Figure 7 illustrates the sequence of byte transfers involved in reading N bytes in Random Write Mode, observing the **I²C-bus** protocol.

(i) Transmission of Slave Device Address.

This operation is initiated by Procedure VERIF_EEP which generates a START condition, loads the Device address (with the 8th bit set to "1") in SPIDR, thus initiating the transfer, and then returns to the calling program.

In addition, this routine loads the following values into the Status Registers:

```
STAT_TRANS_SPI          <-1 (#T_ADD_SLAVE)
STAT_EEP                <-2 (#VERIF_ON)
```

(ii) Read of 1st Data Byte.

After the retransmission of the 8 bits of the Device Address, an Interrupt is raised and entry made to Interrupt Procedure IT_END_TRANS. The logical flow then follows the path AJ (refer to Figure 8), as a result of which a value of 0FFh is loaded into SPIDR, so effecting the required byte transfer from the Slave Memory.

In addition, this routine loads (or retains) the following values in the Status Registers:

```
STAT_TRANS_SPI          <-4 (#TRANS_RD_DATA).
STAT_EEP                <-2 (#VERIF_ON)
```

(iii) Read of Subsequent Data Bytes.

After transmission of Byte #M (1 M N), an Interrupt is raised and entry made to Interrupt Procedure IT_END_TRANS. The logical flow then follows the path BEL (refer to Figure 8), as a result of which Data Byte #M + 1 is loaded into SPIDR, so effecting the required byte transfer.

The following values are retained in the Status Registers:

```
STAT_TRANS_SPI          <-4 (#TRANS_RD_DATA).
STAT_EEP                <-2 (#VERIF_ON)
```

(iv) Read of the final Data Byte.

After transmission of Byte #N, an Interrupt is raised and entry made to Interrupt Procedure IT_END_TRANS. The logical flow then follows the path BK (refer to Figure 8), as a result of which the STOP condition is generated and the EEPROM free bit set in STAT_EEP.

ILLUSTRATIVE CALLING ROUTINES

Appendix B contains listing of suitable calling routines to WRITE 4 bytes to the Serial EEPROM or to READ 6 bytes. Included also in Appendix B are the appropriate ST9 Core System and Peripheral initialization routines (see also Reference 1).

These programs make use of the File of ST9 Standard Register and Register Bit Definitions listed in Application Note AN411, **SYMBOLS.INC.**

It will be noted that the calling routines, after initiating the data transfers, wait in test and branch loops until the EEPROM is free. In a practical real-time application this waiting time (>N.5 mS for an N byte WRITE transfer) could be used for useful processing.

REFERENCES

(1) Application Note 413, “Initialization of the ST9”, Pierre Guillemin and Alan Dunworth, SGS-THOMSON Microelectronics.

(2) The “ST9 Technical Manual”, SGS-THOMSON Microelectronics.

Appendix A. EEPROM I²C-bus Manager Routine

```
.title“ ST9 SPI use with I 2C protocol.              January 24 1990 ”
.sbttl“ EEPROM manager                                  version 2.0 ”
.list bex
.global            IT_END_TRANS, TEMPO, EEP_MAN
.extern            RESET_START


;**************************
;* Module Macro Definitions *
;**************************
.library    “c:\st9\inc\bitmacro.inc”     ; change as required
.mcall      ifbit, attbit
;————————————————————————————————
.macroDELAI ?loop_var

      ld    COUNTER,#03h          ; 10 Tcy.
loop_var:
      dec   COUNTER               ; 6 Tcy.
      jrnz  loop_var              ; 12 Tcy: A loop = 1.5 fs
                                  ; with a 12 MHz system clock.
.endm
;————————————————————————————————


;————————————————————————————————
.macroDIS_SPI_IT           ; Disable SPI interrupt.
      and   EIPR,#~ipb0m          ; Reset the B0 ( SPI interrupt) pending bit.
      and   EIMR,#~ib0m           ; Disable SPI channel (B0).
.endm
;————————————————————————————————


;————————————————————————————————
.macroEN_SPI_IT            ; Enable SPI interrupt.
      and   EIPR,#~ipb0m          ; Clear request on SPI channel (B0).
      nop
      or    EIMR,#ib0m            ; Enable SPI channel (B0).
.endm
;————————————————————————————————


;————————————————————————————————
.macroINIT_TRANS_READ      ; Initialize SPI register and interrupt
                                  ; for read operation.
      ld    STAT_TRANS_SPI,#TRANS_RD_DATA       ; Initialisation for read operation.
      spp   #0
      ld    SPI_TAMP,#0FFH        ; To read the data from the EEPROM.
.endm
;————————————————————————————————
```

Appendix A. EEPROM I²C-bus Manager Routine (Continued)

```
;***********************
; Register declarations.
;***********************

EEP_ADD            =       R0     ; Operation address in the EEPROM.
WRITE_DATA         =       R1     ; Data to be programmed in the EEPROM.
write_data         =       r1
READ_DATA          =       R2     ; Data which has been read from the EEPROM.
read_data          =       r2
EEP_FUNCT          =       R3
STAT_EEP           =       R4
STAT_TRANS_SPI     =       R5
NB_BYTE            =       R6     ; Number of bytes to be written
                                  ; (maximum 8) or to read.
nb_byte            =       r6
SPI_TAMP           =       R7
MEMO_NB_BYTE       =       R14
COUNTER            =       R15
DATA_TABLE         =       31     ; The real beginning of the table
;                          (1Fh)  ; to store data is R20h.

;***********************
; Constant declarations.
;***********************

ADD_EEP_W          ==      0A0h   ; Address the external EEPROM slave
                                  ; for WRITE operation.
ADD_EEP_R          ==      0A1h   ; Address the external EEPROM slave
                                  ; for READ operation.
SDI_MASK           =       02h    ; SDI = bit 1 of port 2.
SCK_MASK           =       04h    ; SCK = bit 2 of port 2.
SDO_MASK           =       08h    ; SDO = bit 3 of port 2.

;——— Status of EEP_FUNCT register.
;     This register is used to indicate the EEPROM manager the
;     function to be executed.

READ_FUNCT         ==      1      ; Read mode: read after transferring the
                                  ; address pointer.
                                  ; ie: Read from the current address.
WRITE_FUNCT ==     2      ; Write mode.
VERIF_FUNCT ==     3      ; Alternate read mode:
                                  ;Read operation without programming
                                  ;the address pointer.
```

Appendix A. EEPROM I²C-bus Manager Routine (Continued)

```
;——— Status of STAT_TRANS_SPI register.
;     This register permits the EEPROM manager (in the SPI interrupt routine) ...
;     ... to know the type of the byte which has just been transmitted.

T_ADD_SLAVE ==     1       ; The eeprom address has been transferred.
T_ADD_EEP          ==      2      ; The operation address has been transferred.
TRANS_WR_DATA      ==      3      ; The data to be written has been transferred.
TRANS_RD_DATA      ==      4      ; The data to be read has been received.

;——— Status of STAT_EEP register.
;           This register permits the caller to know the status of the EEPROM.

EEP_OK      ==     0       ; EEPROM is OK.
LECT_ON            ==      1      ; EEPROM is reading a byte.
VERIF_ON           ==      2      ; EEPROM is reading the current byte.
ECR_ON      ==     3       ; EEPROM is programming a byte.
NO_ACK      ==     4       ; EEPROM has not acknowledged.

EEP_FREE_MASK      ==      80h    ; EEPROM is ready for a new operation...
                                  ; ... if this bit is equal to 1.
.text
;*****************************************************************************
; EEP_MANAGER:     EEPROM MANAGER.
;
;*****************************************************************************
proc  EEP_MAN [PPR] {             ; Save page pointer.
      spp   #0
      DIS_SPI_IT
      ld    MEMO_NB_BYTE,NB_BYTE  ; Save NB_BYTE Before decrement for
                                  ; programmation tempo.
      switch [ EEP_FUNCT ] {
            case   #READ_FUNCT:
                   call    READ_EEP
            case   #VERIF_FUNCT:
                   call    VERIF_EEP
            case   #WRITE_FUNCT:
                   call    WRITE_EEP
      } ;—- End of switch.
} ;— End of proc.
```

Appendix A. EEPROM I²C-bus Manager Routine (Continued)

```
;*****************************************************************************
;     READ_EEP:    Normal read mode.
;                  Read of some bytes after setting the slave address.
;
;*****************************************************************************
proc  READ_EEP [PPR] {
      ifbit STAT_EEP,#EEP_FREE_MASK      ; Test if EEPROM free.
;     {
            call   INIT_START_I2C         ; SPI and related interrupt initialization..
                                          ; ... to support I2C protocol...
                                          ; ... Generate a start condition.
            ld     STAT_TRANS_SPI,#T_ADD_SLAVE;  Slave address will be transferred.
            ld     STAT_EEP,#LECT_ON      ; A read condition is started.
                                          ; EEPROM is not FREE = EEP_FREE_BIT = 0.
            ld     SPIDR,#ADD_EEP_W       ; EEPROM address in write mode to transfer
                                          ; pointer.
      } ;-- End of if.
} ;-- end of proc.

;*****************************************************************************
;     VERIF_EEP:   Alternate read mode.
;                  Read of some bytes without setting the address pointer.
;
;*****************************************************************************
proc  VERIF_EEP [PPR] {
      ifbit STAT_EEP,#EEP_FREE_MASK      ; Test if EEPROM free.
;     {
            call   INIT_START_I2C         ; SPI and related interrupt initialization..
                                          ; ... to support I2C protocol...
                                          ; ... Generate a start condition.
            ld     STAT_TRANS_SPI,#T_ADD_SLAVE;  Slave address will be transferred.
            ld     STAT_EEP,#VERIF_ON     ; A verif condition is started.
                                          ; EEPROM is not FREE = EEP_FREE_BIT = 0.
            ld     SPIDR,#ADD_EEP_R       ; EEPROM address in read mode.
      } ;-- End of if.
} ;-- end of proc.
```

Appendix A. EEPROM I²C-bus Manager Routine (Continued)

```
;*****************************************************************************
;      WRITE_EEP:  Write of some bytes.
;
;*****************************************************************************
proc  WRITE_EEP [PPR] {
      ifbit STAT_EEP,#EEP_FREE_MASK      ; Test if EEPROM free.
;     {

            call   INIT_START_I2C        ; SPI and related interrupt initialization..
                                         ; ... to support I2C protocol...
                                         ; ... Generate a start condition.
            ld     STAT_TRANS_SPI,#T_ADD_SLAVE;  Slave address will be transferred.
            ld     STAT_EEP,#ECR_ON      ; A write condition is started.
                                         ; EEPROM is not FREE = EEP_FREE_BIT = 0.
            ld     SPIDR,#ADD_EEP_W      ; EEPROM address in write mode.
      } ;-- End of if.
} ;-- end of proc.

;*****************************************************************************
;     INIT_START_I2C:
;           Initialize SPI to support I2C protocol.
;           Generation of a start condition.
;
;*****************************************************************************
proc  INIT_START_I2C [PPR] {
;-- SPI initialization.
      spp   #0                     ; SPI and ext. interrupts registers in page 0.
      ld    SPICR,#042h            ; SPI is Disabled = SDA and SCK in HZ (1).
                                   ; I2C bus mode is selected.
                                   ; SCK frequency # 100 kHz.
;-- START condition generation.
      and   P2DR,#~SDO_MASK; Prepare "0" on output buffer of SDO.
      spp   #P2C_PG
      and   P2C0R,#~SDO_MASK       ; SDO line in output - SDA line = "0".
      DELAI                        ; Wait for start condition hold time.
      spp   #0
      or    SPICR,#spen            ; Enable SPI.
      EN_SPI_IT                    ; Enable SPI interrupt.
      spp   #P2C_PG
      or    P2C0R,#SDO_MASK; SDO line in AF.
} ;-- End
```

Appendix A. EEPROM I²C-bus Manager Routine (Continued)

```
;*****************************************************************************
;     GEN_STOP:    Generation of a stop condition.
;
;*****************************************************************************
proc  GEN_STOP [PPR] {
      spp   #0
      DIS_SPI_IT                  ; Disable SPI interrupt.
      and   P2DR,#~SDO_MASK; Prepare “0” on output buffer of SDO.
      spp   #P2C_PG
      and   P2C0R,#~SDO_MASK      ; SDO line in output - SDA line = “0”.
      spp   #0
      and   SPICR,#~spen          ; Disable SPI - Release SCK line - SCK = “1”.
      DELAI                       ; Wait for stop condition setup.
      spp   #P2C_PG
      or    P2C0R,#SDO_MASK; SDO in AF - Release SDA line - SDA = “1”.
} ;—- End

;*****************************************************************************
;     GEN_ACK:     ACK pulse generation,
;                  and force the SDA line to 0 for Acknowledgement.
;
;*****************************************************************************
proc  GEN_ACK [ PPR ] {
      and   P2DR,#~SDO_MASK; Prepare “0” on output buffer of SDO.
      spp   #P2C_PG
      and   P2C0R,#~SDO_MASK      ; SDO line in output - SDA line = “0”.
      spp   #0
      and   SPICR,#~spen          ; Disable SPI - Release SCK line - SCK = “1”.
      DELAI                       ; Wait for ACK hold time.
      or    SPICR,#spen           ; Enable SPI - Force SDA and SCK low.
      spp   #P2C_PG
      or    P2C0R,#SDO_MASK; SDO line in AF.
} ;—- End of proc.
```

Appendix A. EEPROM I²C-bus Manager Routine (Continued)

```
;*************************************************************************
;     TEST_ACK:    ACK pulse generation,
;                  and check the slave acknowledgment.
;*************************************************************************
proc  TEST_ACK [ PPR ] {
      and          SPICR,#~spen          ; Release SPI lines in disabling it.
      attbit       P2DR,#SCK_MASK        ; Wait for SCK going high.
      ifbit P2DR,  #SDI_MASK             ; Check if receiver has acknowledged.
                                         ;(SDA = 0).
;     {                                  ; If no ACK.
            ld     STAT_EEP,#NO_ACK
      } else {                           ; If ACK ok.
            DELAI                        ; Wait for high period of the clock.
      } ;-- End of if.
      or    SPICR,#spen                  ; Enable SPI - Force SDA low.
} ;-- End of proc.

;*************************************************************************
;     IT_END_TRANS:        SPI end of transmission interrupt service routine.
;                          This interrupt is connected to channel B0 in the ST9.
;*************************************************************************
IT_END_TRANS::
      pushu        PPR
      pushuw       RPP
      srp   #0
      spp   #0
      if [ STAT_TRANS_SPI == #TRANS_RD_DATA ] {

;-- A data to be read has been received from EEPROM.
            ld     read_data,SPIDR;  For the next instruction addressing mode.
            ld     DATA_TABLE(nb_byte),read_data       ; Save the received data.
            dec    nb_byte             ; Number of bytes to be read.
            if     [ SETZ ] {          ; If the last byte has been read.
                   call    GEN_STOP    ; Generate STOP condition in I2C protocol.
                   ld      STAT_EEP,#EEP_FREE_MASK     ; Indicates to the caller than
                                                        ;EEPROM is OK and FREE.
            } else {
                   call    GEN_ACK     ; ACK pulse generation and force SDA line
                                       ; to 0.
                   INIT_TRANS_READ
            } ;-- End of else.
      } else {
            call   TEST_ACK       ; ACK pulse generation and test EEPROM
                                  ;response..
```

Appendix A. EEPROM I²C-bus Manager Routine (Continued)

```
            if     [ STAT_EEP == #NO_ACK ] {          ; If no acknowledge from EEPROM.
                   call    GEN_STOP                   ; Stop generation.
                   or      STAT_EEP,#EEP_FREE_MASK    ; Indicates to the caller than
                                                      ; EEPROM is free.
                   switch [ STAT_TRANS_SPI ] {
                   case    #T_ADD_SLAVE:
;—- The slave address has been transferred.
                           if     [ STAT_EEP == #VERIF_ON ] {
;—- The slave address has been transmitted for a verif operation.
                           } else {
;—- The slave address has been transmitted for a write or a random read operation.
                                  ld     STAT_TRANS_SPI,#T_ADD_EEP
                                                      ; Transfer of the address of
                                                      ; the EEPROM operation.
                                  spp    #0
                                  ld     SPI_TAMP,EEP_ADD     ; To transfer the read
                                                              ;or write address.
                   case    #T_ADD_EEP:
;—- The write or random read address has been transmitted.
] {
;—- The random read addresss has been transmitted.
                                  call   INIT_START_I2C       ; A start condition is
                                                              ; necessary here.
                                  ld     STAT_TRANS_SPI,#T_ADD_SLAVE
                                                              ; The slave address must
                                                              ; be transmitted again.
                                  ld     SPI_TAMP,#ADD_EEP_R  ; EEPROM address in read
                                                              ; mode.
                                  ld     STAT_EEP,#VERIF_ON   ; The next sequence is
                                                              ; the same than verif
                                                              ; sequence.
;—- The write address has been transmitted.
                           } else {
                                  spp    #0
                                  ld     STAT_TRANS_SPI,#TRANS_WR_DATA
                                                      ; Initialisation for transfer
                                                      ; of data to be written.
                                  ld     write_data,DATA_TABLE(nb_byte)
                                                      ; The first data to programm.
                                  ld     SPI_TAMP,write_data
                           } ;—- End of else.

                   case    #TRANS_WR_DATA:

;—- The data to be written has been transmitted.
```

Appendix A. EEPROM I²C-bus Manager Routine (Continued)

```
                        spp    #0
                        dec    nb_byte        ; Number of bytes to write.
                        if     [ CLZ ] {      ; If the last byte has not yet
                                              ; been written.
                               ld     write_data,DATA_TABLE(nb_byte)
                               ld     SPI_TAMP,write_data
                        } else {              ; If all data have been programmed.
                                              ; Write sequence is finished.
                               call   PROG_DELAY    ; Initialise watch dog timer
                                                    ;to generate a 5 ms delay.
                               call   GEN_STOP      ; STOP condition generation.
                        } ;—- End of else.
                  } ;—- End of switch.
            } ;—- End of else.
      } ;—- End of else.
      popuw RPP
      ld    SPIDR,SPI_TAMP                    ; Data to transmit via SPI.
      popu  PPR
      iret
;*****************************************************************************
;     PROG_DELAY   Initialize the watchdog-timer  to generate the delay
;                  necessary for programmation.
;
;*****************************************************************************
proc  PROG_DELAY [ PPR ] {
      pushuw       RPP
      spp   #WDT_PG
      srp   # (15 * 2)                  ; To access in paged registers with r.
      ld    wcr,#wden                   ; watch dog mode disabled, no wait states.
      clr   wdtpr                       ; To have 333 ns (with system clock = 12 MHz)
                                        ; in minimum count,
                                        ; prescaler = 0.
      ldw   wdtr,#15015                 ; 15015 * 333 ns = 5 ms.

                  while    [ CLZ ] {
            addw  wdtr,#15015           ; 5 ms delay is multiplied by
                                        ; the number of bytes to programm.
            dec   MEMO_NB_BYTE
      }
      or    wdtcr,#( stsp | sc )        ; Timer starts down counting.

                                        ; Single mode.

                                        ; Watch Dog disabled.
```

Appendix A. EEPROM I²C-bus Manager Routine (Continued)

```
                                        ; Input section disabled.
                                        ; Output disabled.
                                        ; Interrupt A0 on Timer EOC.
                                        ; Top Level Interrupt on SW TRAP.
      popuw RPP
}
;*****************************************************************************
;     TEMPO:      Interrupt service routine of the watchdog timer end of count.
;                 This interrupt is connected to the A0 channel in the ST9.
;
;*****************************************************************************
TEMPO:
      pushu PPR
      spp   #0
      and   EIPR,#~ipa0m                ; Reset of WD/Timer EOC interrupt pending
                                        ; bit.
      or    STAT_EEP,#EEP_FREE_MASK     ; Write sequence is finished.
      popu  PPR
      iret
```

Appendix B. Examples of Calling Programs

```
.title“ Main example for EEPPROM manager call            January 24 1990 ”


.extern            IT_END_TRANS, EEP_MAN, TEMPO
.global            RESET_START
;*************************
; Module Macro Definitions.
;*************************
.library    “c:\st9\inc\bitmacro.inc”    ; change if required
.mcall      attbit

;***********************
; Register declarations.
;***********************
EEP_ADD            =       R0     ; Operation address in the EEPROM.
WRITE_DATA         =       R1     ; Data to be programmed in the EEPROM.
write_data         =       r1
READ_DATA          =       R2     ; Data which has been read in the EEPROM.
read_data          =       r2
EEP
FUNCT =     R3
STAT_EEP           =       R4
STAT_TRANS_SPI     =       R5
NB_BYTE            =       R6
nb_byte            =       r6
CPT_DELAY          =       RR8
DATA_TABLE         =       31     ; The real beginning of the table to store
                                  ; data is R20h.
;                  01Fh

;***************************
; INTERRUPT VECTOR ADDRESSES.
;***************************
ORE_IT_VECT :=     00h     ; Core interrupt vectors
EXT_IT_VECT :=     20h     ; External interrupt vectors

;******************
; START of PROGRAM.
;******************
START_PROG         :=      100h   ; start address program


;*******************
; STACK Declaration.
;*******************
SSTACK      :=     ( 14 * 16 ) - 1; System stack address group D C
USTACK      :=     ( 12 * 16 ) - 1; User stack address group B
```

Appendix B. Examples of Calling Programs (Continued)

```
;*****************************************
; Declaration of the interrupt vectors table.
;*****************************************
      .text                              ; start of program
            .org   CORE_IT_VECT          ; Core interrupt vector
                                         ; *********************
                   .word   RESET_START   ; power on interrupt vector
            .org   EXT_IT_VECT           ; External interrupt vector
                                         ; ************************
                   .word   TEMPO         ; Channel A0 for Watchdog Timer.
                   .word   0000          ; Channel A1 not used/
                   .word   IT_END_TRANS  ; Channel B0 for SPI.

;*********************
; Start of main module.
;*********************
            .org   START_PROG            ; start of code
RESET_START:
      spp   #0
      ld    MODER,#( sspm | uspm| div2m ); CLOCK MODE REGISTER
                                         ; internal stack
                                         ; no precaling
                                         ; external clock divided by 2
;—- SPI and related I/O initialization.
      spp   #P2C_PG                      ; P21 = SDI: IN/TRI/TTL.
      ld    P2C0R,#00001110b             ; P22 = SCK: AF/OD/TTL.
      ld    P2C1R,#11111101b             ; P23 = SDO: AF/OD/TTL.
      ld    P2C2R,#00001110b             ; Others = OUT/PP/TTL.
      spp   #0
      ld    CICR,#( gcenm | iamm | cplm ); CENTRAL INTERRUPT CONTROL REGISTER
                                         ; priority level = 7
                                         ; Nested Arbitration mode
                                         ; disable interrupt
                                         ; enable counters
      spp   #0

      srp   #(15 * 2)                    ; To access page 0 registers
      clr   eipr                         ; Disable all the external interrupt
                                         ; pending bits.
      nop                                ; See WARNING (Technical Manual - Chapter 8)
      ld    eivr,#EXT_IT_VECT            ; External interrupt vector.
                                         ; IAOS - TLIS = 00 = ...
      ld    eiplr,#0FBh                  ; Priority level for group INTA0
                                         ; INTA1 = 6, 7.
```

Appendix B. Examples of Calling Programs (Continued)

```
ld    eimr,#01                          ; Unmask Interrupt A0 channel
                                        ; (WDT End Of Count).
                                        ; ( SPI EOT ).
                                        ; bit is active.
      clr   FLAGR                       ; init flag
      ld    SSPLR,#SSTACK + 1           ; load system stack pointer
      ld    USPLR,#USTACK + 1           ; load user stack pointer
      ld    STAT_EEP,#EEP_FREE_MASK     ; EEPROM is free, no function in service.
      ei
;*************************************************************************
;Exemple of call to the EEPROM manager to programm 4 bytes from the address 010h.
;*************************************************************************
begin_write::
      ld    EEP_FUNCT,#WRITE_FUNCT      ; Function to be executed by the
                                        ; EEPROM manager.
      ld    EEP_ADD,#010h               ; 1st address to be programmed.
      ld    NB_BYTE,#4                  ; Number of bytes to program.
      ld    R#(DATA_TABLE+4),#78h ; 1st data to programm.

      ld    R#(DATA_TABLE+3),#49h ; 2nd data to programm.
      ld    R#(DATA_TABLE+2),#10h ; 3rd data to programm.
      ld    R#(DATA_TABLE+1),#94h ; 4th data to programm.
      call  EEP_MAN
      attbit STAT_EEP,#EEP_FREE_MASK    ; Wait for end of WRITE procedure
                                        ; (programming delay also).
                                        ; by the ST9.
      nop                               ; To replace by a JR instruction
                                        ; under SDBST9 for DEBUG.
      nop
;*************************************************************************
;Example of call to the EEPROM manager to read 6 bytes from the address 0fh.
;This can be a verification of the last programmation.
;*************************************************************************
begin_read::
      ld    EEP_FUNCT,#READ_FUNCT ; Function to be executed by the
                                        ; EEPROM manager.
      ld    EEP_ADD,#0fh                ; Read address in EEPROM.
      ld    NB_BYTE,#6                  ; Number of data to be read.
      call  EEP_MAN
      attbit STAT_EEP,#EEP_FREE_MASK    ; Wait for end of read procedure.
                                        ; Here some instructions
                                        ; could be executed by the ST9.
end_read::
      jr    end_read
```

Appendix C. Module Macro Definitions

```
 title“ BITMACRO.INC                                        05 December 1989 ”
;*****************************************************************************
;*****************************************************************************
; BITMACRO: Macro file allowing bit test like PSEUDO_MACROS programmation,
; User must declare the macro used in his ST9 source file like the following
example


;.library   “c:\st9\inc\bitmacro.inc”
;.mcall            ifbit, ifnobit, and so on
;*****************************************************************************
;*****************************************************************************
;*****************************************************************************
;—- macro-instruction IFBIT: test if a bit is 1.
;    Parameters: - destination: All addressing mode allowed by
;                                  the”tm*instruction.
;                              - mask selecting the bit to be tested.
;                                    ex: 00000010b for bit 1 test.
;
; !!! DO not forget the “}” after instructions executed when the condition is
;      TRUE.
; application example
;     ifbit dest,mask
;           ...
;           ...
;     }
.macroifbit dest,mask
      tm    dest,mask
      if    [ CLZ ] {             ; The bit is set to 1.
.endm
;*****************************************************************************
;*****************************************************************************
;     macro-instruction WHILEBIT: DO WHILE bit is 1.
;    Parameters: - destination: All addressing mode used for “tm” instruction.
;                              - mask selecting the bit to be tested.
;                                    ex: 00000010b to test bit 1.
;
; application example
      do {
;           ...
;           ...
      whilebit     dest,mask
.macrowhilebit     dest,mask
      tm    dest,mask
      }     while  [ CLZ ]        ; The bit is set to 1.
.endm
```

Appendix C. Module Macro Definitions (Continued)

```
;*****************************************************************************
;*****************************************************************************
;—- macro-instruction IFNOBIT: test if a bit is 0.
;    Parameters: - destination: All the addressing mode used for the “tm”
;                                  instruction.
;                              - mask selecting the bit to be tested.
;                                    ex: 00000010b to test bit 1.
;
; !!! Do not forget the “}” after instructions executed when the condition is
;      TRUE.
; application example
;     ifnobit      dest,mak
;           ...
;           ...
;     }
.macroifnobit     dest,mask
      tm    dest,mask
      if    [ SETZ ] {            ; the bit is set to 1.
.endm
;*****************************************************************************
;*****************************************************************************
;—- macro-instruction WHILENOBIT: DO WHILE bit = 0.
;    Parameters: - destination: All the addressing mode used for the “tm”
;                                  instruction.
;                              - mask selecting the bit to be tested.
;                                    ex: 00000010b to test bit 1.
;
; application example
      do    {
;           ...
;           ...
      whilenobit   dest,mak
.macrowhilenobit   dest,mask
      tm    dest,mask
      }     while  [ SETZ ]; The bit is set to 1.
.endm
;*****************************************************************************
;*****************************************************************************
;—- WAITBIT: waiting for a bit to be 1.
;    Parameters: - destination: All the addressing mode used for “tm”
;                    instruction.
;                              - mask selecting the bit to be tested.
;                                    ex: 00000010b to test bit 1.
;
.macrowaitbit     dest,mask
      ********
      do {
           tm     dest,mask
      } while [ SETZ ]            ; WAITING for bit = 1.
.endm
```

Appendix C. Module Macro Definitions (Continued)

```
;*************************************************************************
;*************************************************************************
WAITNOBIT: waiting for a bit to be a 0.
;    Parameters: - destination: All the addressing mode used with the “tm”
;                  instruction.
;                          - mask selecting the bit to be testesd.
;                                ex: 00000010b to test bit 1.
;
.macrowaitnobit    dest,mask
      do {
            tm     dest,mask
      } while [ CLZ ]                   ; WAITING for the bit = 0.
.endm
;*************************************************************************
```