Springer Series in

ADVANCED MICROELECTRONICS 18

Springer Series in

ADVANCED MICROELECTRONICS

The Springer Series in Advanced Microelectronics provides systematic information on all the topics relevant for the design, processing, and manufacturing of microelectronic devices. The books, each prepared by leading researchers or engineers in their fields, cover the basic and advanced aspects of topics such as wafer processing, materials, device design, device technologies, circuit design, VLSI implementation, and subsystem technology. The series forms a bridge between physics and engineering and the volumes will appeal to practicing engineers as well as research scientists.

18 **Microcontrollers in Practice**
By I. Susnea and M. Mitescu

19 **Gettering Defects in Semiconductors**
By V.A. Perevoshchikov and V.D. Skoupov

20 **Low Power VCO Design in CMOS**
By M. Tiebout

Volumes 1–17 are listed at the end of the book.

M. Mitescu I. Susnea

Microcontrollers in Practice

With 117 Figures, 34 Tables and CD-Rom

Springer

Marian Mitescu
Razoare Street 2
6200 Galati
Romania
mitescum@yahoo.com

Ioan Susnea
Brailei Street 179
800578 Galati
Romania
isusnea@yahoo.com

ISSN 1437-0387

ISBN-10 3-540-25301-7 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-25301-3 Springer Berlin Heidelberg New York

Library of Congress Control Number: 2005927903

Springer is a part of Springer Science+Business Media.

springeronline.com

Typesetting and production:
LE-TEX Jelonek, Schmidt & Vöckler GbR, Leipzig, Germany
Cover design: *design & production* GmbH, Heidelberg

Printed on acid-free paper SPIN: 11327721 57/3141/YL - 5 4 3 2 1 0

This book is dedicated to the great inventor John W. Halpern,
who taught me that life can be re-invented,
and to Simona,
who taught me that it's worth doing so.

Ioan Susnea

To my sons George Dan and Cosmin

Marian Mitescu

Preface

The Aim of this Book

The manufacturers of microcontrollers report annual sales of hundreds of millions of units. To support this massive market demand, they offer tens of thousands of pages of free, good quality technical documentation: data sheets, application notes, articles, etc.

The problem is that the more information is available on this subject, the harder it is to find the information useful for *you*. Therefore, the need for synthesis seems to be growing.

While the vast majority of the literature available is monographic, dedicated to a specific circuit, this book tries to emphasize that various microcontrollers have many common structural characteristics; in fact they are all implementations of the same concept. When starting with the big picture, it's easier to focus on details from time to time, than to build the big picture, starting from details.

Throughout this book, we present three different families of microcontrollers: HC11, AVR, and 8051 and we aim to make reading of this book more rewarding for the reader than reading three monographs dedicated to each of the above families. If you have ever studied one microcontroller, by reading this book you will discover that you already know a lot of things about *all* microcontrollers.

Another important aim of this book is to help the reader to make the small, yet decisive step between theory and practice. The book presents the design of three *development boards*, one for each microcontroller discussed, which can serve as platforms for a large number of experimental projects. The design examples presented demonstrate that, regardless of the microcontroller selected, and the complexity of the project, the software applications can be built according to the same general structure.

What's in this Book

The book is structured into three sections. Chapters 1–8 aim to create a detailed overview of microcontrollers, by presenting their subsystems *starting from a general functional block diagram, valid for most microcontrollers* on the market.

In each case, we describe the distinctive features of that specific subsystem for HC11, 8051 and AVR. This whole section has a more theoretical approach, but, even here, many practical examples are presented, mainly regarding the initializations required by each subsystem, or the particular use of the associated interrupts. The purpose of this section is to create a perspective that views the microcontroller as a set of resources, easy to identify and use.

Chapters 9–16 contain eight complete projects, described from the initial idea, to the printed circuit board and detailed software implementation. Here too, we permanently focus on the similarities between the microcontrollers discussed, from the hardware and software perspectives.

All chapters contain exercises, suggesting modifications or improvements of the examples in the book. Most exercises have solutions in the book; for the others the solutions can be found on the accompanying CD.

Finally, the appendices contain additional information intended to help the reader to fully understand all the aspects of the projects described in the previous sections. We chose to present these details separately in these appendices, in order to avoid fragmentation of the flow of the main text.

Who Should Read this Book

Most of the available books on microcontrollers are either "guides for idiots", assuming that the reader knows nothing on the subject, or "rocket science books" for a limited academic audience. Little is offered to the majority of readers that are in between.

This book is primarily aimed at students of technical universities, but can be rewarding reading for anyone having a reasonable dose of "technical common sense", like service technicians, hobbyists, inventors, etc. We assume that the reader knows the fundamentals about binary representation of numbers, Boolean algebra, logic functions, and logic circuits, and knows how to locate and read data sheets and other technical literature supplied by the manufacturers.

We also assume that he has the basic skills required to use a personal computer, and knows how to install a software application, how to view and create text files, etc. Thus, we can spare the space needed for explaining these notions for more substantial projects.

However, eliminating puerile explanations doesn't mean that we give up all explanations.

Don't let yourself be intimidated by projects that seem very complex, like the "fuzzy logic temperature controller". Even the most complex projects presented in this book are built according to the same structure as the simplest, LED flashing type projects.

How to Read this Book

The chapters are intended to be self-contained, so that advanced users can, in principle, read the chapters individually as they choose.

However, the density of information requires permanent contact with practice in order to consolidate the knowledge acquired. We strongly recommend downloading and installing the software tools needed to test the examples and exercises presented in the book. For your convenience, all the source files for the examples and exercises presented in the book have been included on the accompanying CD.

This CD also contains the schematic files and the PCB layout drawings for the main projects described in the book. In our opinion, PCB design skills are equally important as good knowledge of the software, and therefore we recommend that you download and install the freeware version of the Eagle™ layout editor, from CadSoft, so that you can view and edit the projects included.

The information in this book is not intended to replace the manufacturers' data sheets; therefore it is a good idea to keep at hand the data sheets for the microcontrollers discussed here: 68HC11F1, AT90S8535, and AT89C51.

If you encounter new terms, try using the glossary of terms included in the appendices, which contains short definitions for the most common terms used in the book.

No special conventions have been used in writing this book. The only notation worth mentioning here is that the names of active LOW signals are written using a backslash as terminator, e. g. RD\, WR\, etc.

Disclaimer

The information in this book is for educational purposes only. Although we have made every effort to provide accurate information, the examples in this book should not be interpreted as a promise or a guarantee that you will be able to reach a certain level of knowledge or skills by reading this material.

The author and publisher shall in no event be held liable to any party for any direct, indirect, punitive, special, incidental or other consequential damages arising directly or indirectly from any use of this material, which is provided "as is", and without warranties.

All internet links indicated are for information purposes only and are not warranted for content, accuracy or any other implied or explicit purpose.

Contents

1

Resources of Microcontrollers

1.1 In this Chapter

This chapter is a presentation of the main subsystems of microcontrollers, seen as *resources*, organized according to one of the fundamental *architectures*: Von Neumann and Harvard. It also contains a description of the internal *CPU registers*, the general structure of a *peripheral interface*, and an overview of the *interrupt system.*

1.2 Microcontroller Architectures

A *microcontroller* is a structure that integrates in a single chip a microprocessor, a certain amount of memory, and a number of peripheral interfaces.

The Central Processing Unit (CPU) is connected to the other subsystems of the microcontroller by means of *the address and data buses*. Depending on how the CPU accesses the program memory, there are two possible architectures for microcontrollers, called Von Neumann, and Harvard.

Figure 1.1 shows the structure of a computer with Von Neumann architecture, where all the resources, including program memory, data memory, and I/O registers, are connected to the CPU by means of a unique address and data bus.

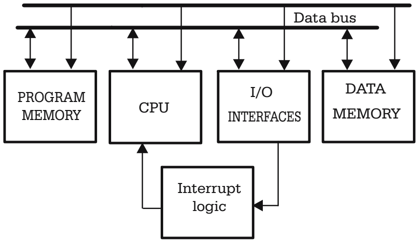


Fig. 1.1. Block diagram of Von Neumann architecture

A typical microcontroller having Von Neumann architecture is 68HC11 from Motorola. In HC11, all resources are identified by unique addresses in the same address space, and can be accessed using the same instructions. For example, in case of the instruction:

```
LDAA     <address>     ;load accumulator a from <address>
```

the operand indicated by the label <address> can be any of the microcontroller's resources, from I/O ports, to ROM constants. This way of accessing resources allows the existence of complex instructions like this:

```
ASL      35,x          ;arithmetic shift left the memory
                       ;location with the address
                       ;obtained by adding 35 to the
                       ;index register X.
```

Therefore, the Von Neumann microcontrollers tend to have a large instruction set, including some really complex instructions. This is the reason why computers having the Von Neumann architecture are often called CISC, or Complex Instruction Set Computers.

The main disadvantage of this architecture is that the more complex the instruction, the longer it takes to fetch, decode, execute it, and store the result. The instruction in the above example takes six machine cycles to execute, while the instruction for integer divide, IDIV, needs no less than 41 machine cycles to complete.

The Harvard architecture was created to increase the overall speed of computers in the early years, when very slow magnetic core memory was used to store the program. It includes an additional, separate bus to access the program memory (refer to Fig. 1.2).

The presence of the second bus makes the following things possible:

- While an instruction is executed, the next instruction can be fetched from the program memory. This technique is called *pipelining* and brings a significant increase of computer speed.
- The program memory can be organized in words of different size from, and usually larger than, the data memory. Wider instructions mean a greater data flow to the CPU, and therefore the overall speed is higher.

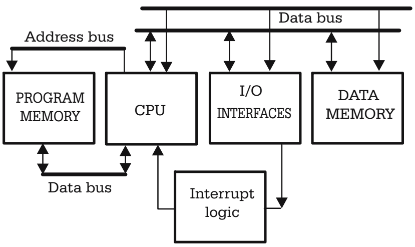


Fig. 1.2. Block diagram of Harvard architecture

Such architecture, along with reducing and optimizing the instruction set, mean that most instructions execute in a single machine cycle. Since the Harvard architecture is often accompanied by the reduction of the size and complexity of the instruction set, computers with this architecture are also called Reduced Instruction Set Computers (RISC). For example, some PIC microcontrollers have an instruction set of only 35 instructions, compared to more than 100 for HC11. The speed increase is even higher.

The separate bus for the program memory makes the access of the program to constants (such as tables, strings, etc.) located in ROM more complicated and more restrictive. For example, some PIC microcontrollers have the program memory organized in 14-bit wide *words*, which makes locating and accessing a constant presented as a *byte* possible only by embedding the constant in a special instruction. For this purpose, the instruction “RETLW k” (Return from subprogram with constant k in register W) has been provided.

The AVR microcontrollers have the program memory organized into 16-bit words, which makes the task of accessing constants in program memory easier, because each 16-bit word can store two 8-bit constants. A special instruction LPM (Load from Program Memory) allows access to ROM constants.

1.3 The Memory Map

From the programmer’s point of view, a microcontroller is a set of resources. Each resource is identified by one or more *addresses* in an *address space*. For example, the 68HC11E9 microcontroller has its internal RAM memory organized as 512 locations, having addresses in the range $0000–$01FF, the ROM memory occupies the addresses in the range $D000–$FFFF (12288 locations), and the I/O register block takes the address form $1000–$103F (64 locations).

The *memory map* is a graphic representation of how the resources are associated with addresses (see Fig. 1.3 for an example of a memory map).

Obviously, not all addresses are related to existing resources – in some cases it is possible to add external memory or I/O devices, to which we must allocate distinct addresses in the address space.

Normally, the memory map is determined by the hardware structure formed by the microcontroller and the external devices (if any), and cannot be dynamically modified during the execution of a program.

However, there are situations when, by writing into some special configuration registers, the user can disable resources (such as the internal ROM memory, or the EEPROM memory) or can relocate resources in a different area of the address space. But even in these cases, the access to the configuration registers is restricted, and the modification becomes effective after the next RESET.

Figures 1.3 and 1.4 show the memory maps for a microcontroller with Von Neumann architecture, MC68HC11E9, operating in single-chip mode, and for a RISC microcontroller, the AVR AT90S8535.

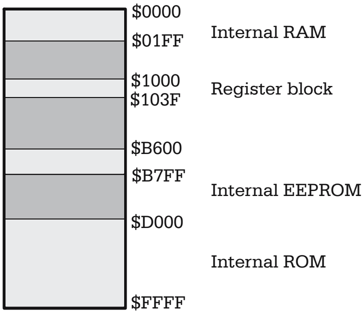


Fig. 1.3. Memory map for 68HC11E9 operating in single-chip mode

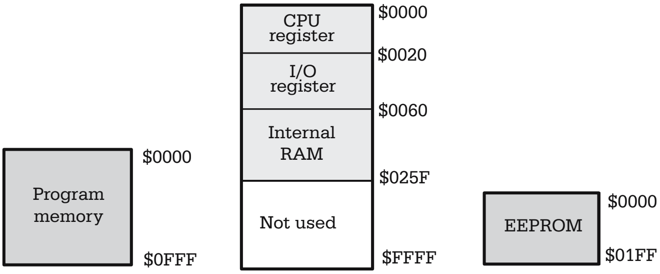


Fig. 1.4. Memory map for AT90S8515 operating in single-chip mode

Note, for the AVR microcontroller, the presence of three different address spaces, one for data memory and I/O registers, and two more for the program memory and the EEPROM.

The 8051 microcontrollers are considered to belong to the Harvard architecture, but they are CISC, and do not allow pipelining; therefore they look more like Von Neumann computers with the capability to access program memory, and data memory as different *pages*. The two distinct memory pages are accessed through *the same physical bus*, at different moments time in. Fig. 1.5 shows the memory map for an 8051 MCU operating in single-chip mode. There are two address spaces here too, one for the program memory and the other for data memory and special function registers.

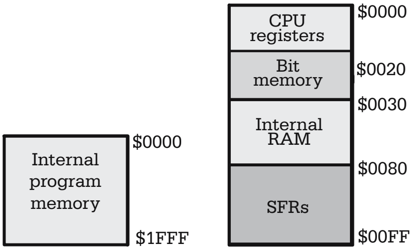


Fig. 1.5. Memory map for 8051 operating in single-chip mode

One unique feature of this microcontroller is the presence of a RAM area, located in the address range \$0020–\$002F, which is bit addressable, using special instructions. This artifice allows the release of RAM memory by assigning some Boolean variables to individual bits in this area, rather than using a whole byte for each variable, because 8051 is low on this resource: only 80 RAM locations are available for variables and stack.

Standard 8051 microcontrollers do not have internal EEPROM memory.

1.4 CPU Registers

The good thing about CPU registers is that they are part of the CPU, and an operand located in these registers is immediately available as input to the arithmetic and logic unit (ALU). Since the instructions having operands in the registers of the CPU are executed faster, the microcontrollers designed for higher speed tend to have more internal registers. While HC11 has only two accumulator registers, the AVR family has as many as 32 such registers.

1.4.1 The CPU Registers of HC11

HC11 has seven internal registers, plus the CPU status register, called the Condition Code Register (CCR).

The accumulator registers A and B are general-purpose 8-bit registers. They can be concatenated to form a 16-bit register called D, where A is the most significant byte, and B is the least significant byte. This feature creates a remarkable flexibility for 16-bit arithmetic operations.

The index registers X and Y are 16-bit registers, which can also be used as storage registers, 16-bit counters; and most important, they can store a 16-bit value, which, added with an 8-bit value contained in the instruction itself, form the effective address of the operand when using the indexed addressing mode.

The Stack Pointer (SP) register is a 16-bit register, that must be initialized by software with the *ending address* of a RAM memory area, called the *stack*. SP automatically decrements each time a byte is pushed to the stack, and increments when a byte is pulled from stack. Thus, SP always points to the first free location of the stack. The stack is affected in the following situations:

- During the execution of the instructions BSR, JSR (Branch or Jump to Subroutine), the return address is automatically pushed on to the stack and the SP is adjusted accordingly. The instruction RTS (Return from Subroutine) pulls this value from the stack and reloads it into the program counter.
- During the execution of push and pull type instructions, used to save and restore the contents of the CPU registers to the stack.
- During the execution of an interrupt, and when returning from an interrupt service routine upon the execution of the RTI (Return from Interrupt) instruction.

SP may be directly accessed by means of the LDS (load SP) and STS (Store SP) instructions or indirectly, using transfer instructions like TXS, TYS (Transfer X/Y to SP) or TSX, TSY (Transfer SP to X/Y).

The Program Counter (PC) register is a 16-bit register, that contains the address of the instruction following the instruction currently executed.

The Condition Code Register (CCR) is an 8-bit register with the following structure:

CCR	7	6	5	4	3	2	1	0
	S	X	H	I	N	Z	V	C
RESET	1	1	0	1	0	0	0	0

The bits C (Carry/Borrow), V (Overflow), Z (Zero), N (Negative) and H (Half Carry) are status bits, set or cleared according to the result of the arithmetic and logic instructions. Refer to the data sheet for details on how these bits are affected by each instruction.

The bits I (General Interrupt Mask), X (XIRQ Interrupt Mask), and S (Stop disable) are control bits used to enable/disable the interrupts, or the low-power operating mode. When I = 1 all maskable interrupts are disabled. X = 1 disables the non-maskable interrupt XIRQ, and S = 1 blocks the execution on the STOP instruction, which is treated like a NOP.

Some CCR bits (C, V, I) can be directly controlled by means of the instructions SEC (Set Carry), CLC (Clear Carry), SEV (Set Overflow Bit), CLV (Clear Overflow Bit), SEI (Set Interrupt Mask), and CLI (Clear Interrupt Mask). The CCR as a whole may be read or written using the instructions TPA (Transfer CCR to A) and TAP (Transfer A to CCR)

1.4.2 The CPU Registers of AVR

The CPU of the AVR microcontrollers has 32 general-purpose registers, called R0–R31. The register pairs R26–R27, R28–R29, R30–R31 can be concatenated to form the X, Y, Z , registers, which can be used for indirect addressing (R26 is XL – lower byte of X, R27 is XH – higher byte of X, R28 is YL, R29 is YH, R30 is ZL and R31 is ZH). The registers R16–R31 may be the destination of immediate addressed operands like LDI (Load Register Immediate) or CPI (Compare Immediate). *Unlike HC11, the CPU registers of AVR are present with distinct addresses in the memory map.*

The Program Counter (PC) has functions similar to those of the PC register of HC11. The difference is that the size of PC is not 16 bits, and is limited to the length required to address the program memory (in case of AT90S8515 only 12 bits are needed to address the 4K of program memory). PC is cleared at RESET.

The Stack Pointer (SP) has 16 bits, and is placed in the I/O register address space, which makes it accessible to the programmer only by means of the IN and OUT instructions, as two 8-bit registers SPH, and SPL.

The CPU status register is called SREG and has the following structure:

SREG	7	6	5	4	3	2	1	0
	I	T	H	S	V	N	Z	C
RESET	0	0	0	0	0	0	0	0

The meaning of the bits in SREG is slightly different from those of HC11:

The I bit – Global Interrupt Enable/Disable Bit – has an opposite action: when set to 1 the interrupts are enabled. The instructions that control this bit have the same mnemonic SEI (Set I bit) and CLI (Clear I bit).

T – Bit Copy Storage. The status of this bit can be modified by the instructions BST (Bit Store) and BLD (Bit Load), thus allowing the program to save the status of a specific bit from a general-purpose register, or transfer this information to a bit from another register. There is also a pair of conditional branch instructions which test this bit: BRTS (Branch if T bit is Set), and BRTC (Branch if T bit is Clear)

S –Sign Bit – It is the exclusive OR between N and V

The other bits in SREG (C, Z, N, V, H) have the same meaning described for HC11. The AVR microcontrollers have distinct SET–CLEAR instructions for each of the SREG bits.

1.4.3 The CPU Registers of 8051

The accumulator A is a general-purpose 8-bit register, used to store operands or results in more than a half of the instruction set of 8051.

The R0–R7 registers are 8-bit registers, similar to the registers R0–R31, described for the AVR family of microcontrollers. There are four sets (or *banks*) of such registers, selected by writing the bits [RS1:RS0] in the CPU status register PSW, described below.

The four sets of eight registers each occupy 32 addresses in the address space of data memory, at the addresses [0000h–0007h], [0008h–000Fh], [0010h–0017h], [0018h–001Fh] (refer to Fig. 1.4).

The accumulator B is another general-purpose 8-bit register, having functions similar to the R0–R7 registers. Besides that, the accumulator B is used to store one of the operands in the case of the arithmetic instructions MUL AB and DIV AB.

The Data Pointer Register (DPTR) is a 16-bit register, used for indirect addressing of operands, in a similar way to the X, Y, Z registers of AVR.

The Program Counter (PC) is a 16-bit register similar to the PC of HC11. PC is cleared at RESET, thus all programs start at the address 0000h.

The Stack Pointer (SP) has the following distinctive features, compared to HC11 and AVR:

- It is an 8-bit register, i.e. it can address a memory area of 256 bytes maximum. 8051 can only use the internal memory for the stack.

- Unlike HC11 and AVR where SP is initialized with an address at the end of RAM, and decrements with each byte pushed on to the stack, the SP of 8051 increments when data is added to the stack.
- For HC11 and AVR, SP points to the first free byte of the stack area. The SP of 8051 indicates the last occupied location of the stack. At RESET, SP is automatically initialized with 07h, hence the first byte pushed to the stack will occupy the location with the address 08h.

The Processor Status Word (PSW) is similar to CCR of HC11 or SREG of AVR, and has the following structure:

PSW	7	6	5	4	3	2	1	0
	CY	AC	F0	RS1	RS0	OV	–	P
RESET	0	0	0	0	0	0	0	0

The bits CY, AC and OV have similar functions to the bits C, H, and V of HC11 and AVR.

[RS1:RS0] – Register bank select bits

P – Parity bit. P = 1 if the accumulator contains an odd number of 1s, and P = 0 if the accumulator contains an even number of 1s. Thus the number of 1s in the accumulator plus P is always even. The bits PSW1 and PSW5 (F0) are uncommitted and may be used as general-purpose status flags.

1.5 The Peripheral Interfaces

Microcontrollers are designed to be embedded in larger systems, and therefore they must be able to interact with the outside world. This interaction is possible by means of the peripheral interfaces. The general structure of a peripheral interface is shown in Fig. 1.6.

Depending on the complexity of the specific circuits to be controlled by the program, any peripheral interface contains one or more control and status registers, and one or more data registers. These registers are normally located in the address space of the data memory, and are accessed as RAM locations.

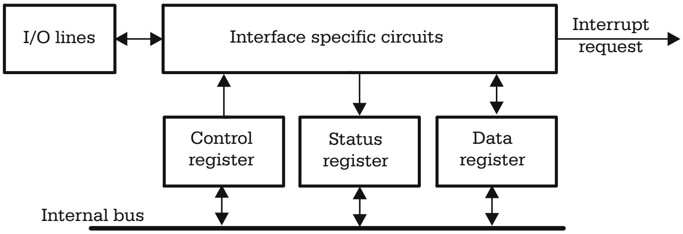


Fig. 1.6. Typical structure of a peripheral interface

The most common peripheral interfaces, present in almost all the usual microcontrollers, are:

- The I/O (Input/Output) ports.
- The asynchronous serial interface (SCI, UART)
- The synchronous serial interface (SPI)
- Several types of timers
- The analog to digital (A/D) converters

The following chapters contain detailed descriptions of each of the above peripheral interfaces. Most of the peripheral interfaces have a common feature, which is the capability to generate *interrupt requests* to the CPU, when some specific events occur. This feature is analyzed in the next paragraph.

1.6 The Interrupt System

1.6.1 General Description of the Interrupt System

Most of the events related to the peripheral interfaces, like the change of status of an input line, or reception of a character on the serial communication line, are asynchronous to the program running on the CPU. There are two possible ways to inform the CPU about these events:

- One solution is to write the program so that it periodically tests the status of some flags associated with the external events. This technique is called *polling*.
- The other solution is to interrupt the main program and execute a special subroutine when the external event occurs.

An *interrupt* is a mechanism that allows an external event to temporarily put on hold the normal execution of the program, forcing the execution of a specific subroutine. Once the interrupt service subroutine completes, the main program continues from the point where it was interrupted.

At the CPU level, this mechanism involves the following steps:

1. The identification of the interrupt source. This is automatically done by hardware.
2. Saving the current value of the PC register, thus providing a means to return from the interrupt service routine. The contents of PC are saved to the stack, and the operation is also done by hardware.
3. Then, the PC is loaded either with, or from, the address of a reserved memory area, called the *interrupt vector*. For each possible interrupt, a unique vector is assigned. *The interrupt vectors are hardwired and cannot be modified by the user.*
4. At the address of the interrupt vector, the program must contain either the address of the interrupt service routine (HC11 uses this technique) or an instruction for an unconditional jump to this routine (AVR and 8051 work this way).
5. The next step is the execution of the *Interrupt Service Routine (ISR)*. This is a program sequence similar to a subroutine, but ending with a special instruction

called Return from Interrupt (RTI, RETI). To make sure that the main program is continued exactly from the status it had in the moment when the interrupt occurred, it is crucial that all the CPU registers used by the interrupt service routine are saved at the beginning of the ISR, and restored before returning to the main program. Some microcontrollers, like the HC11 family, are provided with a hardware mechanism to save the whole CPU status, upon reception of an interrupt request. The status is restored by the instruction RTI (Return from Interrupt) before the actual return to the main program. *In all other cases, it is the user's responsibility to save and restore the CPU status in the interrupt service routine.*

6. The final step in handling an interrupt is the actual return to the main program. This is done by executing a RTI (RETI) instruction as mentioned before. When this instruction is encountered, the contents of PC, saved in step 2, are retrieved from the stack and restored, which is equivalent to a jump to the point where the program was interrupted. The process of returning from an ISR is similar to returning from a regular subroutine, but there is an important difference: the interrupt service routines cannot be interrupted, and therefore once an interrupt has been acknowledged, further interrupts are automatically disabled. They are re-enabled by the RTI (RETI) instruction. All interrupts occurring during the execution of an ISR are queued and will be handled one by one, once the ISR is serviced.

Important note. The stack is essential for the interrupt system. Both the PC and the CPU status are saved in the stack when handling interrupts. Therefore, the SP must be initialized by software before enabling the interrupts.

The interrupt service routine *must* save the CPU status and restore it before returning to the main program.

If two or more interrupt requests occur simultaneously, they are serviced in a predetermined order according to a hardwired priority. Refer to the data sheet for each microcontroller for details.

The software control over the interrupt system is exerted either globally, by enabling/disabling all the interrupts by means of specific instructions, or individually, by setting or clearing some control bits, called *interrupt masks*, associated with each interrupt. In other words, the process of generating an interrupt request is double conditioned, as shown in Fig. 1.7.

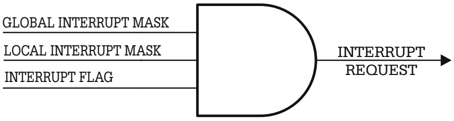


Fig. 1.7. Double conditioning of interrupt requests

The INTERRUPT FLAG is the actual interrupt source, and, usually, is a flip-flop set by the external event. This bit is, in most cases, accessible for the program as a distinct bit in the status register of the peripheral interface.

The LOCAL INTERRUPT MASKS are control bits, located in the control registers of the interface. When set to 1 by software, the interrupts from that specific interface are enabled.

The GLOBAL INTERRUPT MASK is a bit located in the CPU status register (CCR, SREG, PSW) that enables or disables all interrupts.

In some cases, it is required that the CPU is informed immediately about certain important internal or external events, regardless of the status of the global interrupt mask. The solution to this problem is the *non-maskable interrupt*, which is unconditionally transmitted to the CPU. A special case of non-maskable interrupt can be considered the RESET. Basically, the behavior of the MCU at RESET is entirely similar to the process of identification and execution of a non-maskable interrupt.

1.6.2 Distinctive Features of the Interrupt System of HC11

The most important feature of the interrupt system of HC11 is that the CPU status is automatically saved by hardware, right after the PC is saved. This feature simplifies the programmer's work, but it wastes time saving and restoring *all* CPU registers. In most cases, the interrupt service routine does not use *all* the CPU registers, but needs to be executed as fast as possible.

The global control of the interrupt system is performed by means of the I bit in the CCR register. When $I = 1$, all maskable interrupts are disabled. When $I = 0$, the interrupts coming from a specific peripheral interface are enabled if the local mask associated with that interface is set to 1. The I bit can be controlled by means of the instructions SEI (Set Interrupt Mask) equivalent to Disable Interrupts, and CLI (Clear Interrupt Mask), equivalent to Enable Interrupts.

Besides the maskable interrupts, HC11 has three non-maskable interrupts, without counting the three possible RESET conditions (activation of the external RESET line, clock monitor fail reset, and watchdog reset). These are: XIRQ – External non-maskable interrupt, ILLOP – Illegal opcode trap, and SWI – software interrupt.

The XIRQ interrupt is generated upon detection of a logic level LOW on the XIRQ input line, if the X bit in CCR is clear. The X bit acts similarly to I, but it only affects the XIRQ interrupt, and it is not affected by the SEI and CLI instructions. The only way the user can alter the status of this bit is by the TAP (Transfer A to CCR) instruction.

The illegal opcode trap is an internal interrupt generated when an unknown opcode is fetched and decoded into the CPU.

A software interrupt is generated when the instruction SWI is decoded. This is useful for defining breakpoints in a program for debug purposes.

The priority of the interrupts is hardwired. However, it is possible to define one of the interrupts as the highest priority non-maskable interrupt. For this purpose, the bits [PSEL3–PSEL0] (Priority Select bits) in register HPRIO (Highest Priority Interrupt Register) code the interrupt with the highest priority.

The vector area for HC11 is located at the end of the address space between the addresses $FFC0–$FFFF. See the data sheets for the list of exact addresses assigned to each interrupt vector.

1.6.3 Distinctive Features of the Interrupt System of AVR

There are a few differences between the interrupt system of AT90S8535 and that of HC11. They are listed below:

- The interrupt vector does not contain *the address* of the interrupt service routine, but a jump instruction to that routine.
- The vector area is located at the beginning of the program memory address space, between the addresses $0000 and $0010.
- There are no non-maskable interrupts besides RESET.
- The I bit in SREG acts differently, compared to HC11: when I = 1, the interrupts are enabled.
- There is no equivalent to the HPRIO register, and no other means to modify the hardwired relative priority of interrupts.

1.6.4 Distinctive Features of the Interrupt System of 8051

8051 has only five possible interrupt sources, compared to 16 for AVR, and 18 for HC11. The vectors are placed at the beginning of the program memory address space and must be initialized by the software to contain a jump to the interrupt service routine.

The interrupts are enabled and disabled according to the same principles described for HC11 and AVR. The difference is that all the control bits associated with the interrupt system are placed in a Special Function Register (SFR) called IE (Interrupt Enable register) located at the address A8h. This register contains the global interrupt control bit, called in this case EA (Enable All interrupts), and bits to enable each individual interrupt.

One interesting distinctive feature of the interrupt system of 8051 is the possibility to choose between two priority levels (low and high) for each interrupt. To this purpose, a special register called IP (Interrupt Priority register) contains a bit associated with each interrupt. When the priority bit is 0, the associated interrupt has a low priority level, and when the priority bit is 1, the interrupt has high priority. *Unlike HC11 and AVR, for 8051 a high-priority interrupt can interrupt a low-priority interrupt service routine.*

8051 does not save the CPU status automatically, therefore the interrupt service routine *must* save and restore the registers used, including PSW.

1.7 Expanding the Resaurces of Microcontrollers

In many cases it is possible that the internal resources of a microcontroller are insufficient for certain applications. A typical example is when the number of variables used to store data exceeds the capacity of the internal RAM memory. The obvious solution to these situations is to add external components by creating an *expanded microcontroller structure*.

The disadvantage of this solution is that a significant number of the available I/O lines are used to create the external bus for accessing the new resources, and are no longer available for normal I/O operations. Note that not all microcontrollers can operate with an external bus. The following paragraphs describe how to create and use expanded microcontroller structures.

1.7.1 HC11 Operating with External Bus

The HC11 microcontrollers have two pins, called MODA and MODB, which control the operating mode. At RESET the status of these pins is read and, according to the result, the microcontroller selects one of the operating modes listed in Table 1.1.

As shown in Table 1.1, there are four possible operating modes, among which two are *special* and the other two are *normal*. In the *special bootstrap* mode, a small ROM memory area becomes visible in the memory map. This ROM contains a short program, called a bootloader, which is executed after RESET, allowing the user to load and run a program in the internal RAM. This is useful, for example, to program the internal ROM memory. See App. A.5 for details on how to do this.

The special test operating mode is destined for factory testing, and will not be discussed in this book.

In the expanded operating mode, some of the MCU I/O lines are used to implement the external bus. In some cases, to reduce the number of I/O lines used for this purpose, the external data bus is multiplexed with the lower byte of the address bus.

Demultiplexing requires an external latch, and a signal AS (Address Strobe), generated by the MCU, as shown in Fig. 1.8.

Besides AS, the MCU generates the signal R/W\ (Read/Write) to specify the direction of the transfer on the data bus. A detailed example of using HC11 in expanded mode is presented in App. A.4.

From the programmer's point of view, when using HC11 in expanded operating mode, the following details must be considered:

- All the MCU resources, except the I/O lines used to implement the external bus, are still available, and have the same addresses.
- The internal ROM can be disabled by clearing the ROMON bit in the CONFIG register.

Table 1.1. Selection of the operating mode of HC11

MODB	MODA	Operating mode
0	0	Special bootstrap
0	1	Special test
1	0	Single chip
1	1	Expanded

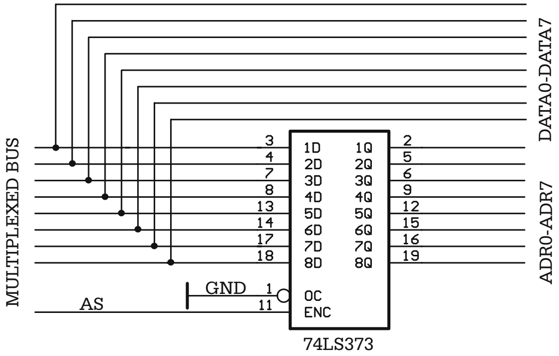


Fig. 1.8. Demultiplexing the external bus

1.7.2 AT90S8515 Operating with External Bus

From the two distinct buses of the AVR microcontrollers, only the bus used for the data memory can be, *in some cases*, expanded to connect external RAM, or RAM – like external devices. AT90S8515 uses the lines of port A to multiplex the data bus with the lower order address bus, and port C for the high order address bus. The signal that strobes the address into the external latch is called ALE (Address Latch Enable). The direction of the transfer is indicated by two signals RD\ (Read) and WR\ (Write) – both active LOW. The circuit for demultiplexing the bus is identical to the one used by HC11, shown in Fig. 1.8.

The external bus is activated by software, writing 1 to the SRE bit (Static RAM Enable) in the register MCUCR (MCU Control Register). When using slower external memory, there is the possibility of including a WAIT cycle to each access to the external RAM. Writing 1 to the SRW (Static RAM Wait) bit of MCUCR enables this feature.

Appendix A.8 shows an example on how to use AT90S8515 with an external bus.

1.7.3 8051 Operating with External Bus

The external bus of 8051 is also multiplexed. Port P0 is used for the data bus and the low-order address bus, and port P2 implements the high-order address bus. The strobe signal for demultiplexing the bus is called ALE (Address latch Enable).

What is specific for 8051 is the capability to access on the same physical bus two pages of external memory: one for program memory, and the other for data memory. The two types of access are distinguished by means of a special signal, called PSEN\ (Program Store Enable), generated by the MCU, active LOW. When PSEN\ is active, the external logic must select an external ROM memory in order to provide program code to the MCU through the data bus. The access to the external RAM is controlled at the hardware level by the signals RD\ (Read) and WR\ (Write), generated by the microcontroller. An additional signal, called EA\ (External Access), active LOW, disables the internal ROM, and redirects all the accesses to the internal program memory (in the address range 0000h–0FFFh) to the external bus. Any access to program memory at addresses higher that 1000h are directed to the external bus regardless of the status of the EA\ input.

At the software level, accessing the external RAM is done by means of the special instruction MOVX (Move to or from external RAM).

Chapter 11 and App. A.12 contain detailed examples of using 8051 with an external bus.

1.8 Exercises

SX 1.1

Write a fragment of program to access a constant stored in the program memory of a MCU with the Von Neumann architecture (HC11).

Solution

Like all Von Neumann computers, HC11 uses a single bus to connect all the resources to the CPU, and has a single address space. Therefore, there is no difference between the ways it accesses constants located in the program memory or variables stored in RAM. The following code fragment does the job:

```
          .....
          LDAA      ROMTAB
          .....
ROMTAB    DB        $55       ;This defines $55 as
                              ;a constant located
                              ;in the program memory area
```

After the execution of the instruction LDAA ROMTAB, the contents of the accumulator A are identical to the contents of the memory location having the address ROMTAB ($55 in this case). The following example uses the X register as a pointer to a table of constants, ROMTAB. After the execution of this code, A contains $55, B equals $AA, and X contains the address ROMTAB.

```
          .....
          LDX       #ROMTAB   ;set X to point to ROMTAB
          LDAA      0,X       ;read one constant into A
          LDAB      1,X       ;read another constant into B
          .....
ROMTAB    DB        $55       ;This defines $55 as a
                              ;constant located in the
                              ;program memory area
          DB        $AA
```

SX 1.2

Write a code fragment to access a constant located in the program memory of a microcontroller having the Harvard architecture (AT90S8515 AVR)

Solution

The AVR microcontrollers use the special instruction LPM to access constants stored in the program memory. LPM copies to R0 the contents of the program memory location with the address specified by the Z register. The problem is that AVR microcontrollers have the program memory organized in 16-bit words and the assembler assigns a word address to any label found in the code section.

LPM interprets the contents of the register Z in the following way: the most significant 15 bits of Z represent the address of the 16-bit word, and the least significant bit is the address of the byte within the 16-bit word: LSB(Z) = 0 indicates the least significant byte, and when LSB(Z) = 1 the most significant byte is addressed.

```
          ....
          LDI       ZH,high(ROMTAB<<1)
          LDI       ZL,low(ROMTAB<<1)
          LPM                     ;read first byte
          MOV       R1,R0         ;save $34 to R1
          ADIW      ZL,1          ;increment Z
          LPM                     ;read second byte
          MOV       R2,R0         ;save $12 to R2
          ....
ROMTAB    DW        $1234
```

In the above example, the expression `(ROMTAB<<1)` means ROMTAB shifted one position to the left, and `high(ROMTAB<<1)` designates the most significant byte of the 16-bit value `(ROMTAB<<1)`. The first LPM reads in R0 the lower byte of the constant located at the address ROMTAB ($34) and, after incrementing the address, (ADIW ZL,1) the second LPM reads the most significant byte of the constant ($12).

SX 1.3

Write a code fragment to access a constant stored in the program memory of an 8051 microcontroller.

Solution

8051 uses a special instruction to read ROM constants. This is MOVC (Move Code) and has the following syntax:

```
MOVC A,@A+DPTR
```

The effect of this instruction is that the accumulator A is loaded with the contents of the program memory location having the address obtained by adding the 16-bit integer in DPTR with the unsigned byte in A.

```
          CLR       A              ;clear accumulator A
          MOV       DPTR,#ROMTAB   ;load DPTR with the address
                                   ;ROMTAB
          MOVC      A,@A+DPTR      ;get constant in A
          .....
          .....
ROMTAB:
          DB        55h            ;define the constant here
          .....
```

SX 1.4

Provide an example of interrupt vector initialization for HC11.

Solution

The HC11 interrupt vector is a 2-byte memory space, which must be initialized with the starting address of the interrupt service routine. For example, the interrupt vector associated with RESET (remember RESET is treated as a non-maskable interrupt) is located at the addresses $FFFE–$FFFF. At RESET, the MCU reads the two bytes from these addresses ($FFFE contains the most significant byte) and loads the resulting 16-bit value into PC.

```
MAIN      .......                  ;Program entry point at
                                   ;RESET
          .......
          ORG       $FFFE          ;store the value of the
                                   ;label MAIN at $FFFE-$FFFF
          DW        MAIN
```

SX 1.5

Provide an example of interrupt vector initialization for AVR.

Solution

The vector area of the AVR microcontrollers start at the address $0000 in the address space of the program memory. Each interrupt vector occupies a 16-bit word. Unlike HC11, the AVR simply loads the hardwired value of the vector into the PC when an interrupt is acknowledged. Initializing the vector consists in placing in the address of the vector an instruction of an unconditional jump to the interrupt service routine. For example, the interrupt vector associated with the analog comparator has the address $000C. The initialization of this vector is shown below:

```
          .......
          .ORG      $000C
          RJMP      ANA_COMP      ;unconditional jump to the
                                  ;interrupt handler
          .......
ANA_COMP:
          .......
          RETI
```

2

Using the Digital I/O Lines

2.1 In this Chapter

This chapter contains an overview of the parallel I/O subsystem of the HC11, AVR, and 8051 microcontrollers, from a hardware and software perspective.

2.2 Overview of the Parallel I/O System

The digital I/O lines are the simplest and most common way microcontrollers interact with the outside world. Figure 2.1 shows how digital input and output lines are connected to the internal bus of the MCU.

In case of an input line, the status of the MCU pin is transferred on the internal bus to the CPU by activating the *internal* signal RP (Read Port), generated during the execution of the instruction used to read the port.

An output line is associated with an internal latch, which can be written from the internal bus and the driver connected to the physical pin.

In practice, things are a bit more complicated. For economic and technological reasons, it is more convenient to assign two or more functions to each pin than to double the number of pins of the capsule. This is the reason why all microcontrollers extensively use *bi-directional* input/output lines. The simplest way to obtain a bi-directional line is to use an open drain driver for the output line. When the output

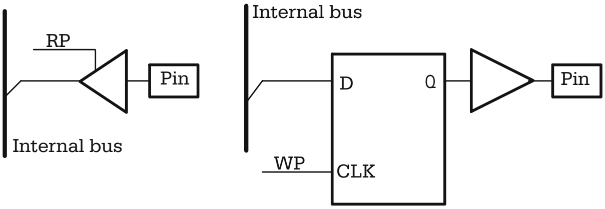


Fig. 2.1. The principle of accessing the input and output lines

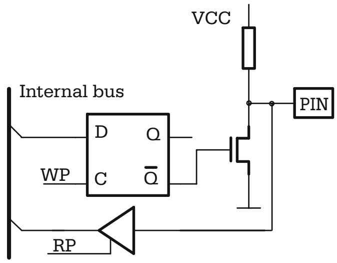


Fig. 2.2. Bi-directional I/O line with open drain output driver

transistor is blocked, an external device can control the line. Figure 2.2 shows the logic diagram of a bi-directional I/O line implemented according to this principle.

When the software writes 1 to the latch associated with the output line, the output transistor is blocked, and the line is turned into an input line. This solution is used by the 8051 microcontrollers.

Another way to do this is to associate to each bi-directional I/O line an additional latch, called a *direction latch*, which controls a tri-state output driver. When the direction latch is set to 1 by software, the I/O line is configured as an output line. A simplified logic diagram of an I/O line implemented according to this principle is shown in Fig. 2.3.

This method of controlling the direction of the I/O lines is characteristic of the microcontrollers belonging to the HC11 and AVR families.

Usually, the I/O lines are grouped into *8-bit ports*, which have individual addresses in the memory map. Similarly, the direction control bits are grouped into 8-bit registers, called Data Direction Registers (DDR), associated with each I/O port.

Note that the I/O port, along with the associated data direction register, form a structure similar to the general structure of the I/O peripheral interface, shown in Fig. 1.6, where the data register is the port itself, and the control register is the data direction register.

Often, the I/O lines have alternate functions in connection with some of the microcontroller's subsystems. For example, the asynchronous serial interface of the

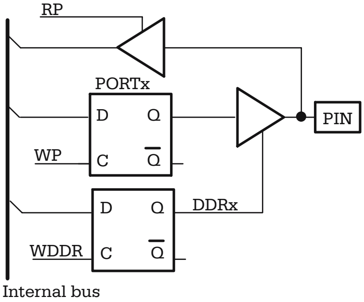


Fig. 2.3. Bi-directional I/O line with direction control

HC11 uses the lines of port D, and the timer subsystem can use the lines of port A. Normally, the external pins are automatically configured for the alternate function when the respective subsystem is enabled by software, overriding the settings in the data direction register, but this is not an absolute rule. Refer to the data sheet of each microcontroller for details on how a certain peripheral interface shares the MCU pins with the I/O ports.

2.3 Electrical Characteristics of the I/O Lines

One simple rule about interfacing a microcontroller is that any output line can safely drive one standard TTL load, and the voltage on any input may swing between the potential of GND and Vdd. In some particular cases, certain output lines can drive up to 20 mA each, but even in these cases, care must be taken that the total power dissipation does not exceed the limit specified by the manufacturer.

If the load current or voltage requirements are higher than the capabilities of the microcontroller, the solution is to use adequate *buffers* between the control circuit and the load. Figure 2.4 shows a possible way to connect a relay to an output line of the 68HC11F1 microcontroller.

The relay is driven by the CMOS circuit 40107, able to drain a current up to 100 mA. Note the presence of the pull-down resistor R1, which connects the MCU output to the ground. Its purpose is to maintain the line at the stable potential of ground during RESET.

This is required because, at RESET, all the I/O lines of the microcontroller are automatically configured as input lines. In the time interval between RESET and the moment when the I/O line is configured as output, the potential at the input of the 40107 is undetermined, and thus the relay can be unintentionally activated.

The value of the pull-down resistor is determined by the fact that it loads the MCU output line, which can supply a maximum current of 8–1 mA. A value of 10 K for this resistor loads the output line with 0.5 mA and provides a safe pull-down for the input of 40107.

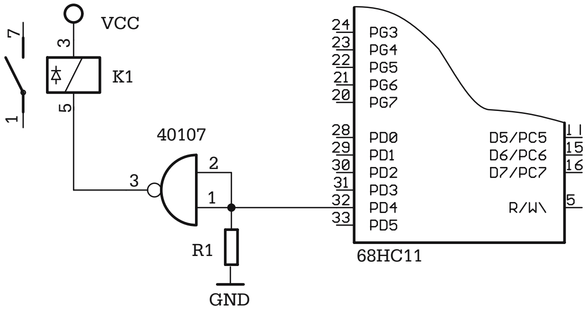


Fig. 2.4. Connecting a relay to an output line of a microcontroller

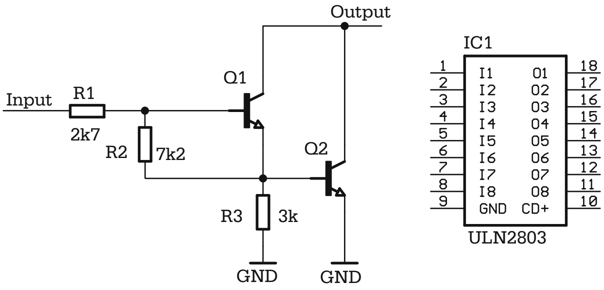


Fig. 2.5. ULN 2803 typical Darlington transistor interface for output lines

When higher current values are required, Darlington transistors are recommended to connect the load. Figure 2.5 shows the schematic of a Darlington driver, as implemented in the circuit ULN2803. This contains an array of eight such Darlington drivers, each being able to sink 500 mA current and to stand 50 V V_{CE} voltage.

As far as the input lines are concerned, the most common way of using them is to read the status of contacts (push buttons, relays, etc.). The interface circuits must provide precise logic levels for each possible status of the contact, and to eliminate, as much as possible, the mechanical vibrations of the contact. A typical interface circuit for digital input lines is presented in Fig. 2.6.

R1 is a pull-up resistor, intended to maintain a stable logic level high, when the contact is open. Typical values are in the range 4.7 K–10 K. The capacitor C1, and the Schmitt trigger 40106 are destined to eliminate the effect of the vibrations of the contact, and to create clean edges for the electrical signal applied to the MCU port.

Often, it is required that the MCU ground be separated from the ground of the input circuits. In this case, it is recommended that the input lines are optically isolated, using circuits similar to that presented in Fig. 2.7. In this circuit, the resistor R1 limits the current through the LED of the optocoupler OK1. When K1 is closed, the transistor of the optocoupler saturates, providing a logic level zero at the input line of the microcontroller. The role of R3 is to drain the charge accumulated in the parasite capacitance of the junction BE of the transistor, thus improving the rising edge of the signal in the collector. Typical values for R3 are in the range 330 K–470 K.

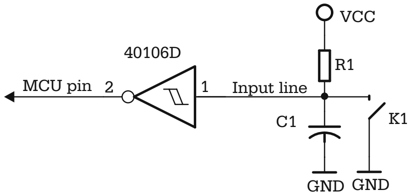


Fig. 2.6. Typical input circuit for reading relay contacts, push buttons, etc.

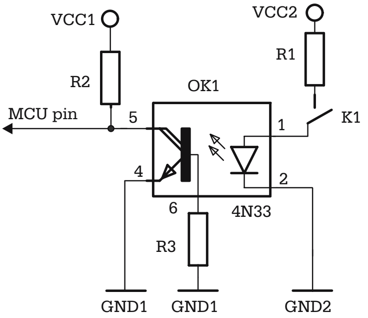


Fig. 2.7. Optically isolated input line

2.4 Controlling the I/O Lines by Software

The HC11 microcontrollers treat the I/O ports and the associated direction registers as memory locations. The software initialization of I/O lines consists in writing to the data direction register a byte having 1 in the positions corresponding to the output lines, like in this example:

```
INIT_PORTD    LDAA    #$30    ;configure PORTD, bit 4,5 as
              STAA    DDRD    ;output
```

For HC11, the software can read a port configured as output. In this situation, the last value written to the port is read. Data written to an *input port* is not visible to the MCU pins until the port is reconfigured as an output port.

The 8051 microcontrollers have two major features, as far as the I/O ports are concerned: there are no direction registers, and the ports are bit-addressable.

All output pins of 8051 have open drain drivers, as shown in Fig. 2.2, and writing a 1 to the port configures the corresponding line as input. Special instructions for bit manipulation allow code sequences like this:

```
MOV     C,P1.2          ;Move P1 bit2 to carry
MOV     P1.3,C          ;move carry bit to P1 bit 3
CLR     P2.1            ;clear P2 bit 1
SETB    P1.0            ;set bit 0 in P1
JB      P3.3,LABEL      ;Jump to LABEL if P3 bit 3 is set
```

The I/O subsystem of the AVR microcontrollers have some interesting distinctive features (refer to Fig. 2.8). First, note the presence of two data registers, having distinct addresses, associated with each port: one, called PINx, is used when the port is configured as input, and the other, called PORTx, is used when the port is configured as output.

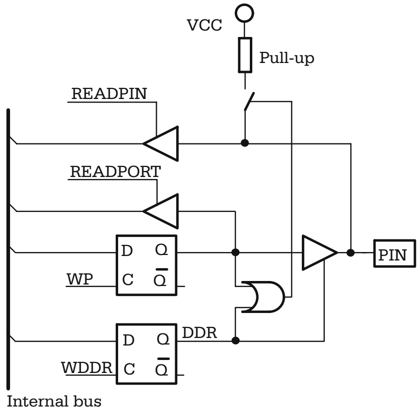


Fig. 2.8. Distinctive Features of the I/O lines of the AVR

Writing to PORTx when DDRx = 0 (i.e. when the port is configured as input) connects internal pull-up resistors to the input lines corresponding to the bits in PORTx set to 1.

The actual status of the input lines is obtained by reading from the address PINx. Reading from the address PORTx, when the port is configured as output, returns the last value written to the port.

Special instructions have been provided to allow software access to the I/O ports or to the individual bits thereof. These are:

```
IN     Rd,ioport     ;read from the address ioport to
                     ;register Rd
OUT    ioport,Rs     ;write data from Rs to ioport
SBI    ioport,bit    ;set specified bit to ioport
CBI    ioport,bit    ;clear specified bit to ioport
```

The following program fragment illustrates aspects of the configuration and access to PORTB:

```
LDI    R16, $F0     ;configure upper nibble of PORTB
                    ;as output
OUT    DDRB,R16
LDI    R16,$0F
OUT    PORTB,R16    ;enable internal pull-ups on
                    ;lower nibble and write 0 in the
                    ;upper nibble
IN     R0,PINB      ;read the input lines
```

2.5 Exercises

SX 2.1

Assuming that the peripheral interfaces that share the lines of PORTD of HC11 are disabled, specify the content of accumulator A after the execution of the following code fragment:

```
LDAA   #$38
STAA   DDRD
LDAA   #$30
STAA   PORTD
LDAA   PORTD
```

Solution

Bits 7 and 6 of PORTD of HC11 are not implemented, and always read 0. By writing \$38 = 00111000b to DDRD, bits 5, 4, 3 of PORTD are configured as output lines. Only these three bits are affected by subsequent write operations to PORTD. Read operations from an output port return the last value written to the port.

For this reason, the content of A after the final read from PORTD in the above example is 00110xxxb. The least significant three bits are input lines, and their status is determined by the logic levels on the external pins of the MCU.

SX 2.2

Write a code fragment to configure the upper nibble of PORTC of AT90S8535 as output, and the lower nibble as input, with the internal pull-up resistors enabled, then read the values of PINC0-3 and write them to PORTC4-7.

Solution

```
LDI     R16,$F0 ;configure port
OUT     DDRC,R16
LDI     R16,$0F ;enable pull-ups
OUT     PORTC,R16
IN      R16,PINC ;read input lines
SWAP    R16 ;swap nibbles
SBR     R16,$0F ;keep pull-ups active
OUT     PORTC,R16 ;write to port
```

SX 2.3

Write a code fragment that configures the line P2.0 of a 8051 as input, then reads the status of this line, and writes the value read to P2.7.

Solution

```
SETB    P2.0      ;P2.0 configured as input
MOV     C,P2.0    ;read input line to carry
MOV     P2.7,C    ;write carry to the output line
```

3

Using the Asynchronous Serial Interface

3.1 In this Chapter

This chapter is an introduction to serial communication. It contains the description of the asynchronous serial communication interface of HC11, AVR, and 8051, as well as an overview of the RS232 and RS422/485 interfaces, and the principles of creating simple microcontroller networks.

3.2 Synchronous vs. Asynchronous Communication

The main distinctive feature of a serial communication system is that data is handled in series, i. e. bit by bit. The simplest serial communication device is the shift register. Consider the example in Fig. 3.1, where two shift registers are connected in such a way that the content of the first, called the *transmitter*, is transferred to the second, called the *receiver*.

Note that, in this case, the shift clock CLK, and the control signals SH/LD\ and RSTR must be generated at the transmitter level, at precise moments of time (see Fig. 3.2.) and transmitted along with data on the communication line. Such a communication system, where the transmission clock is sent to the communication line, is called *synchronous communication*.

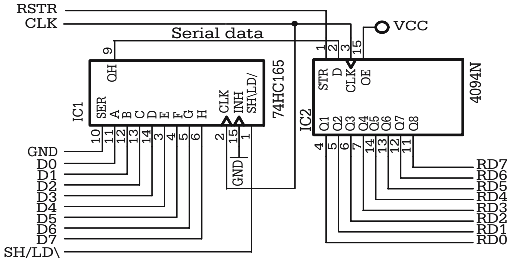


Fig. 3.1. Example of synchronous serial communication circuit

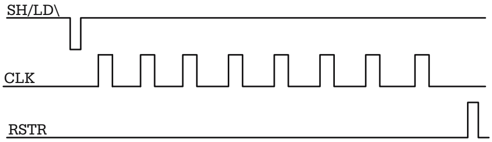


Fig. 3.2. Waveforms of the control signals for the circuit presented in Fig. 3.1

The problem becomes more complicated when it is not possible to send the serial clock over the communication line. In this situation, the receiver must generate its own clock, RxCLK, to shift data into the Rx shift register.

This type of serial communication, where the serial clock is not transmitted on the communication line, is called *asynchronous communication*. The generic block diagram of an asynchronous communication system is shown in Fig. 3.3.

To make this possible, the first requirement is that the transmitter and the receiver clock have exactly the same frequency. A limited number of possible frequencies have been standardized for asynchronous communication. These are: 110, 300, 600, 1200, 2400, 4800, 9600, 19 200, 57 600, 115 200 Hz.

Since the frequency of the transmission clock is directly related to the communication speed, so that with each clock pulse a bit of information is transmitted, the communication speed is measured in *bits per second* or *baud*. The period of the transmission clock is called the *bit time* T_b.

The second requirement is to mark somehow the beginning of the transmission of a sequence of bits. For this purpose, a special synchronization bit, called the *start bit*, has been inserted. This has the polarity opposed to the idle line status and its duration equals one TxCLK period.

The moment when the transmission ends is known, because the number of bits in each packet is known. In most cases, data is sent in 8-bit packets. To make sure the communication line returns to its idle status after each data packet is transmitted, an

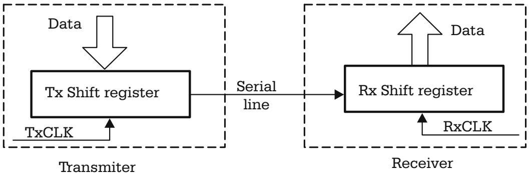


Fig. 3.3. Block diagram of an asynchronous communication system

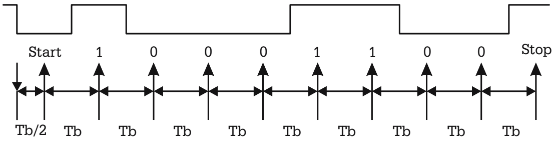


Fig. 3.4. Sampling data line in an asynchronous serial communication

additional stop bit is transmitted. This always has the status of the idle line. Figure 3.4. shows how the data line is sampled at the receiver.

The falling edge of the data line, corresponding to the start bit, starts the reception process.

The data line is sampled after half of the bit time interval to check for a valid start bit, and then at intervals equal to T_b. The values of the data line at these moments are shifted into the receiver data shift register Rx.

Important notes. The first bit transmitted in an asynchronous serial communication is the least significant bit (LSB).

The idle line status is HIGH.

The start bit always has the opposite polarity of the idle line, i. e. it is always 0.

The stop bit always has the polarity of the idle line, i. e. it is always 1.

3.3 Error Detection in Asynchronous Communication

One serious problem when handling asynchronous serial communication is the vulnerability to electromagnetic interference. Several means have been provided to detect communication errors, and the status register of an asynchronous serial interface normally contains special status bits to indicate these errors.

If, for instance, when sampling the data line at the moment $T_0 + T_\text{b}/2$ (refer to Fig. 3.4), a value of 1 is obtained, that means the falling edge detected at the moment T_0 was not a valid start bit, but a spike due to electromagnetic noise. This type of error is called *noise error*.

Similarly, if the data line status at the moment $T_1 = T_0 + T_\text{b}/2 + 9 \times T\text{b}$ is not HIGH, this means that the expected stop bit is invalid, indicating that the byte received is in error. This type of error is called *framing error*.

Obviously, these two control methods are insufficient to detect all possible errors. Another method to verify the integrity of data is *parity control*. For this purpose, a special bit, called the *parity bit*, is transmitted just before the stop bit. This is either the most significant bit of each byte, or an additional ninth bit attached to each byte.

The values of the parity bits are automatically set so that the total number of 1s contained in the byte, plus the parity bit, is always an ODD or EVEN number, at the user's choice. Both the transmitter and receiver must calculate the parity according to the same rule (ODD or EVEN). Upon reception of each byte, the parity is calculated, and, if the parity does not match the rule, the error is reported.

Parity control still cannot detect all errors, but, at the hardware level, the methods described above are all that can be done for error detection. For better error detection, software techniques must be used. Basically, software detection of communication errors relies on the following principles:

- The communication is based on *data packets*, having a determined structure.
- Each data packet contains a special field reserved for a sophisticated checksum. The transmitter computes the checksum for each packet and inserts it into the reserved field of the packet.
- The receiver recalculates the checksum of the packet and compares it with the value calculated by the transmitter and sent along with the packet. If the two values don't match the packet is rejected, and the transmitter is requested to repeat the transmission of the packet.

This method is called Cyclic Redundancy Check Control (CRC). The algorithms used to compute the checksums are so complex that the probability that a packet with errors still has a correct checksum is extremely low.

3.4 The General Structure of the Asynchronous Serial Communication Interface

The general block diagram of an asynchronous serial interface is presented in Fig. 3.5. This circuit is called a Universal Asynchronous Receiver Transmitter (UART). There are numerous stand-alone integrated circuits with this function, but most microcontrollers include a simplified version of UART, with the generic name Serial Communication Interface (SCI). This chapter contains details on the implementation of the SCI of HC11, AVR and 8051 microcontrollers. Even though there are differences in what concerns the names of the registers associated with the interface, or the names and particular functions of the control and status bits, the general structure of the interface is basically the same in all microcontrollers.

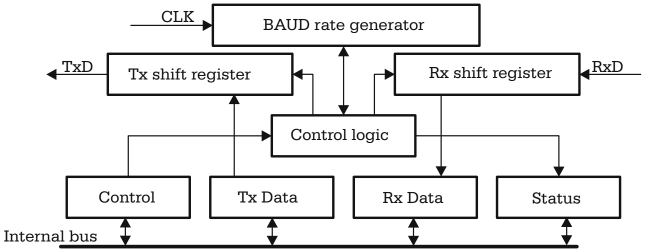


Fig. 3.5. General block diagram of the asynchronous serial communication interface

3.5 The Serial Communication Interface of 68HC11F1

SCDR – SCI Data Register

The transmitter's and receiver's data registers have the same address and the same name: Serial Communication Data Register (SCDR). Physically, they are distinct registers, but the write operations to SCDR are directed to the transmitter's data register, while the read operations from SCDR return the content of the receiver's data register.

BAUD – Baud Rate Generator Control Register

To select the communication speed, a special register, called BAUD, has been provided. This controls how the system clock E is divided in the baud rate generator block, before it is applied to the control logic that actually generates the clock for the serial shift registers. The system clock is first applied to a programmable counter, called a prescaler. The prescaler output is then applied to a second programmable counter. *After the two division stages the frequency of the resulting clock is 16 times the actual baud rate.*

The BAUD register has the following structure:

BAUD	7	6	5	4	3	2	1	0
	TCLR	–	SCP1	SCP0	RCKB	SCR2	SCR1	SCR0
RESET	0	0	0	0	0	0	0	0

- TCLR – Clear Baud Rate Counters – and RCKB – SCI Baud Rate Clock Check, are only used in special test operating mode.
- SCP[1:0] – SCI Baud Rate Prescaller Selects. These two bits control the prescaler division rate, as defined in Table 3.1.
- SCR[2:0] – SCI Baud Rate Selects. These bits control the second stage programmable counter of the baud generator, as shown in Table 3.2

Table 3.1. HC11 prescaler control bits

SCP1	SCP0	Prescaller divide internal clock by:
0	0	1
0	1	3
1	0	4
1	1	13

Table 3.2. HC11 baud rate select bits

SCR2	SCR1	SCR0	Prescaller output is divided by:
0	0	0	1
0	0	1	2
0	1	0	4
0	1	1	8
1	0	0	16
1	0	1	32
1	1	0	64
1	1	1	128

Example Knowing that the oscillator frequency is 8 MHz, and that the output frequency of the baud generator must be 16 times the actual baud rate, determine the value to write in the BAUD register in order to obtain 9600 baud communication speed.

Solution The input clock for the prescaler is the system clock E.

$$f_E = f_{OSC}/4 = 2\,\text{MHz} = 2\,000\,000\,\text{Hz}$$

The output clock must have the frequency:

$$f_1 = 9600 \times 16 = 153\,600\,\text{Hz}$$

This gives the global division rate: $2\,000\,000/153\,600 = 13$.

Choose the prescaler to divide by 13 (SCP1:SCP0 = 1:1), and the secondary counter to divide by 1 (SCR2:SCR1:SCR0 = 0:0:0). The resulting value for the BAUD register is 00110000b = 30H.

Note that for a given oscillator frequency, not all the possible baud rates can be obtained by programming the BAUD register.

With an 8-MHz oscillator clock it is impossible to obtain the baud rate of 19 200 bps, because the resulting global division rate is 6.5 (obviously, the division constant must be as close as possible to an integer value). The solution in this situation is to choose a different oscillator frequency. If, for example, the oscillator frequency is 7.3728 MHz, the global division rate required for 19 200 baud is 6, which can be easily obtained by choosing the prescaler to divide by 3, and the secondary counter to divide by 2.

SCSR – SCI Status Register

SCSR	7	6	5	4	3	2	1	0
	TDRE	TC	RDRF	IDLE	OR	NF	FE	–
RESET	0	0	0	0	0	0	0	0

- TDRE – Transmitter Data Register Empty
  This bit is automatically set when the transmitter's data register is available for a write operation.

  0 = SCDR busy
  1 = SCDR ready for a new operation

- TC – Transmit Complete flag
  The meaning of this bit is very similar to TDRE. The difference is that TC refers to any activity of the transmitter, including sending of break sequences on the serial line.

  0 = Transmitter busy
  1 = Transmitter ready

- RDRF – Receive Data Register Full Flag

  0 = SCDR empty
  1 = SCDR full

  This flag indicates that a character has been received in SCDR and it is ready to be handled by software. RDRF is cleared by reading SCSR followed by a read of SCDR.

- IDLE – Rx Idle Line Detected Flag

  0 = RxD line is active
  1 = RxD line is idle

  IDLE describes the status of the RxD line. It is set to 1 when RxD stays high for at least one character time. IDLE is cleared by reading SCSR then reading SCDR. Once cleared, it is not set again until the line becomes active, and then idles again.

- OR – Overrun Error Flag

  0 = No overrun
  1 = Overrun detected

  This is an error flag. An overrun error occurs when all the bits of a new character are received, and the previous character has not been handled by the software (RDRF = 1).

- NF – Noise Error Flag
  One of the situations when this error flag is set has been described in Sect. 3.3. In fact, the reception data line is sampled several times during each T_{b} interval. NF is set when the samples corresponding to the same interval T_{b} have different values.

  0 = character received without noise
  1 = noise detected for the last character received

- FE – Framing Error
  FE is set when a logic zero is detected in the position of the stop bit.

  0 = Stop bit detected in the right position
  1 = Zero detected instead of a stop bit

All error bits are cleared by reading SCSR followed by a read from SCDR.

SCCR1 – Serial Communications Control Register 1

This is the first of the two control registers of the interface. Only four bits are implemented in this register as follows:

SCCR1	7	6	5	4	3	2	1	0
	R8	T8	–	M	WAKE	–	–	–
RESET	1	1	0	0	0	0	0	0

The bits R8, T8, and M control the number of bits of the character transmitted/received over the interface. If the MODE bit M = 0, 8-bit characters are transmitted along with the corresponding start and stop bits. If M = 1, *9 data bits* are transmitted for each character. The least significant 8 bits are placed in SCDR, and the most significant bit is T8 in case of a transmission operation and R8 in case of reception.

The control bit WAKE is related to an operating mode of the interface, which is specific to HC11, and will not be discussed in this book.

SCCR2 – Serial Communications Control Register 2

This register contains the main control bits of the interface

SCCR2	7	6	5	4	3	2	1	0
	TIE	TCIE	RIE	ILIE	TE	RE	RWU	SBK
RESET	1	1	0	0	0	0	0	0

The most important control bits in this register are TE (Transmitter Enable) and RE (Receiver Enable).

When TE = 1, the whole transmitter subsystem is enabled. Similarly RE = 1 enables the receiver subsystem. These bits are cleared at RESET, therefore they must be set by software in the SCI initialization sequence.

The most significant four bits of this register, TIE, TCIE, RIE, ILIE, are local interrupt masks for SCI-related interrupts.

- TIE – Transmitter interrupt enable

  0 = TDRE interrupts disabled
  1 = An interrupt request is generated when TDRE = 1.

- TCIE – Transmit Complete Interrupt Enable

  0 = TC interrupts disabled
  1 = SCI interrupt requested when TC status flag is set

- RIE – Receiver Interrupt Enable

  0 = RDRF and OR interrupts disabled
  1 = SCI interrupt requested when RDRF (receiver data register full) flag or the OR (overrun error) bit in SCSR is set

- ILIE – Idle-Line Interrupt Enable

  0 = IDLE interrupts disabled
  1 = SCI interrupt requested when IDLE status flag is set

  This bit is used in connection with the wake-up operating mode.

- RWU – Receiver Wakeup Control

  0 = Normal SCI receiver
  1 = Wakeup enabled and receiver interrupts inhibited

- SBK – Send Break 0 = Break generator off
  1 = Break codes generated

  Writing 1 to this bit of SCCR2 causes a *break character* to be generated, i. e. the TxD line is pulled to zero for at least one character time.

3.6 The Asynchronous Serial Communication Interface of AVR Microcontrollers

The information in this paragraph refers to the microcontroller Atmel AT90S8535. Other members of the AVR family may have different structures of the asynchronous serial interfaces. See the data sheet for each specific microcontroller, for other models.

The data registers of the transmitter and receiver share the same address, just like in case of the HC11 microcontrollers. The two physical registers are accessible for the software as a single register, called UDR (UART Data Register). UDR is entirely similar to SCDR of HC11.

The equivalent of the BAUD register of HC11 is called UBRR (UART Baud Rate Register), for the AVR. The difference is that it is easier to use and much more flexible than the HC11 register. The clock frequency of the output signal of the baud rate generator is 16 times the actual baud rate, and is determined using the following formula:

$$16 \times \text{BaudRate} = F_{\text{osc}}/(\text{UBRR} + 1) \tag{3.1}$$

where UBRR is the contents of UBRR, as written by software.

The status register of the interface is called the USR (UART Status Register) and has the following structure:

USR	7	6	5	4	3	2	1	0
	RXC	TXC	UDRE	FE	OR	–	–	–
RESET	0	0	1	0	0	0	0	0

Here is the description of the status bits in USR:

- RXC – Reception Complete. This bit has a similar function to RDRF of the HC11. It is automatically set by hardware when a character is available in UDR. It is cleared by reading UDR, or when the associated interrupt is executed.
- TXC – Transmission Complete. This bit is set by hardware when the transmission of a character completes. It has a similar function to the TC flag of HC11. *TXC is automatically cleared when the associated interrupt is executed or by writing 1 to the corresponding position of USR. This is the only Read/Write bit of USR. All other bits are Read Only.*
- UDRE – USART Data Register Empty. This is the equivalent of the TDRE status bit of HC11, and indicates that the transmitter is ready to accept a new character.

- OR – Overrun, and FE – Framing Error, have exactly the same meaning as in HC11 SCSR. Unlike HC11, these bits are cleared only when a new character has been received and read correctly.

The control register of the interface UCR (UART Control Register) has the following structure:

UCR	7	6	5	4	3	2	1	0
	RXCIE	TXCIE	UDRIE	RXEN	TXEN	CHR9	RXB8	TXB8
RESET	0	0	0	0	0	0	0	0

- RXCIE, TXCIE, and UDRIE are local interrupt masks associated with the flags RXC, TXC, and UDRE, respectively. When a mask is set to 1 by software, the associated flag generates an interrupt when set to 1.
- TXE and RXE have identical functions with TE and RE of HC11. When TXE = 1 the transmitter is enabled, and RXE = 1 enables the receiver.
- CHR9 – this bit is similar to M from HC11. CHR9 = 1 indicates 9-bit transmission. In this case, RX8 and TX8 are the most significant bits of the transmitted/received characters.

3.7 The Asynchronous Serial Interface of 8051

The 8051 serial communication interface is less typical than those presented in the previous paragraphs. *The first significant difference is that the baud rate generator is missing. It is replaced by one of the system timers.*

The interface *data register* is called SBUF, and it is similar to the data register of HC11 and AVR. The control register of the interface, SCON, also contains several status bits. It has the following structure:

SCON	7	6	5	4	3	2	1	0
	FE/SM0	SM1	SM2	REN	TB8	RB8	TI	RI
RESET	0	0	0	0	0	0	0	0

The bits SM[0:1] (Serial communication Mode control) select one of the four possible operating modes for the serial interface, as shown in Table 3.3.

Table 3.3. 8051 serial communication mode select bits

SM0	SM1	MODE	Description	Baud rate
0	0	0	Shift register	$f_{osc}/12$
0	1	1	8-bit UART	Variable (set by timer)
1	0	2	9-bit UART	$f_{osc}/32$ or $f_{osc}/64$
1	1	3	9-bit UART	Variable (set by timer)

In the operating mode 0, the interface acts like a serial shift register. The serial shift clock has a fixed frequency, equal to $f_{\mathrm{osc}}/12$. Eight data bits are transmitted, starting with the least significant bit (LSB). In mode 1, 10 bits are shifted from (to) SBUF: one start bit, eight data bits (LSB first) and one stop bit. The communication speed is variable and programmable using the system timer. In operating modes 2 and 3, nine data bits are sent, packed by a start bit and a stop bit. The communication speed is fixed in mode 3 and variable in mode 3. Modes 2 and 3 are designed for multiprocessor communication. This operating mode is specific for 8051, and will not be discussed in this book. For normal operating modes 0, 1, the SM2 bit must be cleared.

The most significant bit of SCON, FE/SM0, has a dual function. The selection between the two functions is made by the bit SMOD0, in the register PCON (Power Control Register). If SMOD0 = 1, then bit 7 (SCON) = FE, and if SMOD0 = 0, then bit 7 (SCON) = SM0. FE (Framing Error) is set if a zero is detected in the position of the stop bit while receiving a character, and cleared by writing zero to the corresponding position of SCON.

The control bit REN (Receiver Enable) is used to enable (REN = 1) or disable (REN = 0) the receiver. There is no similar bit to enable/disable the transmitter.

TB8 – RB8 contain the ninth data bit (the most significant bit, MSB) in operating modes 2 and 3.

TI (Transmit Interrupt flag) is set by hardware at the end of the transmission of a character. If the interrupt associated with the serial interface is enabled, the condition TI = 1 generates an interrupt. TI must be cleared by software by writing zero to this position of SCON.

RI (Receive Interrupt flag) is hardware set when a character has been received and is available in SBUF. If the local interrupt mask is set to 1, an interrupt is generated. RI must be cleared by writing zero to this position of SCON.

If the interrupts are disabled, TI and RI can be polled by software.

As described in Chap. 1, the interrupt system of 8051 is controlled by the IE register. *For the serial communication interface, one single bit, called ES, is reserved in this register. Therefore it is not possible to enable/disable the receiver and transmitter interrupts separately.*

To get the full picture on the serial communication of 8051, refer to Chap. 6 for an example on how to use the system timer as a baud rate generator.

3.8 Programming the Asynchronous Serial Interface

When programming *any peripheral interface* there are two major aspects to consider: the initialization of the interface, and the actual data handling. Normally, the initialization sequence is executed only once, after RESET. Data handling can be performed either by periodically testing the status bits of the interface (*polling*), or by enabling the interrupts associated with the interface.

This paragraph contains several examples of initialization sequences and serial communication data handling for HC11 and AVR.

3.8.1 Programming the SCI of HC11

The initialization sequence must do the following:

- Enable the transmitter and the receiver.
- Select the communication speed, by writing an appropriate value to the BAUD register.
- Enable interrupts, if this is required.

Here is an example on how to initialize the SCI of 68HC11F1 for 9600 baud, no interrupts. In this example it is assumed that the oscillator frequency is 8 MHz.

```
INIT_SCI    LDAA   #$30        ;see paragraph 3.5.
            STAA   BAUD        ;9600 baud
            CLR    SCCR1       ;clear M for 8 bit
                               ;communication
            LDAA   #$0C        ;TE=1, RE=1
            STAA   SCCR2       ;no interrupts
            ....
```

And the reception and transmission routines may look like this:

```
SCI_REC     LDAA   SCSR        ;read status register
            ANDA   #$20        ;isolate RDRF bit
            BEQ    SCI_REC     ;wait until RDRF is set
            LDAA   SCDR        ;get received character
            STAA   SOMEWHERE   ;and save it
            RTS

SCI_SEND    TAB                ;save character to B
SSLOOP      LDAA   SCSR        ;read status register
            ANDA   #$80        ;isolate TDRE
            BEQ    SSLOOP      ;wait until transmitter
                               ;ready
            STAB   SCDR        ;send character
            RTS                ;and return
```

This way of writing the SCI_REC routine is a very bad idea. *It is always recommended be avoided wait loops, that when the duration of the loop is unknown.* In the above example, the processor spends most of the time waiting for a character from the SCI. A much better solution would be to write the reception routine like this:

```
SCI_REC2    LDAA   SCSR        ;read status register
            ANDA   #$20        ;isolate RDRF bit
            BEQ    FRET        ;failure return
            LDAA   SCDR        ;get received character
            STAA   SOMEWHERE   ;and save it
            SEC                ;Set Carry to inform
            RTS                ;the main program
FRET        CLC                ;Clear carry
            RTS
```

This time, the processor doesn't wait indefinitely for a character. It tests from the beginning whether a character is available in SCDR, by checking the RDRF flag. If a character has been received, this is read and saved in a variable, and the carry flag is set to inform the main program about the event. If no character has been received, the carry bit is cleared. Such a reception routine must be called periodically in a program loop, but it has the advantage that the CPU does not hang up until a character is received.

An even better solution would be to use SCI reception interrupts to handle the reception of characters. For this purpose, the initialization routine must be modified to enable the interrupts generated by RDRF.

```
INIT_SCI    LDAA   #$30        ;see paragraph 3.5
            STAA   BAUD        ;9600 baud
            CLR    SCCR1       ;clear M for 8 bit
                               ;communication
            LDAA   SCSR        ;clear flags if any
            LDAA   SCDR
            CLR    QSCI
            CLR    QSCIERR
            LDAA   #$2C        ;RIE=1, TE=1, RE=1
            STAA   SCCR2       ;enable receiver
                               ;interrupts
            .....
```

The control word written into SCCR2 contains the RIE bit set to 1, thus enabling the reception interrupts. Note that, before enabling the interrupts, SCSR and SCDR are read in this sequence in order to clear any flag that might generate a false interrupt. QSCI and QSCIERR are two variables indicating that a character has been received, or that a communication error has been detected. The interrupt service routine looks like this:

```
SCI_ISR     LDAA   SCSR
            ANDA   #$0E        ;Isolate all error flags
            BNE    SCIERR      ;if error, inform the
                               ;main program
            LDAA   SCDR        ;get character
            STAA   SCIRB       ;save it in a buffer
            INC    QSCI        ;true QSCI
            RTI                ;return from interrupt
SCIERR      LDAA   SCDR        ;read SCDR to clear flags
            STAA   SCIRB
            INC    QSCI        ;true QSCI
            INC    QSCIERR     ;true error flag
            RTI
```

Note that when a reception error occurs, it is important to read the character received to make sure that the flag that has generated the interrupt is cleared. It is seldom required to analyze what error occurred, because in most cases, the only thing to do is to ask for the character to be retransmitted.

Important note. The SCI of HC11 uses two lines of PORTD to implement the transmission and reception lines TxD, RxD. By enabling the SCI transmitter and receiver, the TxD line is automatically configured as an output line, and RxD is configured as an input line, regardless of the contents of DDRD.

3.8.2 Programming the UART of AT90S8535

Here is an example of initializing the UART of AT90S8535 for 19 200 baud, 8 bits per character, no interrupts:

```
.EQU          K19200=25            ;xtal=8 MHz
                                   ;BaudRate=19200
Init_Uart:
              Ldi    R16,K19200    ; set baud rate
              Out    UBRR,R16
              Ldi    R16,$18       ;RXEN=1, TXEN=1
              Out    UCR,R16
              Ret
```

To enable reception interrupts, the control word written to UCR must be modified so that the bit RXCIE = 1 (Reception Complete Interrupt Enable).

```
.EQU          K19200=25            ;xtal=8 MHz
                                   ;BaudRate=19200
Init_Uart:
              Ldi    R16,K19200    ; set baud rate
              Out    UBRR,R16
              Ldi    R16,$98       ;RXEN=1, TXEN=1
              Out    UCR,R16       ;RXCIE=1
              Ret
```

Unlike HC11, AVR microcontrollers clear the interrupt flag automatically by hardware, when the interrupt is executed. The CPU status is NOT saved and restored automatically, and therefore the CPU registers used by the interrupt routine must be saved to the stack by software. Here is an example of a simple interrupt service routine for AVR:

```
Uart_ISR:
              Push   R16           ;save CPU status
              In     R16,SREG
              Push   R16
              In     R16,UDR       ;get character
              Sts    RECBUF,R16    ;save it
              Ldi    R16,$FF
              Sts    QUART,R16     ;true QUART
              Pop    R16           ;restore status
              Out    SREG,R16
              Pop    R16
              Reti                 ;return to main
```

QUART is a software flag that, when true, informs the main program that a character is available. RECBUF is a one-character buffer to store the character received from the UART.

3.8.3 Programming the UART of 8051

The 8051 asynchronous serial interface does not include a dedicated baud rate generator. It uses the internal timer to generate the serial clock. Refer to Chap. 6 to understand how the timer is used in the following initialization routine. The control word written to SCON selects the operating mode 1 for the serial interface, and sets the bit REN = 1 to enable the receiver subsystem.

```
INIT_UART:
              MOV    SCON,#50H     ;UART mode 1, REN=1
              MOV    PCON,#80H     ;SMOD=1
              MOV    TMOD,#20H     ;C/T=0, M1=1, M0=0
              MOV    TH1,#0FAH     ;auto reload value
              MOV    TCON,#40H     ;TR1=1 -- start counting
              RET
```

Serial interface interrupts can be enabled by setting the bit ES in the register IE. Below is an example of serial reception and transmission routines, which use RI and TI polling rather than interrupts:

```
GETCHR:
              CLR    C
              JB     RI,GETCHR1    ;if character received
              RET
GETCHR1:      MOV    A,SBUF        ;get it
              CLR    RI            ;always clear flag!
              SETB   C             ;inform main program
              RET
SENDCHR:
              CLR    C
              JB     TI,SENDCHR1   ;check if transmitter
                                   ;ready
              RET
SENDCHR1:
              CLR    TI
              MOV    SBUF,A        ;start sending
              SETB   C             ;set carry to inform main
                                   ;program
              RET
```

3.9 Hardware Interfaces for Serial Communication

The asynchronous serial interface has been created to allow connection between digital devices over relatively long distances. The problem is that, as the distance increases, the effect of parasite capacitance and inductance of the cables becomes more important, and the electromagnetic interference grows stronger. As a consequence, the signals transmitted in this way are distorted and attenuated to such an extent that the information carried by the signals cannot be extracted at the receiver.

To overcome the perturbing effect of long connection cables, several methods of processing the electrical signals have been developed. Before transmission, the signals are transformed by special interfaces, in order to increase their overall immunity to perturbations. This section describes three such interfaces.

3.9.1 The RS232 Interface

RS is an abbreviation for Recommended Standard. The recommendation comes from the Electronics Industry Association (EIA), which proposed in 1969 the RS232 standard, aiming to bring order in the multitude of technical solutions to the problems of serial communication.

The main characteristic of RS232 is that the signals are carried as single voltages, referred to a common ground. The voltage levels associated with the logic levels 0 and 1 are as follows:

For logic 0, the voltage on the communication line must be in the range +6 V to +15 V.

For logic 1, the voltage must be in the range −6 V to −15 V.

In practice, the transmission devices generate ±12 V. The receiving devices accept as logic 0 any signal ranging from +3 V to +15 V, and as logic 1 any signal with the amplitude in the range −3 V to −15 V.

A number of special RS232 interface circuits have been developed. One very popular circuit with this function is MAX232, from Maxim Semiconductors. This contains two RS232 transmitters and two RS232 receivers in the same package. The major advantage of this circuit is that it uses an internal oscillator and four external capacitors to generate the ±12 V power supply sources, starting from a single +5 V

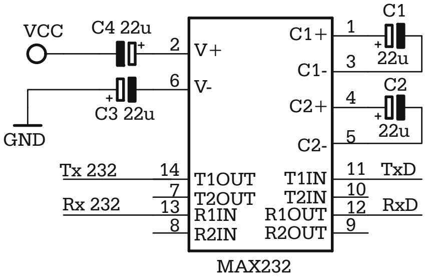


Fig. 3.6. A typical RS232 interface circuit MAX232

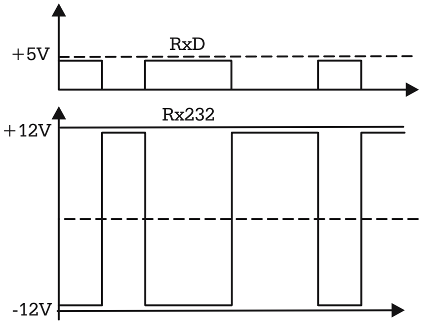


Fig. 3.7. Waveforms illustrating the level conversion performed by MAX232

power supply. Most of the projects described in this book use this circuit. The schematic of an RS232 interface using MAX232 is shown in Fig. 3.6.

Figure 3.7 shows the waveforms of a signal before and after the MAX232 driver.

3.9.2 Differential Communication. The Interfaces RS422 and RS485

The RS232 interface guarantees safe transmission up to a distance of 15 meters, at a maximum communication speed of 19 200 baud. In practice, these limits are often exceeded, but the overall reliability of the communication system drops. For longer distances, and higher communication speed, a different interface is required.

The interfaces RS422 and RS485 use two conductors to transmit each digital signal, the logic levels being defined by the relative difference of potential between the two conductors. The transmitter's line driver controls the voltage levels on the two conductors, so that the difference of potential between them is positive for logic 1 and negative for logic 0, as illustrated in Fig. 3.8. This type of transmission is called *differential*.

The advantage of the differential interface is that the electromagnetic interference and the ground potential differences appear as common mode voltage for the receiver, and they are rejected. This creates a much better immunity to perturbations and allows communication speeds up to 10 Mb per second and cable length up to 1200 meters, using twisted pair cable. The receiver requires at least 200 mV differential input signal in order to reconstruct the original signal.

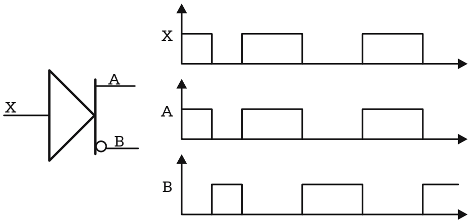
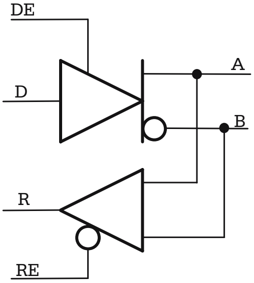


Fig. 3.8. Waveforms for differential transmission



Input D	Enable DE	Output	
		A	B
H	H	H	L
L	H	L	H
X	L	Z	Z

Fig. 3.9. Typical RS422/RS485 transceiver circuit

Both the RS422 and RS485 interfaces use differential communication. The difference is that RS422 is designed for point-to-point communication. Only one transmitter and up to 10 receivers may be connected on a conductor pair, in a simplex transmission. RS485 allows multipoint communication, i. e. up to 32 transmitters/receivers may be connected on a conductor pair, in a half-duplex communication. Therefore, the RS485 transmitters must be able to drive their outputs into a high-impedance status, allowing other transmitters to control the communication line. *Note that the RS485 standard defines only the electrical characteristics of the signals and devices, and not the rules for accessing the RS485 bus, i. e. the communication protocol.*

A typical RS422/RS485 circuit is SN 75176. The functional block diagram of this circuit is shown in Fig. 3.9.

This circuit contains in the same capsule one differential transmitter, and one differential receiver, having individual enable inputs: DE (Driver Enable) active HIGH, and RE\backslash(Receiver Enable) active LOW. When DE = 0, the outputs A, B are in a high impedance status, thus the transmitter is virtually disconnected from the communication line, leaving the control of the line to other similar devices.

3.9.3 The Current Loop Interface

When the length of the communication line is higher than 1 km, or in harsh industrial environments, where galvanic isolation between the transmitter and the receiver is essential, it is possible to use the interface presented in Fig. 3.10.

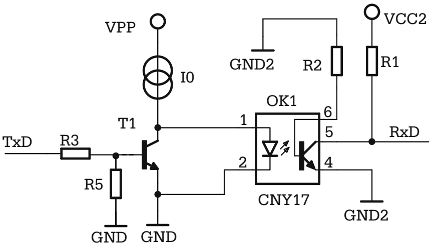


Fig. 3.10. Current loop transmission circuit

The current generated by I_0 closes to the ground either through the transistor T1, when this is saturated, or through the LED of the optocoupler OK1, when T1 is blocked. As a consequence, the waveform of the signal in the collector of the optocoupler's transistor reproduces the TxD signal. Such a circuit can be used for point-to-point communication systems to transmit data at 9600 bps, over cables a few kilometers long.

3.10 Basic Principles of Networking with Microcontrollers, Using the Asynchronous Serial Interface

A microcontroller-based device can be connected to a network if, at the hardware level, it possesses a communication interface which allows multipoint communication, and, at the software level, is programmed to observe a set of rules concerning the access to the communication channel. This set of rules defines the network *communication protocol*.

From the hardware point of view, the most common solution used to implement microcontroller-based networks uses the RS485 differential communication interface.

At the software level, a multitude of protocols has been developed, most of them being structured as MASTER–SLAVE protocols. In a MASTER–SLAVE structured network, a single device, called the MASTER, initiates all data transfers on the communication bus. Each SLAVE device has a unique identification code, or *address*. The MASTER device starts the communication by sending *data packets* that are received and decoded by *all* SLAVE devices. Only one SLAVE is authorized to answer the query – the one that recognizes its own address in the data packet received.

The general structure of data packets transferred in a network is given below:

Header	Body	CRC	Tail

The fields Header and Tail are predefined sequences that define the beginning and the end of the packet. The body of the packet must contain the address of the destination slave device, the operation code, and, of course, a data field. The CRC field is a checksum used to detect communication errors, as described in Sect. 3.3. Assuming that all the bytes in the packet are ASCII codes, then the Header and Tail fields can be reduced to a single byte, e. g. STX (Start of Text – \$02) for the header, and EOT (End of Text+– \$04) for the tail.

Example Design a simple communication protocol for a microcontroller network comprising one MASTER device and up to 10 slave devices. Each slave device must be able to report the status of two digital inputs, and to change the status of two digital outputs, by executing specific commands from the master device.

Solution To match these requirements, the structure of the body of the packets sent by MASTER may be the following:

Opcode	SADR	Data1	Data2

The *opcode* must identify the two possible operations: reporting the status of the inputs, and changing the status of the output lines, e. g. the opcode may be the ASCII character ‘I’ for read input, and ASCII ‘O’ for write output.

SADR is the address of the slave required to perform the operation indicated by the opcode. Since the maximum number of slaves in the network is 10, SADR may be the ASCII representation of the numbers ‘0’–‘9’.

Data1 and Data2 are two bytes reserved for the logic values of the inputs and outputs. In case of a ‘write output’ command, these bytes define the new status of the outputs: ‘0’, ‘1’, or ‘T’ (Toggle). In case of a ‘read input’ command, these bytes have no meaning, and always have the value ‘0’. They are maintained in the packet for the sole purpose of keeping the length of the packets constant.

All packets sent by MASTER must have an answer from the addressed slave. The packets sent by the slaves have the same structure, but the opcodes are different, e. g. ACK (acknowledge $06) for a command received and executed correctly, and NAK (negative acknowledge $15) for a packet received with the wrong CRC. When the slave answers with a NAK type packet, the MASTER repeats the query.

See Chap. 13 for the detailed description of an implementation of this protocol, used to interrogate a number of addressable sensors.

3.11 Exercises

SX 3.1

What is the error contained in the interrupt service routine for the reception of characters on SCI for HC11 listed below?

```
SCI_ISR      LDAA   SCSR
             ANDA   #$0E         ;Isolate error flags
             BNE    SCIERR       ;if error, inform the main
                                 ;program
             LDAA   SCDR         ;get character
             STAA   SCIRB        ;save it in a buffer
             INC    QSCI         ;true QSCI
             RTI                 ;return from interrupt
SCIERR       INC    QSCIERR      ;true error flag
             RTI
```

Solution

The SCI reception interrupt of HC11 is generated when (RIE = 1) AND ((RDRF = 1) OR (OR = 1)), i. e. when the interrupt is enabled, and one of the conditions RDRF = 1 (character received) or OR = 1 (overrun error – a new character is received, and the previous character has not yet been read by the software).

Both RDRF and OR flags are cleared by successively reading SCSR and SCDR.

In the interrupt service routine listed above, everything works fine until an error occurs. In this case, the error is detected, and the program jumps to the label SCIERR, where it reaches RTI without reading SCDR. As a consequence, the flag that generated the interrupt is NOT cleared, and a new interrupt is generated right after RTI. *Remember that any interrupt service routine must clear the condition that generated the interrupt.*

SX 3.2

What is the main difference between the RS422 and RS485 interfaces?

Solution

Both RS422 and RS485 are designed for differential communication. The main difference between the two standards is that RS422 is aimed at point-to-point communication, where only one transmitter is connected to the communication line.

RS485 is intended for multipoint communication where more than one transmitter is allowed to access the serial bus. Therefore, the RS485 line drivers must be able to go to a high impedance status.

Normally the RS485 interface circuits can be used to implement RS422 point-to-point communication by simply keeping the transmission driver permanently active.

SX 3.3

Draw the schematic of an RS422 full duplex communication line, using the SN75176 circuit, described in Sect. 3.9.2.

Solution

In full duplex communication it is possible to receive a character during the transmission of another. For this to be possible, the transmission and reception lines must be physically distinct. In case of differential communication, two conductors are required for transmission and two others for reception.

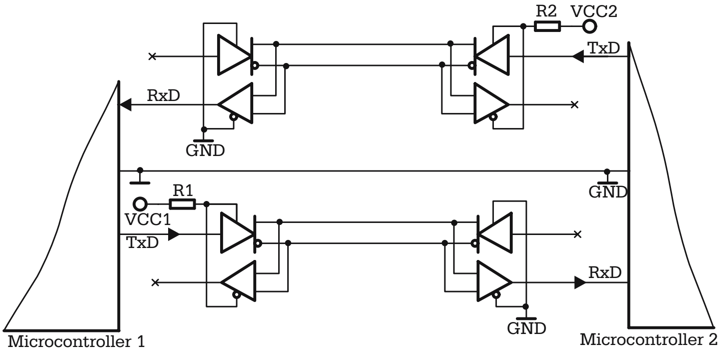


Fig. 3.11. Implementation of a full duplex RS422 communication line using SN75176

The SN75176 circuit can operate as a RS422 transmitter or receiver, if the DE or RE signals are permanently active. This means that two SN75176 circuits are required at each end of the communication line, as shown in Fig. 3.11.

Both the RS232 and RS422 interfaces are used for point-to-point communication links. The difference is that RS422 works over much longer distances than RS232.

4

Using the Synchronous Serial Interface SPI

4.1 In this Chapter

This chapter contains a detailed description of the SPI of HC11 and AVR microcontrollers, with hardware and software applications examples.

4.2 General Description of the SPI

As mentioned in Chap. 3, in synchronous serial communication, both the transmitter and receiver use the same synchronization clock to transfer data. Therefore, the device that generates the transmission clock fully controls the moment and the speed of the transmission, and it is called the MASTER.

The HC11 and AVR families of microcontrollers include a synchronous serial interface, called the Serial Peripheral Interface (SPI), allowing operation either in MASTER mode or in SLAVE mode. The general block diagram of a SPI link is presented in Fig. 4.1.

Four shift registers are involved in this link. All work with the same shift clock, generated by the MASTER device. The transmission and reception processes occur simultaneously. Two data line are provided to this purpose: MOSI – (MASTER Out SLAVE In) and MISO (MASTER In SLAVE Out).

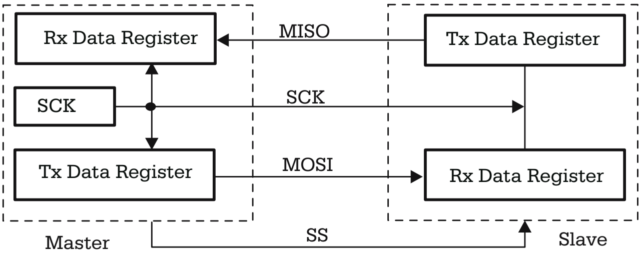


Fig. 4.1. General block diagram of a SPI link

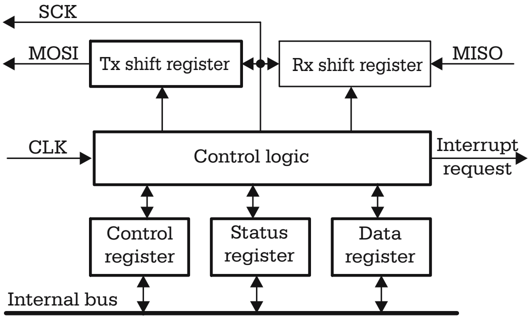


Fig. 4.2. General block diagram of the SPI

When the MASTER writes by software a byte into the transmitter's data register, eight clocks are automatically generated to the SCK line, and the bi-directional transmission begins. Whatever is found at this moment in the SLAVE's Tx Data Register is shifted into the MASTER's Rx Data Register.

When more than one SLAVE device is connected to the SPI bus, only one SLAVE may be enabled at a certain time. An additional signal SS (SLAVE Select), active LOW, is used to select the active SLAVE device.

4.3 The SPI of HC11 Microcontrollers

The structure of the SPI of HC11 follows exactly the general structure of a peripheral interface, shown in Fig. 1.6. The registers of the interface are called SPDR (SPI Data Register), SPCR (SPI Control Register), and SPSR (SPI Status Register).

SPDR – SPI Data Register

When writing to the address of SPDR, data is directly written into the transmitter's shift register. Reading from this address actually reads the last byte received in the receivers shift register. SPDR is double buffered for input, and single buffered for output. This means that SPDR can be read while a data transfer is in progress, but if a byte is written into SPDR before the transmission of the previous character is completed, an error condition occurs.

SPSR – SPI Status Register

The status register has the following structure:

SPSR	7	6	5	4	3	2	1	0
	SPIF	WCOL	–	MODF	–	–	–	–
RESET	0	0	0	0	0	0	0	0

- SPIF – SPI transfer complete flag. This flag is set when a SPI transfer is complete (after eight SCK cycles in a data transfer). It is cleared by reading SPSR (with SPIF = 1), then accessing (reading or writing) SPDR.

  SPIF = 0 No SPI transfer complete or SPI transfer still in progress
  SPIF = 1 SPI transfer complete

- WCOL – Write Collision. This is an error flag. It is set if the software tries to write data into SPDR while SPI data transfer is in progress. Clear this flag by reading SPSR (WCOL = 1), then access SPDR.

  WCOL = 0 No write collision
  WCOL = 1 Write collision

- MODF – Mode Fault. This is another error flag. This error occurs when the input line SS is pulled LOW (the device is hardware configured as SLAVE) while the bit MSTR in the control register SPCR is set to 1.

  MODF = 0 No mode fault
  MODF = 1 Mode fault

  This type of error occurs in multi-MASTER networks when more than one device is trying to become MASTER at a certain moment.

All other bits of SPSR are not implemented and always read 0. Out of RESET, SPSR is cleared.

SPCR – SPI Control Register

SPCR	7	6	5	4	3	2	1	0
	SPIE	SPE	DWOM	MSTR	CPOL	CPHA	SPR1	SPR0
RESET	0	0	0	0	0	1	U	U

- SPIE – SPI Interrupt Enable. When set to 1, this bit enables interrupt requests from the SPI system.

  SPIE = 0 SPI system interrupts disabled
  SPIE = 1 SPI system interrupts enabled

  SPI can generate interrupts in two situations:
  – *Each time the SPIF bit in SPSR is 1 (a serial data transfer is completed).*
  – *Each time MODF is set.*
  SPE – Serial Peripheral System Enable

  SPE = 0 SPI system disabled
  SPE = 1 SPI system enabled

  This control bit enables the whole SPI system and assigns port D bits 2, 3, 4, 5 to SPI. If the SPI works in master mode and DDRD bit 5 is set, then the port D bit 5 pin becomes a general-purpose output line instead of the SS input.

- DWOM – Port D Wired-OR Mode. This bit selects the type of output on PORT D. DWOM affects all port D pins. This bit is not directly related to the SPI. It is placed here because the SPI uses most of the PORTD bits.

  DWOM = 0 PORTD is configured as normal CMOS outputs
  DWOM = 1 PORTD has open-drain outputs

- MSTR – Master Mode Select. This bit must be set by software to configure the device to work as MASTER for the SPI link.

  MSTR = 0 Slave mode
  MSTR = 1 Master mode

- CPOL – Clock Polarity

  CPOL = 0 SCK line idles LOW
  CPOL = 1 SCK line idles HIGH

- CPHA – Clock Phase

  CPHA = 0 SS must go HIGH between successive characters transmitted.
  CPHA = 1 SS can be maintained LOW for all transfers

  CPOL and CPHA must be programmed with identical values for MASTER and SLAVE

- SPR[1:0] – SPI Clock Rate Selects

When the device is configured as MASTER, these two bits select how the SPI clock (SCK) is obtained by dividing the main clock E, as described in Table 4.1. *When the device is configured as SLAVE, these bits have no effect, because the MASTER generates the clock SCK.*

Table 4.1. The effect of programming [SPR1–SPR0]

SPR1-SPR0	SCK is E clock divided by:
00	2
01	4
10	16
11	32

Initialization of the Interface

Obviously, the initialization sequence is different for MASTER and SLAVE. Below is an example of an initialization sequence. Note that the SPI system interrupts are enabled at the SLAVE device.

```
*MASTER SPI initialization routine
SPI_INITM  LDAA   #$20       ;configure PORTD bit 5 as
                             ;general purpose output
           STAA   DDRD
           BCLR   PORTD,$20  ;make SS=0
           LDAA   #$56       ;SPI enable, Master,
                             CPHA=1
                             ;clock:32
           STAA   SPCR
           RTS
* SLAVE SPI initialization routine
SPI_INITS  LDAA   #$C6       ;SPIE, SPE, Slave, CHPA=1
                             ;clock:32
           STAA   SPCR
           RTS
```

4.4 The SPI of the AVR Microcontrollers

The registers of the SPI interface on AVRs are almost identical with those described for HC11. Even the names of the registers are identical: SPDR, SPSR, SPCR. Note that some AVR microcontrollers have a so-called Universal Serial Interface (USI), instead of SPI. The registers of USI are called USIDR, USISR, USICR. This paragraph describes the SPI registers of AT90S8515/8535.

SPSR – SPI Status Register

The status register has the following structure:

SPSR	7	6	5	4	3	2	1	0
	SPIF	WCOL	–	–	–	–	–	–
RESET	0	0	0	0	0	0	0	0

Only two bits are implemented in SPSR: SPIF and WCOL. They have identical functions with the homologous bits of HC11. MODF is not implemented at the AVR, because the behavior of the AVR is slightly different. *If the SS input is pulled LOW when the device is software configured as MASTER, the MSTR bit in SPCR is automatically cleared.*

SPCR – SPI Control Register

SPCR	7	6	5	4	3	2	1	0
	SPIE	SPE	DORD	MSTR	CPOL	CPHA	SPR1	SPR0
RESET	0	0	0	0	0	0	0	0

DWOM is replaced here by DORD (Data Order).

DORD = 1 then the LSB of the data word is transmitted first
DORD = 0 then the MSB of the data word is transmitted first.

The effect of SPR1–SPR0 is also different, as described in Table 4.2.

Table 4.2. The effect of programming [SPR1–SPR0]

SPR1-SPR0	SCK is the system clock divided by:
00	4
01	16
10	64
11	128

Initialization of the SPI Interface

```
INIT_SPIM:                          ;MASTER initialization
           SBI    DDRB,5      ;configure portb, bit 5 as
                              ;output (MOSI)
           SBI    DDRB,7      ;configure portb, bit 7 as
                              output (SCK)
           SBI    DDRB,4      ;configure portb, bit 4 as
                              ;output
           LDI    R16,0X54    ;master, no interrupts,
                              ;fastest clock
           OUT    SPCR,R16
           RET                ;return to main program


INIT_SPIS:                          ;SLAVE initialization
           CBI    DDRB,5      ;configure portb, bit 5 as
                              ;input (MOSI)
           SBI    DDRB,6      ;portb, bit 6 as output
                              (MISO)
           CBI    DDRB,7      ;portb, bit 7 as input
                              (SCK)
           CBI    DDRB,4      ;portb, bit 4 as input
                              (SS)
           LDI    R16,0XC4    ;slave, interrupts
                              enabled,
                              ;fastest clock
           OUT    SPCR,R16
           RET                ;return to main program
```

Note that the AVR microcontrollers do not automatically configure the I/O lines used by the SPI as input or output lines.

4.5 Examples of Using The SPI

4.5.1 Using the SPI To Connect Two Microcontrollers

The SPI is a fast data communication link. It is not designed to work over long distances. SPI can be used to create networks with two or more microcontrollers, when processing tasks are distributed between two or more microcontrollers for faster processing, or to connect special memory or peripheral devices.

A typical hardware SPI connection between two HC11E9 microcontrollers is shown in Fig. 4.3. In all situations, it is important to define a communication protocol between the two devices.

If, for instance, the SLAVE device is used to expand the MASTER's I/O space with two *input ports*, then the MASTER should send al least three bytes through the SCI: the first byte is an opcode needed to inform the SLAVE about the intention of the MASTER to read data from the SLAVE's ports, and two other dummy bytes must be sent in order to generate the clock pulses needed to read the two data bytes from SLAVE.

Note that the SLAVE device does not have any initiative in a SPI link – it's the MASTER that decides when to start the communication, and how many data bytes are to be exchanged between the two devices.

A convenient way to write the software is to write an interrupt-driven SPI communication routine for the SLAVE, like in the example below:

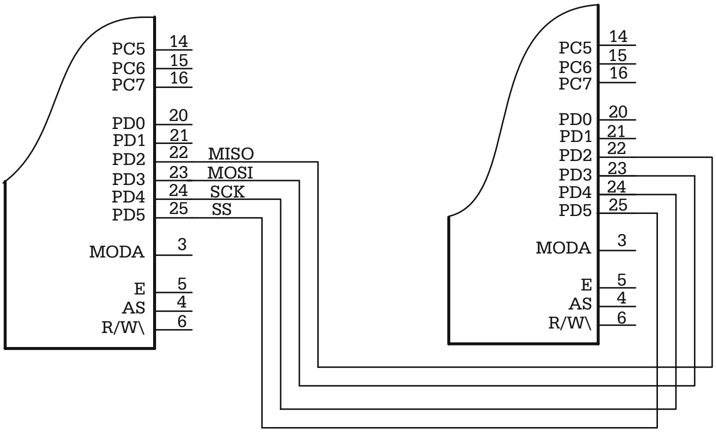


Fig. 4.3. SPI link between two 68HC11E9 microcontrollers

```
* SPI_SENDCH - sends character stored at XSPI over SPI
* returns the character received form SLAVE in RSPI
SPI_SENDCH LDAA   SPSR       ;wait for SPIF
           ANDA   #$80       ;isolate SPIF bit
           BEQ    SPI_SENDCH ;if zero, wait
           LDAA   XSPI       ;get character to send
           STAA   SPDR       ;send it
SPS05      LDAA   SPSR       ;wait for end of
                             ;transmission
           ANDA   #$80
           BEQ    SPS05
           LDAA   SPDR       ;read the byte just
                             ;received
           STAA   RSPI       ;save it to RSPI, and
                             ;return
           RTS


SPISVC     LDAA   SPSR       ;read SPSR to clear SPIF
           LDAA   SPDR
           STAA   SPIRDATA   ;a RAM location to save
                             ;received byte
           INC    QSPI       ;true QSPI
           LDAA   SPIXDATA   ;get the character to be
                             ;sent next time
           STAA   SPDR       ;place it in SPDR
           RTI               ;return from interrupt
```

Note that, at each interrupt, the SLAVE prepares the data to be delivered to MASTER at the *next* interrupt. The main program is responsible for preparing the transmission data in the variable SPIXDATA, according to the value received from MASTER in SPIRDATA. QSPI is a flag indicating that a character has been received and placed in SPIRDATA by the interrupt service routine.

4.5.2 Expanding the I/O Space Using the SPI

In some cases, when the number of MCU I/O lines is insufficient for a certain application, a possible solution to expand the I/O space is to use shift registers connected to the SPI. The following example shows how the I/O space of an ATMega8 AVR is expanded with one 8-bit input port, and one 8-bit output port. For the input port, a parallel-input serial-output shift register (74165) is used. The output port is implemented using a 4094 serial-input parallel-output register. The schematic of the circuit is presented in Fig. 4.4.

Obviously, in this SPI link, the MASTER device is the microcontroller. The serial data input of register IC2 is connected to the MOSI line of the MCU. The shift clock for this register is generated by the microcontroller on the SCK line. The load (LD) input for the 4094 register is driven by an extra output line of the micrcontroller (PB2).

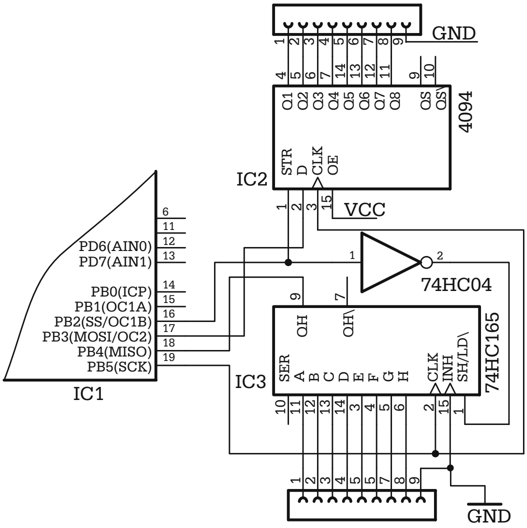


Fig. 4.4. Expanding the I/O space of ATMega8 using the SPI

When the microcontroller writes a byte into SPDR, eight clock pulses are automatically generated on the SCK line, and the content of SPDR is presented to the MOSI line, bit by bit, and shifted into IC2. When the serial transfer completes, the data in IC2 is effectively transferred to the output lines by generating a pulse 0-1-0 on LD (PB2).

The load signal SH/LD\ of the 4094 circuit is active LOW. This signal is obtained by inverting the LD signal with the gate IC6. When 74C165 senses an active LOW pulse on the SH/LD\ input, the status of the input lines is latched into the shift register, and will be shifted into the microcontroller through the MISO line on the same clock pulses SCK used to refresh the status of output registers. The processes of transmission of output data and reception of input data are simultaneous.

The software must do the following steps:

1. Generate a pulse on LD (SH/LD\) to strobe the input lines into 74C165.
2. Write the output data into SPDR.
3. Poll SPIF waiting for the end of transfer.
4. Read the input byte from SPDR, and save it into a variable.
5. Generate another pulse on LD to strobe the output data onto the parallel lines.

Here is the complete listing of a program that does this in an endless loop:

```
           .Include "M8def.inc"
           .Equ   DD_LD=2       ;portb,2 LD signal
           .Equ   DD_MOSI=3     ;portb,3 MOSI
           .Equ   DD_SCK=5      ;portb,5 SCK
           .Def   Tmp1=R16      ;define temporary data
                                ;registers
           .Def   Tmp2=R17
           .Dseg                ;DATA segment
BufT:      .Byte  1             ;byte to send
BufR:      .Byte  1             ;received byte
           .Cseg                ;start of CODE segment
           .Org   0
           Rjmp   Start
Spi_transfer:
           Sbis   Spsr,Spif     ;check SPIF
           Ret                  ;return if still busy
           Sbi    PortB,2       ;pulse LD
           Nop
           Cbi    PortB,2
           In     Tmp1,Spdr     ;Get SPI data from
                                ;previous read
           Sts    BufR,Tmp1     ;save received character
           Out    Spdr,Tmp1     ;start new transmission
           Ret
START:
           LDI    TMP1,LOW(RAMEND) ;init SP
           OUT    SPL,TMP1
           LDI    TMP1,HIGH(RAMEND)
           OUT    SPH,TMP1
           RCALL  INIT_SPI      ;init SPI interface
MAIN:
           RCALL  Spi_Transfer
; ..... other tasks here
           RJMP   MAIN
; init SPI routine
INIT_SPI:
; Configure MOSI SCK and LD
           LDI    TMP1(1<<DD_MOSI)|(1<<DD_SCK)|(1<<DD_LD)
           OUT    DDRB,TMP1
; Init SPI as master device, for 8 bit transmission and
; SCK=CK/16
           LDI    TMP1,(1<<SPE)|(1<<MSTR)|(1<<SPR0)
           OUT    SPSR,TMP1
           RET
```

4.6 Exercises

X 4.1

Draw the schematic of a SPI link between a HC11E9 (master) and an AT90S8535 (slave).

X 4.2

Draw the schematic of a circuit that expands the I/O space with 16 input lines and 16 output lines. Describe how would you connect such circuit to a HC11E9 microcontroller.

5

Using The I2C Bus

5.1 In this Chapter

This chapter continues the presentation of the serial communication interfaces with the description of the I2C bus. It contains suggestions about a possible software implementation of the I2C protocol, and examples on how to connect a 24×256 I2C memory to a microcontroller.

5.2 The Principles of Implementation of the I2C Bus

An I2C bus (Inter IC bus) consists of two signal conductors named SDA (Serial Data) and SCL (Serial Clock) that interconnect at least two devices. Each of the devices connected to the bus must be identified by a unique *address*. In principle, any of the devices connected to the bus can transmit and receive data. The device that initiates and completes a bus transfer, and generates the transfer clock, is named the MASTER.

As any of the devices connected on the bus can become MASTER at a certain time, the SDA and SCL lines must both be bi-directional. In practice open drain or open collector lines are used. The pull-up resistors have usual values between 2 K and 10 K, depending on the chosen communication speed (at high speed, the pull-up resistor has a smaller value). The idle status of the SDA and SCL lines is HIGH. A bus transfer sequence comprises the following steps:

1. The MASTER device generates a START condition on the bus.
2. The MASTER device generates eight SCL pulses controlling the SDA line in such a way that it transmits a byte which contains the SLAVE's device address and codes the type of the transfer (writing or reading). The receiver samples the SDA line while the SCL line is HIGH, and therefore data must be stable while the SCL is HIGH. SDA may change status only when SCL is LOW.
3. The MASTER device sends a ninth clock on the SCL line, while releasing the SDA data line. The device that recognized its address in the first byte transmitted

by the MASTER pulls the line LOW for the period of this clock, thus generating an ACK (acknowledge) condition.

4. The MASTER device continues to generate packages of nine SCL pulses, while the SDA is controlled either by the MASTER or by the SLAVE, depending on the type of the transfer in course. After each byte transmitted, the receiver generates an ACK condition. If the MASTER is the receiver (for example during a SLAVE byte read operation), then THE MASTER will generate ACK.
5. The process continues until MASTER generates a STOP condition and the bus returns to the idle status.

The following paragraphs describe each of the above steps.

5.2.1 The Start Transfer Condition

The START transfer condition consists of a HIGH to LOW transition of the SDA line, while the SCL clock line is maintained HIGH. The waveform corresponding to this condition is presented in Fig. 5.1.

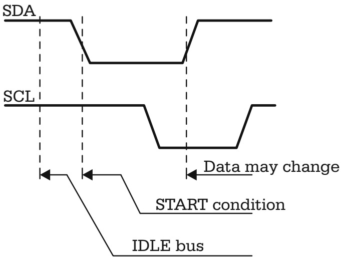


Fig. 5.1. The START TRANSFER condition on the I2C bus

5.2.2 The Data Transfer on the I2C BuS

The actual data transfer, after a START condition, begins after the SCL line is brought to LOW. At this moment, the state of the SDA data line may be modified and set to the value of the bit that follows in the transmission sequence. The receiver samples the SDA line only after the SCL line is raised HIGH. Figure 5.2 shows the waveforms for SDA and SCL for a transmission sequence containing the START condition, and the transmission of the four data bits in the sequence 1101.

The first bit transmitted is the most significant bit (MSB) of each byte.

5.2.3 The ACK Bit

After the transmission of a byte (eight SCL pulses), the MASTER device releases the SDA line and generates a ninth SCL clock. When the SCL line becomes 1, the

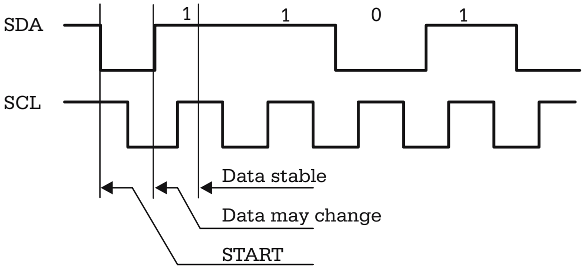


Fig. 5.2. Example of waveform for data transmission on the I2C bus

SLAVE device pulls SDA to LOW, confirming by this the reception of the byte. In all situations the data line is controlled by the receiver during the ACK bit time. In case of a read byte operation, the device that sends ACK is the receiver, i.e. the MASTER device.

5.2.4 The STOP Condition

A STOP transmission condition is generated when a LOW to HIGH transition of the SDA line occurs while the SCL line is HIGH. After the STOP condition, both SDA and SCL bus lines return to the idle state, and a new transmission must be preceded by a START sequence.

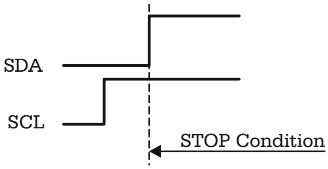


Fig. 5.3. The STOP TRANSFER condition on the I2C bus

In practice, the I2C protocol is more complicated, meaning that it allows operation in multimaster mode. In this case, an arbitration mechanism is required to allocate the bus in situations when two or more devices try to become MASTER at the same time. In this book only single-master I2C buses are discussed.

5.3 A Software Implementation of the I2C Protocol

There is a variety of circuits available, which are designed to communicate with a microcontroller via the I2C bus: memories, LCD displays, A/D and D/A converters, etc. Usually these are delivered in small capsules, with extremely low power consumption and require only two I/O lines of the microcontroller for connection.

Besides that, the access to any device connected to the I2C bus can be controlled by a library of *reusable* software modules. All these advantages recommend the I2C devices for many microcontroller applications, especially when the use of an expanded microcontroller structure is difficult or impossible.

The SDA and SCL lines are controlled by software through simple subroutines. The following example, written for HC11, illustrates the generation of the START condition on the I2C bus:

```
I2CSTART   BSET   PORTA,SCL  ;make sure the clock is
                              HIGH
           JSR    DELAY      ;wait a few microseconds
           BCLR   PORTA,SDA  ;HIGH to LOW transition of
                              SDA
           JSR    DELAY
           RTS
```

Similarly, a STOP condition is generated in the following way:

```
I2CSTOP    BSET   PORTA,SCL  ;make sure the clock is
                              HIGH
           JSR    DELAY      ;wait a few microseconds
           BSET   PORTA,SDA  ;LOW to HIGH transition of
                              SDA
           JSR    DELAY
           RTS
```

5.4 Accessing 24C256 Memory Devices

In the example presented in Fig. 5.4, the two 24Cxx circuits have the address lines A0, A1, A2, hardwired to different values. In this configuration, the microcontroller is, obviously, the MASTER device. After the START sequence, the microcontroller sends a byte that contains the address of the SLAVE device and the type of the operation: read or write. The structure of this byte is as follows:

MSB							LSB
1	0	1	0	A2	A1	A0	R/W

R/W\ = 1 indicates that a read operation follows.
R/W\ = 0 indicates a memory write operation

The upper-nibble of this byte is always 1010 for this type of memory. The A2, A1, A0 bits contain the SLAVE's device address. After the transmission of the byte with the SLAVE address, the device that recognizes its address pulls the SDA line LOW, for the duration of the ninth clock, thus generating the ACK condition.

In the case of a write operation to the 24C256 memory, after the byte containing the SLAVE address, the microcontroller sends two bytes containing the address of

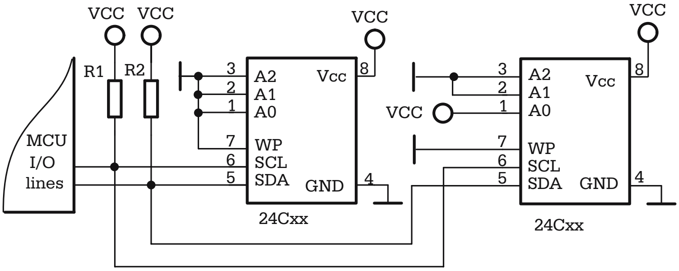


Fig. 5.4. Connecting I2C memory devices to a microcontroller

the location that is to be written, followed by a byte containing the value to be written in the memory. After each received byte, the memory sends an ACK bit, and, after the last ACK, the microcontroller generates a STOP condition to complete the write operation. If, instead of generating a STOP condition, the microcontroller sends a new byte, this will be written in the memory to the next address. In this manner, up to 64 bytes can be written consecutively. This type of operation is named "*page write mode*".

To read a byte from a random address (random read operation) the microcontroller initiates a write sequence, in which it sends the byte with the SLAVE address, followed by two bytes with the address of the location that is to be read. Then, a new START is generated, followed by the SLAVE address byte, with the R/W\ bit set to 1, indicating a read operation. After the ACK is received from the SLAVE, the microcontroller sends eight more SCL pulses to get the read byte from memory, then it generates a STOP condition.

If, instead of generating the STOP condition, the microcontroller sends an ACK, followed by eight SCL clock pulses, it reads the byte from the next address. The process continues in the same way. At each ACK generated by the microcontroller, the internal address counter of the memory is incremented, and the process continues with reading of the next byte. The process completes when MCU generates a STOP condition.

The accompanying CD contains examples of subroutines to access I2C memories, written for HC11 and 8051.

5.5 Exercises

SX 5.1

Is the I2C communication synchronous or asynchronous?

Solution

Since there is a dedicated line for transmission of the clock, it follows that the transmission on the I2C bus is synchronous. The START and STOP conditions delimit data packets, not data bytes as in the case of the asynchronous communication.

SX 5.2

SPI and I2C are both synchronous communication methods. Which one is faster?

Solution

In theory at least, SPI is faster, because it is full duplex. At a given frequency of the transmission clock, the data volume transmitted on SPI is double the volume transmitted on I2C, because data is transmitted in both directions on the MOSI and MISO lines.

SX5.3

Draw the waveforms for the SDA and SCL in case of the transmission of a START transfer condition, followed by the transmission of the byte with the value \$A2 = 10100010b, and ACK.

Solution

See Fig. 5.5 for the idealized waveforms of SDA and SCL for this example.

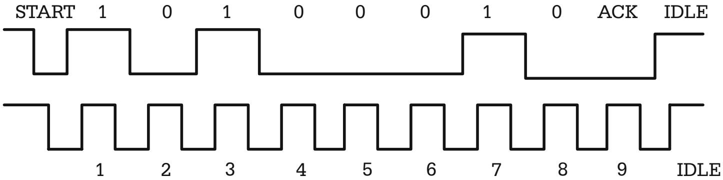


Fig. 5.5. Solution of the exercise SX 5.3

6

Using the MCU Timers

6.1 In this Chapter

This chapter contains a description of the timer system of microcontrollers, including the general-purpose timer, the PWM timer, and the watchdog.

6.2 The General Structure and Functions of the Timer System

Timing is essential for the operation of microcontroller systems, either for generating signals with precisely determined duration, or for counting external events. For this reason, the timer subsystem is present in all microcontroller implementations, and covers a wide range of functions including:

- Generation of precise time intervals
- Measurement of duration of external events
- Counting external events.

Most microcontrollers are provided with dedicated timers, or use the general-purpose timer to implement the following additional functions:

- Real time clock
- Generation of Pulse Width Modulated (PWM) signals
- Watchdog for detecting program runaway situations.

Although there are significant variations between different implementations of the general-purpose timer in different microcontrollers, there are many similarities in the principles of operation and the structure of the timer subsystem.

Figure 6.1 shows a general block diagram of the timer system, illustrating the principles of implementation of most MCU timers.

The central element of the timer subsystem is a counter, TCNT (8 or 16-bits in length), which may be read or (sometimes) written by software. The clock for TCNT is obtained either from the system clock, divided by a programmable prescaler, or an external clock applied to one of the MCU pins. The software control upon the timer is

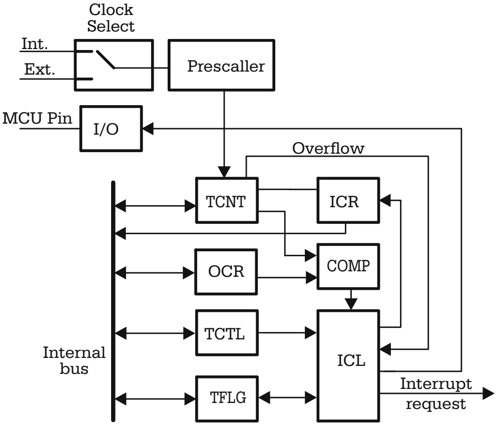


Fig. 6.1. General block diagram of the timer subsystem

performed by means of the control register TCTL and information regarding various events related to the timer may be read from the status register TFLG.

Several operating modes are possible for the timer:

Timer overflow. In this mode, the event of interest is when the TCNT counter reaches its maximum count and returns to zero on the next clock pulse. The overflow signal that marks this event is applied to the interrupt control logic (ICL), which may generate an interrupt request to the CPU.

The time interval between two successive overflows is controlled either by modifying the frequency of the input clock applied to TCNT, or by writing to TCNT an initial value for counting.

- *Input capture*. In this operating mode, the contents of TCNT at the moment of the occurrence of an external event, defined by the edge of an input signal, is transferred in a *capture register* (ICR), and an interrupt request may be generated. By comparing two consecutive values capture by the ICR, it is possible to determine the time interval between the two external events.
- *Output compare*. In this operating mode, the contents of TCNT are continuously compared by hardware to the contents of the OCR (Output Compare Register) by means of the digital comparator COMP. When the contents of the two registers match, an interrupt request may be generated. Optionally, the compare match can be programmed to change the status of one or more output lines.
- *External events counter*. In this operating mode, the input of TCNT is connected to one of the MCU input lines, and TCNT counts the pulses associated with the external events. The software is informed about the recorded number of external events by reading TCNT.

6.3 Distinctive Features of the General-Purpose Timer of HC11

The 16-bit TCNT counter of HC11 can count on the internal clock only, and only upwards. It can be read by software, but cannot be cleared or written. The prescaler is a programmable 4-bit counter, which divides the system clock E by 1, 4, 8, or 16.

There are four 16-bit output compare registers (OCR), called TOC1, TOC2, TOC3, and TOC4, three input capture register (ICR), called TIC1, TIC2, and TIC3, and an additional register, which can be configured by software to operate as a fifth OCR register, under the name of TOC5, or as a fourth input capture register TIC4. The various timer functions are associated with the I/O lines of PORT A, as shown in Table 6.1.

Table 6.1. Alternate functions of the I/O lines of PORT A

PORTA	Associated timer function
PA0	input capture 3
PA1	input capture 2
PA2	input capture 1
PA3	input capture 4/output compare 5
PA4	output compare 4
PA5	output compare 3
PA6	output compare 2
PA7	output compare 1

6.3.1 The Control and Status Registers of the HC11 Timer

Although the counter TCNT, and the prescaler are unique, the presence of the eight ICR/OCR registers, each having distinct status flags, associated I/O lines, along with the possibility to generate distinct interrupt requests, makes the HC11 timer act as eight different timers. Therefore, the number of control and status registers associated with the timer is higher than the average number of registers of a peripheral interface. For clarity of the presentation, the registers of the timer system are described in connection with the basic operating modes of the timer.

6.3.1.1 The Timer Overflow Operating Mode

The prescaler is controlled by the bits PR1:PR0 in register TMSK2 (Timer Interrupt Mask Register 2, bits [1:0]), which select the dividing rate of the system clock E to obtain the clock for TCNT.

After the transition of the counter TCNT from $FFFF to $0000, a flag is set by hardware. This is the TOF (Time Overflow Flag) bit in the status register TFLG2 (bit 7). If the associated local interrupt mask, TOI (Time Overflow Interrupt enable) from register TMSK2 (bit 7), is set, then an interrupt request is generated.

Note that the interrupt service routine must clear TOF by writing 1 in the corresponding position of the TFLG2 register.

6.3.1.2 The Input Capture Operating Mode

In order to use the input capture feature, the first step required is to configure the corresponding line of PORTA as input, by clearing the corresponding bit in DDRA. For IC4, then the IC4/OC5 control bit in the PACTL register must be set to 1. This bit is cleared at RESET, thus OC5 is enabled.

The next step is to select the edge of the input signal that triggers the capture. For each of the four capture inputs, two bits are assigned in the TCTL2 register (Timer Control Register 2), called EDGxB and EDGxA,.

The structure of TCTL2 is as follows:

TCTL2	7	6	5	4	3	2	1	0
	EDG4B	EDG4A	EDG1B	EDG1A	EDG2B	EDG2A	EDG3B	EDG3A
RESET	0	0	0	0	0	0	0	0

And the effect of [EDGxB:EDGxA] is described in Table 6.2.

Table 6.2. Selection of the capture edge for input capture

EDGxB	EDGxA	Type of capture
0	0	capture disabled
0	1	capture on rising edge
1	0	capture on falling edge
1	1	capture on both edges

The occurrence of an edge with the selected polarity on the input line associated with the input capture timer sets a flag in the TFLG1 register (Timer Interrupt Flag Register 1). This flag can be polled by software, or may generate an interrupt if the input capture interrupt is enabled by setting the local mask bit in the TMSK1 register (Timer Interrupt Mask Register 1).

The structure of the TFLG1 and TMSK1 registers is presented below:

TFLG1	7	6	5	4	3	2	1	0
	OC1F	OC2F	OC3F	OC4F	I4/O5F	IC1F	IC2F	IC3F
RESET	0	0	0	0	0	0	0	0

TMSK1	7	6	5	4	3	2	1	0
	OC1I	OC2I	OC3I	OC4I	I4/O5I	IC1I	IC2I	IC3I
RESET	0	0	0	0	0	0	0	0

ICxF is the flag that indicates an event on the ICx input, while OCxF reports a match between the contents of TCNT and OCxR. ICxI and OCxI are the local interrupt masks that, when set to 1 by software, allow ICxF and OCxF to generate interrupts.

The ICxF and OCxF flags are cleared by writing 1 to the corresponding position of the TFLG1 register. The interrupt service routine must clear the flag that generated the interrupt, otherwise a new interrupt is generated, right after the execution of the RTI (Return from Interrupt) instruction.

6.3.1.3 The Output Compare Operating Mode

The software initialization sequence for the output compare timers is very similar to that required by the input capture timers. It starts by configuring the associated lines of PORT A as output lines by writing 1 to the corresponding bits of DDRA.

Then the software must specify the action to be taken on compare match. The register TCTL1 (Timer Control Register 1) serves this purpose.

TCTL1	7	6	5	4	3	2	1	0
	OM2	OL2	OM3	OL3	OM4	OL4	OM5	OL5
RESET	0	0	0	0	0	0	0	0

OMx and OLx are, respectively, Output Mode and Output Level control bits associated to the OCx output. The effect of these bits is described in Table 6.3.

Table 6.3. The effect of the control bits in TCTL1

OMx	OLx	Action taken on compare match
0	0	Timer disconnected from output pin
0	1	Toggle OCx output line
1	0	Clear OCx output line to 0
1	1	Set OCx output line to 1

The event flags associated to OCx and the interrupt mask bits are located in the registers TFLG1 and TMSK1, described in the previous section.

To increase the flexibility of the HC11 timer, the TOC1 timer has been provided with the *capability to simultaneously control* two or more of the PORTA lines associated to the timer, i. e. PORTA [3–7]. Two additional registers have been provided for this purpose, called OC1M (OC1 Mask) and OC1D (OC1 Data).

Only five bits are implemented in these registers, as follows:

OC1M	7	6	5	4	3	2	1	0
	OC1M7	OC1M6	OC1M5	OC1M4	OC1M3	–	–	–
RESET	0	0	0	0	0	–	–	–

OC1D	7	6	5	4	3	2	1	0
	OC1D7	OC1D6	OC1D5	OC1D4	OC1D3	–	–	–
RESET	0	0	0	0	0	–	–	–

- OC1M indicates the lines of PORTA lines to be affected at the next OC1 match, and OC1D contain the data to be written to PORTA.

  OC1Mx = 1 – line x of PORTA will be written with OC1D value at the next compare match.
  OC1Mx = 0 – line x of PORTA will not be affected by OC1.

Note that TOCx interrupts may be used without affecting the associated outputs of PORTA (OMx = 0, OLx = 0), only for generating interrupts at precise time intervals.

6.3.1.4 Counting External Events

The main timer of HC11 cannot count on the external clock. To solve the problem of counting external events, an additional 8-bit counter, called the Pulse Accumulator, has been provided. This is a supplementary, simplified timer, without the input capture and output compare registers, but having the ability to count either external pulses, applied on an input pin, or an internal clock.

The line PA7 is used as the pulse accumulator input (PAI) for the external clock. When configured to count on the internal clock, this clock has a fixed frequency E/64, and the PAI line is used to enable/disable counting.

The control register of this timer is PACTL, which has the following structure:

PACTL	7	6	5	4	3	2	1	0
	–	PAEN	PAMOD	PEDGE	–	I4/O5	RTR1	RTR0
RESET	0	0	0	0	0	–	–	–

- PAEN – Pulse Accumulator Timer Enable. Writing 1 in this position enables the entire subsystem.

- PAMOD – Pulse Accumulator Mode

  PAMOD = 0. The pulse accumulator operates as an event counter, counting pulses applied to the PAI input.
  PAMOD = 1. The pulse accumulator operates in Gated Time Accumulation mode, counting on an internal clock obtained by dividing the system clock E by 64. When PAMOD = 1, counting of the internal clock pulses is enabled by the logic level on the PAI line. PAMOD works in conjunction with the bit PEDGE, as shown in Table 6.4.

- PEDGE – Pulse Accumulator Edge control. In the Event Counter operating mode (PAMOD = 0), this bit selects the *edge* of the input signal that increments the counter. In Gated Time Accumulation mode (PAMOD = 1), PEDGE selects the level of the signal on the PAI input, which inhibits counting of the internal clock.

Table 6.4. Pulse accumulator operating modes

PAMOD	PEDGE	Action on timer
0	0	PA counts on falling edge of PAI
0	1	PA counts on rising edge of PAI
1	0	PAI = 0 inhibits counting.
1	1	PAI = 1 inhibits counting.

The other bits in PACTL refer to other subsystems, or are unimplemented.

There are two status bits associated with the pulse accumulator timer interface, located in TFLG2:

- PAOVF – Pulse Accumulator Overflow Flag. This is automatically set when PA overflows from $FF to $00, regardless of the clock (internal or external) selected for counting. PAOVF is cleared by writing 1 in the corresponding position (bit 5) of the TFLG2.
- PAIF – Pulse Accumulator Input Edge Flag. This bit is automatically set at the detection of an edge (selected by PEDGE) of the signal on the PAI input. It is cleared by writing 1 in the corresponding condition (bit 4) of TFLG2.

These two flags have associated interrupt mask bits in the TMSK2 register. When PAOVFI = 1 (bit 5 from TMSK2), the setting of PAOVF generates an interrupt. Similarly, when PAII = 1 (bit 4 from TMSK2), an interrupt is generated at the occurrence of the selected edge of the PAI input.

6.3.2 Exercises Regarding the Use of the General-Purpose Timer of HC11

SX 6.1

Write the initialization routine that enables the interrupts at the detection of a rising edge of a signal applied on PA0.

Solution

PA0 is associated with the input capture timer IC3 (see Table 6.1). The initialization sequence must configure this bit of PORTA as the input, along with the following additional operations:

- Select the rising edge of the signal on PA0, by writing the bits [EDG3B:EDG3A] in TCTL2 with [0:1].
- Enable TIC3 interrupts by setting to 1 the local mask IC3I (IC3 interrupt enable) in TMSK1

Here is the program sequence that performs these operations:

```
ITIC3     BCLR   DDRA,$01    ;PA0 input
          LDAA   #$01        ;[EDG3B:EDG3A]=[0:1]
          STAA   TCTL2       ;select rising edge
          BSET   TFLG1,$01   ;clear IC3F if any
          BSET   TMSK1,$01   ;enable TIC3 interrupts
```

SX 6.2

Knowing that the external oscillator frequency is 8 MHz, write the initialization sequence and the interrupt routine to generate a 500-Hz clock on PA5.

Solution

PA5 is associated with TOC3. The initialization sequence must configure PA5 as output, define the action to be performed on the OC3 output at compare match, by writing [OM3:OL3] bits from TCTL1, and enable TOC3 interrupts. The interrupt service routine must clear the interrupt flag, and prepare the next interrupt by writing a new value in TOC3. For an 8-MHz frequency of the external oscillator the internal E clock has a frequency of 2 MHz (0.5 microseconds/period).

Since [PR1:PR0] bits in TMSK2 are cleared at RESET, the prescaler is configured to divide E by 1. An output frequency of 500 Hz, corresponds to a period of 2 milliseconds, i.e. 4000 periods of the E clock. The interrupt routine must add the constant 4000 to the current value of the register TOC3, and write the TOC3 register with the result. Thus, the next moment when TCNT matches the contents of TOC3 comes after 4000 E clock periods, which is equivalent to 2 ms.

The output line associated to TOC3 must be programmed to toggle at every compare match, by writing the control bits [OM3:OL3] in TCTL1 with [0:1].

Here is the initiation sequence that matches these requirements:

```
ITOC3     BSET   DDRA,$20    ;PA5 output
          LDAA   #$10        ;[OM3:OL3]=[0:1]
          STAA   TCTL1       ;toggle output selected
          BSET   TFLG1,$20   ;clear IC3F if any
          BSET   TMSK1,$20   ;enable TOC3 interrupts
```

And the interrupt service routine for TOC3 is:

```
TOC3SVC   BSET   TFLG1,$20   ;clear OC3F
          LDD    TOC3        ;get current value of TOC3
          ADDD   #4000       ;add 4000 - 2 ms more
          STD    TOC3        ;update TOC3
          RTI                ;return from interrupt
```

SX 6.3

Write the initialization sequence for the pulse accumulator timer, so that it generates an interrupt every tenth rising edge of the signal applied on PA7.

Solution

The required initialization sequence must perform the following operations:

- Configure PORTA bit 7 as the input line.
- Enable the pulse accumulator timer.
- Select the event counter operating mode.
- Select the rising edge of the input signal as the active edge.
- Initialize the counter PACNT with 246, so that the tenth pulse produces an overflow.
- Enable the PAOVF interrupt by setting the PAOVFI bit in TMSK2.

The resulting initialization sequence looks like this:

```
INIT_PA    BCLR   DDRA,$80    ;PA7 input
           LDAA   #$50        ;PAEN = 1, PAMOD = 0, PEDGE = 1
           STAA   PACTL       ;write control register
           LDAA   #246
           STAA   PACNT       ;init counter
           BSET   TFLG2,$20   ;clear PAOVF if any
           BSET   TMSK2,$20   ;enable PAOVF interrupts
```

The interrupt routine must do the following:

- Clear the PAOVF flag, by writing 1 in position 5 of TFLG2.
- Write the constant 246 to PANCT, so that the next overflow occurs at the tenth pulse on PAI.

Below is the listing of the interrupt service routine that does this:

```
PAOVF_SVC  BSET   TFLG2,$20   ;clear PAOVF
           LDAA   #246        ;write 246 to PACNT
           STAA   PACNT
           ...
           RTI                ;return from interrupt
```

6.4 Distinctive Feature of the Timer of the AVR Microcontrollers

Unlike the HC11 family, where the timer subsystem remains the same for all family members, for the AVR there may be significant differences in the implementation of the timer from one family member to another. This section contains the description of the timer subsystem of the microcontroller AT90S8515.

This, in fact, contains two distinct timers, named Timer0 and Timer1.

6.4.1 The 8-Bit Timer/Counter Timer0

Timer0 is built around an 8-bit counter TCNT0. The clock for TCNT0 is selected by means of the control bits [CS02:CS01:CS00] in TCCRO (Timer Counter Control Register 0), according to Table 6.5.

Table 6.5. Clock selection for TIMER0 of AT90S8515

CS02	CS01	CS00	TCNT0 clock
0	0	0	Stop, the Timer/Counter0 is stopped.
0	0	1	CK
0	1	0	CK/8
0	1	1	CK/64
1	0	0	CK/256
1	0	1	CK/1024
1	1	0	External Pin T0, falling edge
1	1	1	External Pin T0, rising edge

The bits [CS02:CS01:CS00] are located in the positions [2:1:0] in the TCCR0 register. The rest of the bits of this register are not implemented. When the external clock is selected for TCNT0, this is applied on the T0/B0 input. The I/O line must be configured as input by clearing bit 0 of DDRB.

The only event reported about TIMER0 is the overflow. When the counter TCNT0 changes status from $FF to $00, the flag TOV0 (bit 1 from the TIFR register (Timer Interrupt Flag Register)) is set. This flag is cleared, just like the HC11 interrupt flags, by writing 1 to the respective position of the TIFR.

When set, TOV0 can generate an interrupt, if the interrupts are enabled by setting the TOIE0 bit (Timer Overflow Interrupt Enable 0) in TIMSK (bit 1).

Note the many similarities between the structure and operation of Timer0 of the AVR and the pulse accumulator timer of HC11. The presence of the prescaler increases the flexibility of Timer0 compared to pulse accumulator timer, but the lack of the gated time accumulation option is a minus.

6.4.2 The 16-Bit Timer/Counter Timer1

The structure of Timer1 of the AVR is very similar to the main timer of HC11. The central element of this timer is the 16-bit counter TCNT1. This is accessible for read and write operations on the 8-bit internal bus, as two registers: TCNT1H and TCNT1L. This is accompanied by two output compare registers, called OCR1A and OCR1B, and an input capture register, called ICR1.

The I/O lines associated with Timer1 are ICP (Input Capture Pin), T1, (the input of the external clock), and OC1A, OC1B (Output Compare 1 A, B). For AT90S8515, ICP and OC1B are available as dedicated pins, while OC1A and T1 share the MCU pins with PD4 and PB1, respectively. Refer to the data sheet for other AVR models.

6.4.2.1 The Timer Overflow Operating Mode

The clock for TCNT1 is selected by the bits [CS12:CS11:CS10] in the TCCR1B register, as shown in Table 6.6.

Table 6.6. The effect of programming [CS12:CS11:CS10]

CS12	CS11	CS10	TCNT1 clock
0	0	0	Stop, the Timer/Counter1 is stopped.
0	0	1	CK
0	1	0	CK/8
0	1	1	CK/64
1	0	0	CK/256
1	0	1	CK/1024
1	1	0	External Pin T1, falling edge
1	1	1	External Pin T1, rising edge

The flag that indicates the change of status of TCNT1 from $FFFF to $0000 is called TOV1 (Timer1 Overflow) and is located in bit 7 of TIFR (Timer Interrupt Flag Register). TOV1 = 1 can generate an interrupt, if the mask bit TOIE1 (Timer Overflow Interrupt Enable- bit 7 from TIMSK) is set to 1.

6.4.2.2 The Input Capture Operating Mode

This operating mode is controlled by two bits in the register TCCR1B (Timer/Counter1 Control Register B):

- ICNC – Input Capture Noise Canceler is bit 7 of TCCR1B.

  ICNC1 = 0. The capture is triggered by the selected edge of the ICP (Input Capture Pin), without further checking.
  ICNC1 = 1. The ICP line is sampled four times, at the frequency of the clock CK, after the active edge is detected, and the capture is enabled only if the ICP line is stable for the duration of the four samples.

- ICES1 – Input Capture Edge Select is bit 6 of TCCR1B.

  ICES1 = 0. Capture on the falling edge.
  ICES1 = 1. Capture on the rising edge.

The input capture flag ICF1 is bit 3 in TIFR (Timer Interrupt Flag Register). When set, ICF1 can generate an interrupt, if TICIE1 (Timer Input Capture Interrupt Enabled), bit 3 in TIMSK, is set to 1.

An interesting distinctive feature is that the ICF1 can be cleared in two ways, either by writing 1 in the corresponding position from TIFR, or by hardware, at the execution of the jump to the interrupt vector.

Another unique feature of AVR microcontrollers is that the capture can be triggered by the transitions of the output of the built-in analog comparator. See Chap. 7 for details on the use of this feature.

6.4.2.3 The Output Compare Operating Mode

Upon detection of a compare match between TCNT1 and one of the registers OCR1A or OCR1B, three things can happen:

- A flag is set in the TIFR register. The flags associated with the two output compare registers are called OC1FA and OC1FB.
- If the corresponding interrupt mask bit OCIE1A or OCIE1B is set, then an interrupt request is generated.
- The status of the output line associated with the OCR register can be changed, according to the control bits [COM1A1:COM1A0] or [COM1B1:COM1B0] (Compare Output Mode select bits) in the control register TCCR1A. The four combinations of these two bits correspond to the situations in Table 6.7.

Table 6.7. Action executed on compare match

COM1x1	COM1x0	Action on OC1x pin (x = A,B)
0	0	Timer/Counter1 disconnect from output pin OC1x
0	1	Toggle OC1x output line
1	0	Clear OC1x output line to zero
1	1	Set OC1x output line to one

The AVR microcontrollers allow the counter TCNT1 to be automatically cleared by hardware after a compare match. This option is controlled by the CTC1 (Clear Timer/Counter on Compare Match) bit in the TCCR1B register.

6.4.3 Synopsis of the Timer I/O Registers of AT90S8115

- TCNT0 – Timer0 8-bit counter
- TCCR0 – Timer Counter Control Register 0

TCCR0	7	6	5	4	3	2	1	0
	–	–	–	–	–	CS02	CS01	CS00
RESET	0	0	0	0	0	–	–	–

- TCNT1 – Timer1 16-bit counter
- TCCR1A – Timer/Counter1 Control Register A

TCCR1A	7	6	5	4	3	2	1	0
	COM1A1	COM1A0	COM1B1	COM1B0	–	–	PWM11	PWM10
RESET	0	0	0	0	0	0	0	0

- TCCR1B – Timer/Counter1 Control Register B

TCCR1B	7	6	5	4	3	2	1	0
	ICNC1	ICES1	–	–	CTC1	CS12	CS11	CS10
RESET	0	0	0	0	0	0	0	0

- OCR1A and OCR1B – Timer Output Compare Registers A and B
- ICR1 – Timer Input Capture Register
- TIFR – Timer Interrupt Flag Register

TIFR	7	6	5	4	3	2	1	0
	TOV1	OCF1A	OCIFB	–	ICF1	–	TOV0	–
RESET	0	0	0	0	0	0	0	0

- TIMSK – Timer Interrupt Mask Register

TIMSK	7	6	5	4	3	2	1	0
	TOIE1	OCIE1A	OCIE1B	–	TICIE1	–	TOEI0	–
RESET	0	0	0	0	0	0	0	0

6.4.4 Summary of the Unique Features of the AVR Timer

- 10-bit prescaler.
- TCNT1 is read/write.
- TCNT1 can count on the external clock.
- TCNT1 can be automatically cleared at compare match.
- Input capture can be triggered by the built-in analog comparator.

6.4.5 Exercises Regarding the Use of AVR Timers

SX 6.4

Starting from an 8-MHz clock, use Timer0 to generate interrupts at 1-ms intervals.

Solution

The frequency of 8 MHz corresponds to a period of 0.125 microseconds. This means that the interrupts must occur at intervals of 8000 periods of the clock CK.

$8000 = 64 \times 125$, so if the prescaler is programmed to divide by 64, then TCNT0 must count 125 pulses before generating an overflow interrupt. Thus, TCNT0 must be initialized with the value $256 - 125 = 131$.

The initialization sequence must set TOIE0 (bit 1) in TIMSK to enable the TOV0 interrupt. The interrupt service routine must reload TCNT0 with this value.

Here is the code to does all this:

```
*MASTER SPI initialization routine
           .DEF   TEMP1=R16     ;definitions
           .DEF   R_INT=R1      ;use r1 only for this
                                ;interrupt
           .EQU   KTOVF=131     ;constant to load in TCNT0

INIT_T0:   LDI    TEMP1,$03     ; prescaler divide by 64
           OUT    TCCR0,TEMP1
           LDI    TEMP1,KTOVF   ;init TCNT0
           MOV    R_INT,TEMP1
           OUT    TCNT0,R_INT
           LDI    TEMP1,$02
           OUT    TIFR,TEMP1    ;clear flag if any
           OUT    TIMSK,TEMP1   ;enable interrupt
           RET                  ;end of initializations
;The interrupt service routine starts here
T0_ISR     OUT    TCNT0,R_INT   ;reload counter
           ....
           RETI                 ;return from interrupt
```

SX 6.5

Starting from an 8-MHz clock, use Timer1 in output compare mode to generate on OC1A a 500-Hz clock having 50% duty cycle.

Solution

For a 500-Hz clock, OC1A must toggle two times faster, i.e. at 1 KHz. The initialization sequence must select the clock for TCNT1, by writing the bits [CS12:CS11:CS10] in TCCR1B with [0:1:0], which corresponds to a division factor of 1 for CK.

Besides that, the CTC bit in TCCR1B must be set to force clearing of TCNT1 after each compare match. The OCR1A register must be initialized with 8000, and the action upon OC1A must be set to ‘toggle’, by writing $40 in TCCR1A.

Finaly, the compare match interrupt for OC1A must be enabled by writing $40 in TIMSK. The interrupt routine is only needed to clear the OCF1A flag, because TCNT1 is automatically cleared at compare match, and OCR1A remains unchanged after initialization.

The program sequence that executes the operations mentioned above is:

```
*MASTER SPI initialization routine
            .DEF   TEMP1=R16      ;definitions
            .EQU   KH=$1F         ;higher byte of constant
            .EQU   KL=$40         ;lower byte of constant
INIT_T1:    LDI    TEMP1,$09      ;prescaler divide by 1
            OUT    TCCR1B,TEMP1   ;and CTC1=1
            LDI    TEMP1,$40      ;toggle OC1A
            OUT    TCCR1A,TEMP1

            LDI    TEMP1,KH       ;write higher byte first !!
            OUT    OCR1AH,TEMP1
            LDI    TEMP1,KL
            OUT    OCR1AL,TEMP1
            LDI    TEMP1,$40      ;clear flag
            OUT    TIFR,TEMP1
            OUT    TIMSK,TEMP1    ;and enable interrupt
            RET
```

6.5 Distinctive Features of the Timer System of the 8051 Microcontrollers

The timer of the 8051 family of microcontrollers does not have the output compare and the input capture features. In the standard configuration, there are two timers, named Timer0 and Timer1, each having as central element a 16-bit counter, called T0 and T1, respectively.

These are capable of counting, on an internal or external clock, and are accessible from the internal bus for read and write, as two 8-bit registers: TH0–TL0 for Timer0, TH1–TL1 for Timer1.

The only event reported by timers to the CPU is the timer overflow condition.

The logic diagram of the circuit for the clock selection and control is presented in the Fig. 6.2.

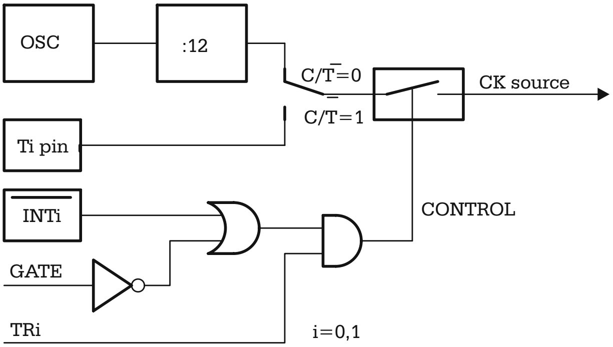


Fig. 6.2. Logic diagram of the clock control circuit for the 8051 timer

6.5.1 The Control and Status Registers of the Timer

Timer1 can operate in three distinct modes and Timer0 in four modes. The control bits in register TMOD select the timer operating mode and the clock applied to the counter. The structure of TMOD is detailed below:

TMOD	7	6	5	4	3	2	1	0
	GATE	C/T	M1	M0	GATE	C/T	M1	M0
RESET	0	0	0	0	0	0	0	0

The two nibbles of this register are identical. The lower nibble refers to Timer0, and the upper nibble refers to Timer1.

[M1:M0] Mode select bits. These bits control the operating mode of the timer as described in Table 6.8.

Table 6.8. Selection of the operating mode for 8051 timers

M1	M0	Operating mode
0	0	Mode 0 – 13-bit timer/counter
0	1	Mode 1 – 16-bit timer/counter
1	0	Mode 2 – 8-bit auto reload timer/counter
1	1	Mode 3 for Timer0, Timer 1 stopped

- C/T – Counter/Timer select bit.

  C/T = 1 selects the external clock applied on T0 for Timer0, or T1 for Timer1.
  C/T = 0 selects an internal clock having the frequency $f_{\mathrm{OSC}}/12$.

- GATE – Gated operation control bit.
  GATE = 1 Counting is enabled by a logic level HIGH on the input pin INTi, associated with the timer
  GATE = 0. Counting is only conditioned by the TRi bit in TCON (Timer Control Register) register.

TCON – The timer control register has the following structure:

TCON	7	6	5	4	3	2	1	0
	TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0
RESET	0	0	0	0	0	0	0	0

- TFi – Timer i overflow flag. These bits are set by hardware when the counter overflows, and are automatically cleared at the execution of the associated interrupt service routine.

- TRi – Timer Run control. This is set and cleared by software to start/stop counting. Refer to Fig. 6.2 for a description of the effect of this control bit.

- ITI, IEi – These control bits are not related to the timer system.

The overflow flags TF0, TF1 can generate interrupt requests, if the interrupts are enabled by setting the bits ET0, ET1 in the IE (Interrupt Enable) register. The IE register controls all the possible interrupts of 8051 and has the following structure:

IE	7	6	5	4	3	2	1	0
	EA	–	–	ES	ET1	EX1	ET0	EX0
RESET	0	0	0	0	0	0	0	0

- EA – Enable All. This is the global interrupt mask.

  EA = 0 all interrupts are disabled.
  EA = 1 each interrupt can be enabled/disabled individually.

- ES – Enable Serial Port Interrupt

  ET1 – Enable interrupts from Timer1.
  ET1 = 1 The flag TF1 generates an interrupt request when set.

- EX1 – Enable external interrupts from INT1
- ET0 – Enable interrupts from Timer0 (TF0)
- EX0 – Enable external interrupts from INT0

6.5.2 Description of the Timer Operating Mode 0

In operating mode 0, the counter Ti is 13-bits wide, with 5 bits in TLi, and 8 bits in THi. The clock can be either internal (*timer mode*) or external (counter mode). Selection between the internal or external clock is made by means of the C/T control bit in the TMOD register.

The overflow condition occurs at the transition from status $1FFF to $0000, and is indicated by setting the corresponding flag TFi to 1.

6.5.3 Description of the Timer Operating Mode 1

The logical diagram describing the 8051 timers operating in modes 0 and 1 is presented in Fig. 6.3. The only difference between mode 1 and mode 0 is that, in mode 1, the counter is 16-bit wide, and the overflow condition occurs at the transition from status $FFFF to $0000.

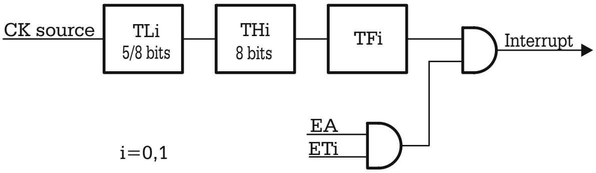


Fig. 6.3. 8051 timer in mode 0/1

6.5.4 Description of the Timer Operating Mode 2

In mode 2, the lower half of the counter (TLi) operates as an 8-bit counter, while the upper half (THi) acts as a register that holds the reload value for TLi.

At overflow, the value written to THi is automatically transferred in TLi, and counting continues from this value. The logic diagram for the operating mode 2 of the 8051 timer is presented in Fig. 6.4. The clock source for the counter is selected as shown in Fig. 6.2.

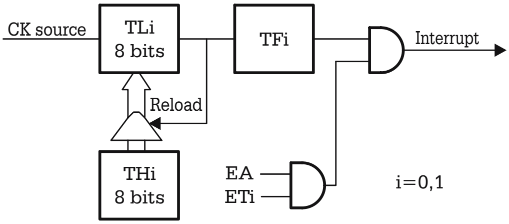


Fig. 6.4. The 8051 timer operating in mode 2

6.5.5 Description of the Timer Operating Mode 3

This operating mode is specific only for Timer0. In mode 3, the counter T0 is split into two 8-bit counters that count on different clocks. The lower half, TL0, operates in a way similar to modes 0, and 1, but the length of the counter is limited to 8 bits. At overflow, the flag TF0 is set, and an interrupt is generated if $\mathrm{ET0} = 1$.

The upper half of T0, called TH0, acts like a second 8-bit counter, which counts a fixed frequency $f_{\mathrm{OSC}}/12$ clock. At overflow, TF1 is set, and an interrupt can be generated if $\mathrm{ET1} = 1$. The logic diagram of the timer in this operating mode is presented in Fig. 6.5.

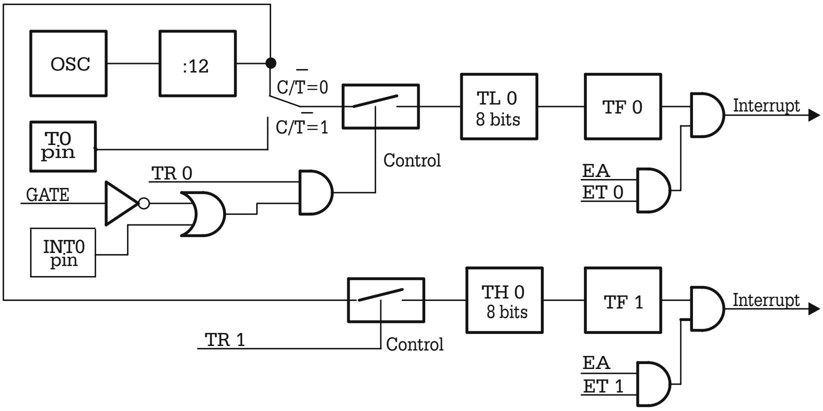


Fig. 6.5. The 8051 Timer in mode 3

6.5.6 Using Timer1 as a Baud Rate Generator

Timer1 is used to generate the communication clock for the serial port. When the serial port is configured in mode 1 or 3, the baud rate is determined by the Timer1 overflow rate, according to the following formulas (SMOD is bit 7 in PCON):

When SMOD = 0 Baud = (Timer1_Overflow_Rate)/32

When SMOD = 1 Baud = (Timer1_Overflow_Rate)/16

Both internal or external clock sources may be selected; the only thing that counts is the overflow rate. In practice, for the usual baud rates, the use of the internal clock is recommended, and to program Timer1 in mode 2, autoreload. The software initialization sequence loads TH1 with the reload value and starts the timer. No further action is required for the software.

Timer0 can be configured to operate in mode 3, and use the control signals TF1, TR1, while Timer1 is used as baud rate generator.

Compared to the AVR and HC11 timers, the general-purpose timer of the 8051 is weaker, for at least two reasons:

- The only event related to the timer reported to the CPU is timer overflow
- The timer cannot directly control any of the MCU I/O lines.

These minuses have been corrected in the next generation of microcontrollers derived from 8051. The 80x52 family of microcontrollers includes an additional timer, called Timer2, which solves these problems.

6.5.7 Exercises for Programming the 8051 Timer

SX 6.6

Starting from an oscillator frequency of 11.059 MHz, write an initialization sequence to use Timer1 as a baud rate generator for 9600 baud.

Solution

The frequency of the internal clock is: $f_{\text{COUNT}} = f_{\text{OSC}}/12 = 0.92158\,\text{MHz}$. The frequency of the UART clock is: $f_{\text{UART}} = 16\times\text{Baud\_rate} = 16\times 9600 = 153\,600\,\text{Hz}$.

The resulting overflow rate for Timer1 is:

$$\text{Timer1\_Overflow\_Rate} = f_{\text{COUNT}}/f_{\text{UART}} = 6\ .$$

This gives the reload value for Timer1:

$$(\text{TH1}) = 255 - 6 + 1 = 250 = 0\text{xFA}\ .$$

The other initializations required refer to:

- SMOD (bit 7 of PCON) must be set to 1
- Select operating mode 2 for Timer1 (M1 = 1, M0 = 0)
- Select the internal clock for Timer1 (C/T = 0)
- Start the timer (TR1 = 1)

The resulting values for the registers involved are: PCON = 80h, TMOD = 20h, and TCON = 40h.

Here is the initialization sequence required:

```
INIT_T1:
            MOV     PCON,#80H   ;SMOD=1
            MOV     TMOD,#20H   ;C/T=0, M1=1, M0=0
            MOV     TH1,#0FAH   ;auto reload value
            MOV     TCON,#40H   ;TR1=1 -- start counting
            RET
```

SX6.7

Starting from an oscillator clock frequency of 20 MHz, write the initialization sequence and the interrupt service routine to configure Timer0 in operating mode 1, to generate interrupts at 10-ms intervals.

Solution

In operating mode 1, with C/T = 0, T0 is a 16-bit counter, using the internal clock with a frequency of $f_{\mathrm{CLOCK}} = f_{\mathrm{OSC}}/12$, which corresponds to a period $T_{\mathrm{CLOCK}} = 0.6\ \mu\mathrm{s}$. The required 10-ms interval corresponds to a number of $10\,000/0.6 = 16\,666\ T_{\mathrm{CLOCK}}$ periods. To overflow in 16 666 periods, the timer must start counting at the value: $65\,535 - 16\,666 + 1 = 48\,870 =$ 0BEE6H.

The other initializations required concern selecting the operating mode 1 ($M1 = 0$, $M0 = 1$) in TMOD, and starting the timer, by setting TR1 = 1 in TCON. Finally the initialization sequence must enable a Timer0 interrupt, by setting the bits EA and ET0 in register IE. The interrupt service routine must reload the value 0BEE6H in TH0:TL0.

Here is the required initialization sequence:

```
INIT_T0:
            MOV     TMOD,#01H   ;C/T=0, M1=0, M0=1, Timer0
            MOV     TH0,#0BEH   ;TH0
            MOV     TL0,#0E6H   ;TL0
            MOV     TCON,#20H   ;TR0=1 -- start counting
            MOV     IE,#82H     ;enable interrupts
            RET
```

The interrupt service routine must reload the initialization values in TH0:TL0. Since each MOV instruction takes two cycles to execute, the reload value must be decreased by 4, and becomes 0BEE2H.

```
            MOV     TH0,#0BEH   ;TH0
            MOV     TL0,#0E2H   ;TL0
```

6.6 PWM Timers. Principles of Operation

A PWM signal is, basically, a signal with the duty cycle dynamically controlled. If this signal is passed through a low-pass filter, the output of the filter is the analog signal $V_{\mathrm{OUT}} = K \times A$, where A is the amplitude of the PWM pulses, and K is the duty cycle.

This is a simple and cheap D/A converter, and therefore most recent microcontrollers include a dedicated PWM timer, or have the main timer designed with the capability to generate PWM signals.

The Motorola 68HC11 series K microcontrollers include a dedicated PWM timer, consisting of a free-running up-counter, PWCNT, whose content is permanently compared with two programmable registers, called PWPER and PWDTY. Refer to the block diagram of this timer, presented in Fig. 6.6.

PWPER defines the period of the output signal, and PWDTY controls the duty cycle of the PWM output. When the contents of the counter PWCNT match the contents of PWDTY, the control logic changes the polarity of the output signal, and when PWCNT reaches the value in PWPER, the counter is automatically cleared.

The register PWCTL contains control bits to select the frequency of the input clock for PWCNT, the polarity of the output signal, and enable the entire PWM system.

The operation of the PWM timer is synthetically presented in Fig. 6.7.

The series K microcontrollers 68HCHC11 include four 8-bit PWM channels. These can be configured to operate as two 16-bit PWM timers. The advantage of the structure presented in Fig. 6.6 is that it allows fine-tuning of the period of the output signal in a wide range.

The AVR family of microcontrollers uses Timer1 to generate PWM signals. The PWCNT counter is implemented using the least significant 8, 9, or 10 bits of TCNT1. There is no PWPER register, so that the period of the output signal can only be adjusted by selecting the frequency of the input clock.

The length of the PWCNT counter is software selectable, by means of the bits [PWM11:PWM10] in register TCCR1A. The functions of the PWDTY register are executed by the OCR1 register. The difference is that when operating as a PWM

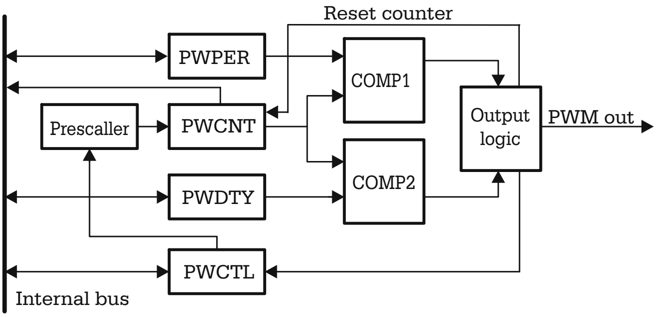


Fig. 6.6. Simplified block diagram of the PWM timer of 68HC11 series K

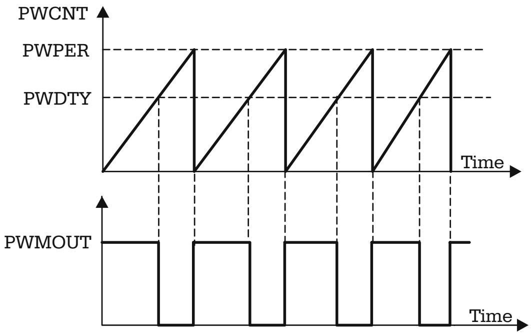


Fig. 6.7. Functional diagram of the PWM timer of 68HC11 series K

timer, TCNT1 is forced to be reversible. It counts up from $0000 to a TOP value, determined by the length of the counter (8, 9, or 10 bits). When it reaches the TOP value, it starts counting down to zero. The polarity of the output signal is changed

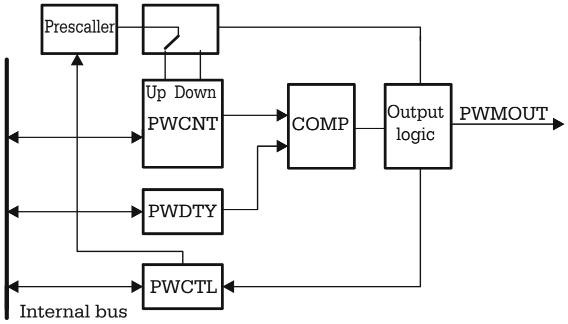


Fig. 6.8. Simplified block diagram of the PWM AVR timer

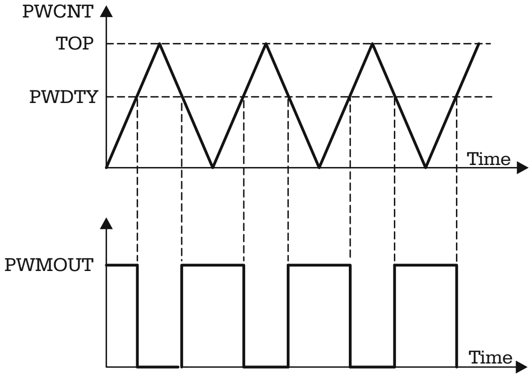


Fig. 6.9. Functional diagram of the PWM AVR timer

in opposite directions, when TCNT1 matches the value of OCR1 when counting upwards, and downwards. Refer to Fig. 6.8, and 6.9 for details of the operation of the PWM AVR timer.

This PWM system is far less flexible than that of the HC11, but it is simple and cheap, so that it has been implemented in many AVR microcontrollers.

6.7 Watchdog Timers

The block diagram of a watchdog timer is presented in Fig. 6.10. The system consists of a counter, having the overflow time programmable in a range from a few milliseconds to a few seconds. When the watchdog overflows, a hardware RESET is generated.

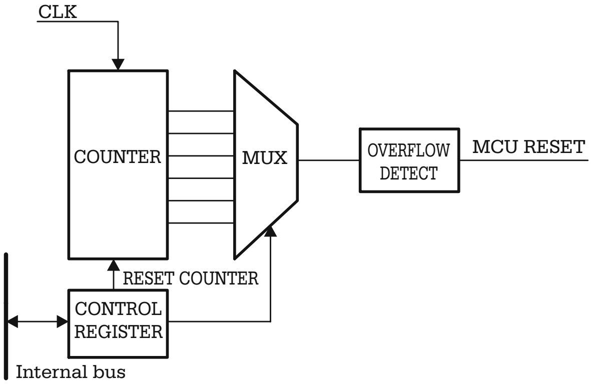


Fig. 6.10. Block diagram of the watchdog timer

If the watchdog is enabled, the program running on the microcontroller must be organized so that, periodically, at time intervals shorter than the overflow time, it resets the watchdog's counter, otherwise a hardware RESET is generated.

The control register is used to enable the watchdog, to select the overflow time, and to reset the counter.

6.7.1 The Watchdog of HC11

The watchdog system of HC11 is called the *COP Timer* (Computer Operating Properly Timer).

The watchdog overflow time is selected by means of the control bits [CR1:CR0] (COP Rate select) in the OPTION register, as shown in Table 6.9. [CR1:CR0] can only be written during the first 64 E clock cycles, after RESET. This artifice is intended to prevent unintentional modification of the watchdog settings in case of program runaway.

Table 6.9. Programming the watchdog timeout for HC11

CR[1:0]	Divide E by	Watchdog timeout for XTAL = 8.0 MHz
0 0	2^{15}	16.384 ms
0 1	2^{17}	65.536 ms
1 0	2^{19}	262.14 ms
1 1	2^{21}	1.049 s

For the same safety reason, the control bit that enables the whole watchdog system is the NOCOP bit in the CONFIG register. As mentioned before, CONFIG is a non-volatile, EEPROM type register, which can only be modified using special programming sequences, as described in Chap. 8.

A special register COPRST has been provided to clear the watchdog counter. This is cleared by writing bytes \$55 and \$AA, in this order, to COPSRT:

```
LDAA    #$55        ;clear watchdog counter
STAA    COPRST
LDAAA   #$AA
STAA    COPRST
```

6.7.2 The Watchdog of AT90S8515

The watchdog of the AVR family counts on a separate clock, distinct from the MCU clock. This is generated by an internal RC oscillator and has an approximate frequency of 1 MHz at $V_{CC} = 5$ V.

The software control on the watchdog system is performed by means of the WDCTR (Watchdog Control) register, which has the following structure:

WDCTR	7	6	5	4	3	2	1	0
	–	–	–	WDTOE	WDE	WDP2	WDP1	WDP0
RESET	0	0	0	0	0	0	0	0

[WDP2:WDP1:WDP0] – (Watchdog prescaler select) These bits determine the overflow time of the timer as shown in Table 6.10.

Before modifying [WDP2:WDP1:WDP0], it is recommended to disable or reset the watchdog, to avoid accidental activation.

The watchdog is enabled by writing 1 in the WDE (Watchdog Enable) bit. Disabling is more complicated, for safety reasons. An additional control bit WDTOE (Watchdog Turn-Off Enable) has been introduced Prior to clearing WDE, *WDTOE and WDE must be both set to 1 in the same operation*, and, in the next four cycles, WDE can be written with 0.

Table 6.10. Programming the watchdog timeout for AVR

WDP2	WDP1	WDP0	Typical time-out at $V_{CC} = 5.0$ V
0	0	0	15.0 ms
0	0	1	30.0 ms
0	1	0	60.0 ms
0	1	1	0.12 s
1	0	0	0.24 s
1	0	1	0.49 s
1	1	0	0.97 s
1	1	1	1.9 s

```
WD_DISABLE:
           LDI    R16,$1F    ;WDTOE=1,WDE=1
           LDI    R17,$17    ;WDTOE=1,WDE=0
           OUT    WDCTR,R16
           OUT    WDCTR,R17
           RET
```

To clear the watchdog counter, and reset the watchdog, the special instruction WDR (Watchdog Reset) has been provided.

7

Interfacing to Analog Signals

7.1 In This Chapter

This chapter presents several ways in which the microcontrollers interact with analog signals. It contains a description of the analog comparator of AT90S8535, the A/D converter of HC11 and AVR, and an example of interfacing 68HC11 to an external D/A converter.

7.2 The Analog Comparator

The simplest way for a microcontroller to interact with an analog signal is by using an analog comparator. It seems that all the information a comparator can offer about a signal is one bit, containing answer to the question whether the amplitude of the respective signal is or isn't higher than a certain threshold. In fact, it's not only the amplitude of the signal that counts the moments when the signal crosses the threshold are also important.

This paragraph presents the interface with the analog comparator of the AVR, an ingenious and flexible implementation. The block diagram of the interface is shown in Fig. 7.1.

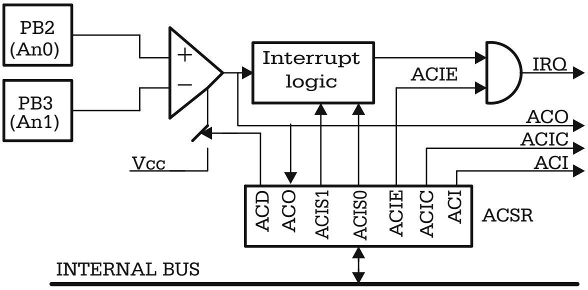


Fig. 7.1. Block diagram of the analog comparator interface of AT90S8535

The inputs of the comparator are connected to pins PB2, PB3 of the circuit. The corresponding lines of port B must be configured as inputs, with the internal pull-up resistors disabled (refer to Chap. 2 for details).

The Control and Status Register ACSR

The bits in the Control and Status Register (ACSR) of the comparator offer the following control features:

- Control of the comparator's power supply, by means of the ACD (Analog Comparator Disable) bit. Setting this bit to 1 disables the comparator.
- Read the status of the comparator's output in the ACO (Analog Comparator Output) bit.
- Select the event that triggers the interrupt mechanism by using the [ACIS1:ACIS0] control bits (Analog Comparator Interrupt Mode Select). The event selected by these bits sets the ACI flag (Analog Comparator Interrupt flag), which may generate an interrupt request. Refer to Table 7.1 for a description of how the events that set the ACI flag are related to the bits ACIS1:ACIS0.
- If the ACIC (Analog Comparator Input Capture Enable) control bit is set, the output of the comparator ACO is directly connected to the control logic that selects the input capture event for Timer1. In this case, the settings described in Chap. 6 for the selection of the capture edge apply.

The configuration of the bits in the ACSR control and status register is as follows:

ACSR	7	6	5	4	3	2	1	0
	ACD	–	ACO	ACI	ACE	ACIC	ACIS1	ACIS0
RESET	0	0	–	0	0	0	0	0

The ACI interrupt flag is erased by writing 1 in this position of the ACSR register, or by hardware, when the interrupt is executed.

Table 7.1. The effect of programming [ACIS1:ACIS0]

ACIS1	ACIS0	Interrupt mode
0	0	Interrupt on output toggle
0	1	Reserved
1	0	Interrupt on falling edge of ACO
1	1	Interrupt on rising edge of ACO

Example: Consider the following sequence of instructions:

```
LDI    R16,$88
OUT    ACSR,R16
```

The control byte written to ASCR contains the ACD and ACIE bits set to 1. The immediate effect of writing this byte to ACSR is to disable the analog comparator, but the interrupts remain active, and therefore false interrupts may be generated when power is restored to the analog comparator. *It is recommended that the analog comparator interrupts be disabled when the comparator is enabled/disabled by means of the ACD bit.*

The analog comparator interface is not implemented in the HC11 and 8051 microcontrollers.

7.3 The General Structure of the A/D Converter Subsystem

An analog signal can be represented as a continuous amplitude–time function, which, in principle, can have any shape, as in the example shown Fig. 7.2.

The only way a digital system, like a microcontroller, can acquire information about the evolution of such a signal is to measure, at discrete time intervals (T1, T2, T3, T4, ...), samples from the signal, which are converted to series of numerical values S1, S2, S3, S4, ... This measuring process is performed by the A/D converter.

Obviously, as the time between successive samples decreases, the resulted *discrete signal* better approximates the analog signal input. There is, however, a limit to the time between which two successive samples can be reduced, imposed by the *conversion time*, which is a constructive constant of the A/D converter, defined as the time required to obtain the numerical value of a sample of the analog signal.

An important parameter of the A/D converter is the *resolution,* defined as the number of bits of the result. The more bits are used for representing the number, the better is the accuracy of the representation.

Figure 7.3 shows the general block diagram of a typical A/D converter, as implemented in most microcontrollers.

Usually, a microcontroller has several analog inputs (eight, typically). The inputs are processed one by one, the selection being made by a *multiplexer* MUX, depending on the selection bits in the control register. The selected input is applied to a sample & hold (S & H) circuit, which retains the sampling value until the conversion is completed. The actual converter is of the *'successive approximation'* type, and comprises

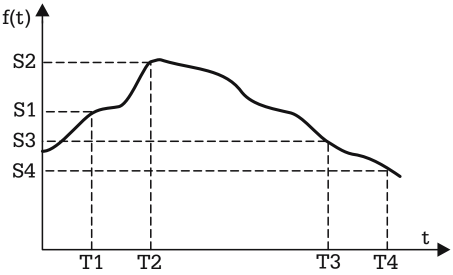


Fig. 7.2. Example of sampling an analog signal

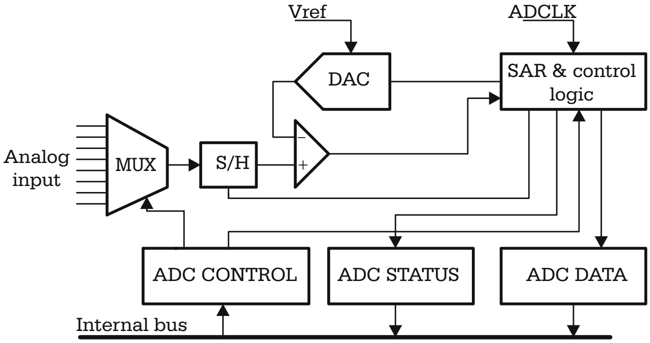


Fig. 7.3. Block diagram of a typical ADC

the SAR (Successive Approximation Register), a DAC (Digital to Analog Converter), and an analog comparator.

The conversion result is delivered in an ADC data register, along with an End of Conversion Flag in the ADC status register. Optionally, the End of Conversion Flag can generate an interrupt.

7.4 The A/D Converter of the HC11 Family of Microcontrollers

The HC11 A/D converter has the following characteristics:

- 8-bit resolution.
- 128 clock periods conversion time.
- Separate AVDD and AGND supply pins, to reduce the effect of noise.
- The entire A/D conversion system power supply is software controllable using the ADPU (Analog to Digital Converter Power Up) bit from the OPTION register.
- Separate reference voltage inputs V_{RH} and V_{RL} to define the margins of the input voltage domain. An input voltage equal to V_{RL} is converted to \$00, and an input voltage equal to V_{RH} is converted to \$FF (full scale).
- The conversion clock can be either the system clock E, or an internal clock generated by a distinct internal oscillator. The selection of the clock is made by means of the CSEL bit in the OPTION register. However, the use of the internal oscillator is recommended only when the frequency of the E clock is below 750 kHz.
- Four conversions are performed in each conversion cycle, either by converting four times the same input, or by converting four different inputs. The selection between the single-channel mode and multiple-channel mode is done using the MULT control bit from the ADCTL control register.
- There are four registers to store the conversion results named ADR1–ADR4 (A/D Result Registers). When the converter operates in single-channel mode (MULT = 0), the four ADR registers store the results of the four consecutive

conversions performed on the selected channel. When the converter operates in multiple-channel mode (MULT = 1), each of the four ADR registers contains the converted value of a different input from the selected *input group*.

- The ADC can operate either under software control, when a conversion is started by writing to the ADCTL register, or in continuous mode, when, after finishing a conversion cycle, a new cycle is automatically restarted. To select the continuous conversion mode, set the SCAN control bit to 1.
- There is no interrupt associated with the A/D converter for HC11. This event is marked only by setting the CCF (Conversion Complete Flag) status bit in the ADCTL register.

The control and status bits related to the A/D converter are distributed in OPTION and ADCTL register as follows:

The OPTION Register

SPCR	7	6	5	4	3	2	1	0
	ADPU	CSEL	IRQE	DLY	CME	FCME	CR1	CR0
RESET	0	0	0	1	0	0	0	0

- ADPU – A/D Power-Up. At RESET, ADPU is cleared, therefore it is required to set this bit to 1 by software, to enable the A/D subsystem. Note that, after power-up, the converter needs a *settle time* of about 10 ms.
- CSEL – AD clock select.

  CSEL = 0 selects the system clock E as the A/D clock.
  CSEL = 1 selects a clock supplied by an internal RC internal oscillator.

The other bits in the OPTION register refer to other subsystems or functions of the microcontroller, and will not be discussed here.

The ADCTL Register

ADCTL	7	6	5	4	3	2	1	0
	CCF	–	SCAN	MULT	CD	CC	CB	CA
RESET	1	0	x	x	x	x	x	x

- CCF – Conversion Complete Flag. This *read only* bit is automatically erased when writing to the ADCTL which is tantamount to a start conversion command, and is automatically set at the end of a four conversion cycle. When SCAN = 1, CCF is set at the end of the *first* cycle of four conversions.
- SCAN – Continuous Scan Control bit.

  SCAN = 0 single conversion cycle.
  SCAN = 1 continuous scan.

Table 7.2. The effect of programming the bits [CC:CB:CA]

[CC:CB:CA]	Channel selected	Result placed in ADRx if MULT = 1
000	AN0	ADR1
001	AN1	ADR2
010	AN2	ADR3
011	AN3	ADR4
100	AN4	ADR1
101	AN5	ADR2
110	AN6	ADR3
111	AN7	ADR4

- MULT – Multiple-channel/single-channel control.
  MULT = 0 the conversions are made on a single-input channel, selected by the [CD:CC:CB:CA] bits
  MULT = 1 the conversions are executed on a four-channel group. In this case, the CB and CA selection bits have no effect and the conversions are performed on the four input group addressed by the CD:CC
- [CD:CC:CB:CA]-channel select. The CD bit is reserved for *factory testing* – always use CD = 0. The effect of the other three bits is described in Table 7.2.

7.5 Exercises on Programming the A/D Converter of HC11

SX 7.1

Write the initialization sequence that prepares the A/D subsystem to be used with the E clock, in single-channel conversion mode, single conversion cycle.

Solution

At RESET, the APDU bit from the OPTION register is cleared. The initialization sequence must set ADPU to 1 to power up the converter. The E clock is selected by erasing the CSEL bit. These initializations must be made after RESET, at least 10 ms before the first conversion cycle. For the other initializations: erasing the MULT bit to select the single-channel operating mode, and SCAN for single conversion cycle, can be made directly in the *start conversion* routine.

The required initialization routine is this:

```
*MASTER SPI initialization routine
INIT_AD     BSET   OPTION,$80     ;power up ADC (ADPU=1)
            BCLR   OPTION,$40     ;select clock E (CSEL=0)
            CLR    ADCTL          ;MULT=0, SCAN=0
            RTS
```

SX 7.2

Write a subroutine that receives as input in A the address of the analog channel. The subroutine starts the conversion of the specified channel, waits for the end of conversion and returns the conversion result in A.

Solution

The conversion is automatically started by any write operation to the ADCTL register. CCF = 1 indicates the end of the conversion. When SCAN = 0, four consecutive conversions are performed on the same analog input. At the end of conversion, the four result registers ADR1–ADR4 contain the results of the four conversions. Here is the subroutine that performs the requested operations:

```
READ_ADC    ANDA    #$F8        ;keep only the channel address
            STAA    ADCTL       ;start conversion MULT=SCAN=0
POLL_ADC    LDAA    ADCTL
            ANDA    #$80        ;poll CCF until true
            BEQ     POLL_ADC
            LDAA    ADR1        ;get result in A
            RTS                 ;return
```

SX 7.3

Write a subroutine that starts the conversion and reads the AN4–AN7 channels, updating the variables ANALOG4–ANALOG7.

Solution

To execute a conversion cycle on a group of four inputs, the bit MULT in ADCTL must be set to 1. The inputs AN4–AN7 are selected by CC = 1 (when MULT = 1, CB and CA are ignored). The required control word for ADCTL is 00010100 = $14. To read the group AN0–AN3, the control word would be $10. The subroutine performing the requested tasks looks like this:

```
*MASTER SPI initialization routine
READ_ADC4   LDAA    #$14        ;MULT=1, CC=1
            STAA    ADCTL       ;start conversion
POLL_ADC    LDAA    ADCTL
            ANDA    #$80        ;poll CCF until true
            BEQ     POLL_ADC
            LDAA    ADR1        ;get result in A for AN4
            STAA    ANALOG4     ;update variables
            LDAA    ADR2
            STAA    ANALOG5
            LDAA    ADR3
```

```
        STAA    ANALOG6
        LDAA    ADR4
        STAA    ANALOG7
        RTS
```

7.6 The A/D Converter of the AVR Microcontrollers

The AVR ADC has some distinguishing features, compared to HC11:

- 10-bit resolution. Therefore, the conversion result is presented in two registers (ADCH:ADCL).
- The ADC can generate an interrupt request at the end of conversion.
- The ADC can operate while the CPU is in *sleep mode.* This reduces the errors caused by noise.
- A dedicated *prescaler* generates the ADC clock.
- The ADC can operate in two different modes: *single conversion* and *free running*. When in *free running* operating mode the ADC works like the HC11 ADC with SCAN = 1. In single conversion mode, only the selected channel is converted, then the ADC halts until a new conversion is started by software.

Details on the Implementation of the ADC of AT90S8535

The control and status register of the A/D Converter, ADCSR, has the following structure:

ADCSR	7	6	5	4	3	2	1	0
	ADEN	ADSC	ADFR	ADIF	ADIE	ADPS2	ADPS1	ADS0
RESET	0	0	0	0	0	0	0	0

- ADEN – AD Converter Enable. When ADEN = 1 the ADC is enabled. This bit is cleared at RESET, therefore it is required to set it to 1 in the initialization sequence.
- ADSC – A/D Start Conversion Command. Writing 1 to this bit starts the conversion. Once written with 1, the bit remains set until the end of the conversion. Writing 0 to this bit has no effect.
- ADFR – A/D Free Running Mode select. When set, this bit selects the *free running* operating mode. In this mode, the ADC permanently samples the input line and executes conversions.
- ADIF – A/D Interrupt Flag. This bit is set automatically at the end of a conversion after the data registers have been updated. If the ADC interrupts are enabled (ADIE = 1) ADIF = 1 generates an interrupt request. ADIF is automatically cleared when executing the interrupt service routine, or by writing 1 in the corresponding position of ADCSR.
- ADIE – AD Interrupt Enable. When set to 1, this bit enables the interrupts from the ADC subsystem. The interrupt is generated at the end of the conversion, by the ADIF flag.

Table 7.3. The effect of programming [ADPS2:ADPS1:ADPS0]

[ADPS2:ADPS1:ADPS0]	ADC clock is XTAL divided by:
000	2
001	2
010	4
011	8
100	16
101	32
110	64
111	128

- [ADPS2:ADPS1:ADPS0] – AD Prescaller Select. The clock applied to the ADC subsystem is obtained by dividing the system clock using a prescaler.
  These bits select the prescaler dividing rate according to Table 7.3.

The selection of the input upon which the conversion is executed is made by the ADMUX register. This has only three bits implemented, in the least significant positions.

ADMUX	7	6	5	4	3	2	1	0
	–	–	–	–	–	MUX2	MUX1	MUX0
RESET	0	0	0	0	0	0	0	0

[MUX2:MUX1:MUX0] Analog multiplexer selection bits. These bits instruct the analog multiplexer to select the input to be converted. [0:0:0] corresponds to ADC0, and so on; [1:1:1] selects ADC7.

Important note. The analog inputs of AT90S8535 use the I/O lines of PORTA. In principle, some of the A port lines can be configured as output lines. However, it is not recommended to activate these output lines while a conversion is in progress, because conversion errors can occur.

7.7 Exercises on Programming the A/D Converter AT90S8535

SX 7.4

Write the initialization sequence to select the ADC to operate using the clock frequency XTAL/2, in single conversion mode, with the interrupts enabled.

Solution

The control word to be written in ADCSR must contain the bits ADEN, ADIE and ADPS2 set to 1, the remaining bits being 0.

```
INIT_ADC:
           LDI    R16,$89
           OUT    ADCSR,R16
           RET
```

SX7.5

Write a subroutine that initializes the ADMUX register with the value of the variable CHAD (Channel Address), then starts a conversion.

Solution

```
START_ADC:
           LDS    R16,CHAD   ;read channel address
           OUT    ADMUX,R16  ;select channel
           SBI    ADCSR,7    ;start conversion
           RET

;The interrupt service routine must get the ADC data
;and store it in RAM variables
```

7.8 Digital-to-Analog Converters

7.8.1 The Principles of the D/A Conversion

In many situations, is not enough for a microcontroller to measure analog signals – it is also required to generate analog signals. There is a variety of industrial equipment that is controlled by analog signals. To communicate with this type of equipment, the microcontroller system must be able to generate analog signals with precisely controlled amplitudes.

One simple way to do this is to use a PWM timer. By applying a PWM signal to a *low-pass filter,* the resulting output signal has the amplitude $V_{\text{out}} = K \times V_{\text{M}}$ where V_{M} is the amplitude of the V_{PWM} signal, and K is the duty cycle (refer to Fig. 7.4).

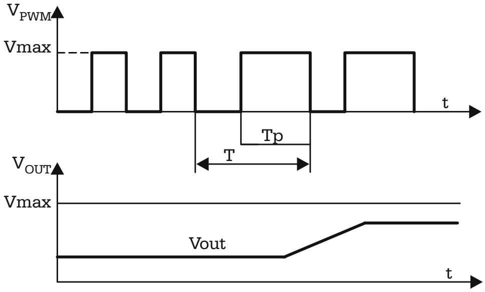


Fig. 7.4. The analog signal associated with a PWM signal

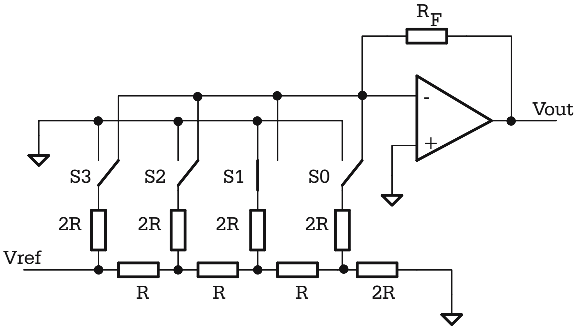


Fig. 7.5. Simplified schematic of a 4-bit DAC, using R-2R ladder network

In cases when a PWM timer is not available, or when the number of PWM channels is less than the required number of analog outputs, the solution is to use special external circuits, called Digital-to-Analog Converter, or DACs. There is a variety of such circuits with different resolutions (the number of bits of the converted word), different conversion time, or different ways to present the result (serial/parallel). The majority of them operate on the same principle, by dividing the current in a ladder type R-2R resistor network. The simplified schematic of a 4-bit DAC based on this principle is presented in Fig. 7.5.

In this circuit, the operational amplifier acts in such a way that the potential of the inverting input is maintained equal to the potential of the non-inverting input, which is connected directly to ground. As a consequence, regardless of the status of the switches S3–S0, the network acts as if all of the 2R resistors, have a terminal connected to ground. This means that the equivalent resistance that loads V_{ref} is constant, equal to R, and the total absorbed current from V_{ref} is also constant, $I_{\mathrm{ref}} = V_{\mathrm{ref}}/R$. The currents flowing through the four switches (S3, S2, S1, S0) are $\mathrm{I3} = I_{\mathrm{ref}}/2$, $\mathrm{I2} = I_{\mathrm{ref}}/4$, $\mathrm{I1} = I_{\mathrm{ref}}/8$, $\mathrm{I0} = I_{\mathrm{ref}}/16$. The output voltage is

$$V_{\mathrm{out}} = -I_{\mathrm{F}} \times R_{\mathrm{F}}$$

where I_{F} is a fraction of I_{ref}, directed by the switches S3–S0 to the inverting input of the operational amplifier,

$$I_{\mathrm{F}} = \mathrm{b3} \times \mathrm{I3} + \mathrm{b2} \times \mathrm{I2} + \mathrm{b1} \times \mathrm{I1} + \mathrm{b0} \times \mathrm{I0}$$
$$I_{\mathrm{F}} = I_{\mathrm{ref}}(\mathrm{b0}/2 + \mathrm{b1}/4 + \mathrm{b2}/8 + \mathrm{b3}/16)$$

were (b3–b0) are the bits of the word that must be converted to an analog value.

It follows that the output voltage V_{out} is proportional to the binary value to be converted.

A typical example of a circuit, built on this principle, is MX7224 from Maxim. The functional block diagram and the pin configuration of the circuit are shown in Fig. 7.6.

The circuit is designed so that it can be directly connected to a data bus. When both the CS\ and WR\ signals are active, the data from the bus is transferred to the

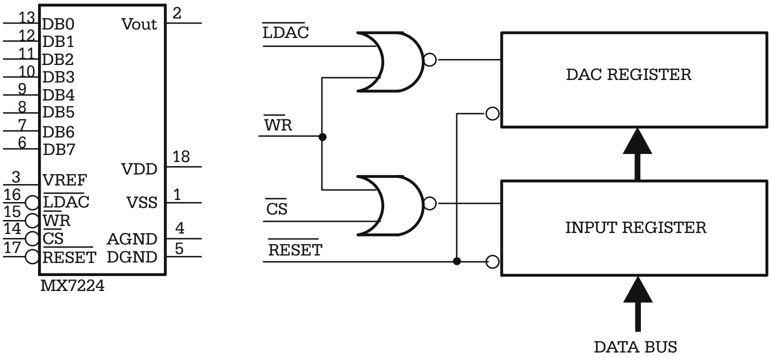


Fig. 7.6. Block diagram and pin configuration of a typical DAC IC

Input Register. The LDAC\ signal controls the transfer of data to the DAC register. When CS\ = 0, WR\ = 0 and LDAC\ = 0, both registers are transparent and the data from the DB0–DB7 inputs is converted to V_{out}. With this configuration of the control signals the circuit can be directly connected on an output port of the microcontroller

7.8.2 Exercise on Using MX7224

SX 7.6

Using the MX7224 circuit, connected to the PORTA of a 68HC11F1 microcontroller, write a program that generates a signal with the characteristics shown in Fig. 7.7. The reference voltage used is 2.5 V, and the frequency of the internal E clock is 2 MHz. The output signal is periodic with 10-ms period. Figure 7.7 gives the values of the samples for one period.

Solution

The values of the 10 samples, corresponding to a period of the output signal, must be written in the port connected to the DAC at 1-ms intervals. To generate the time

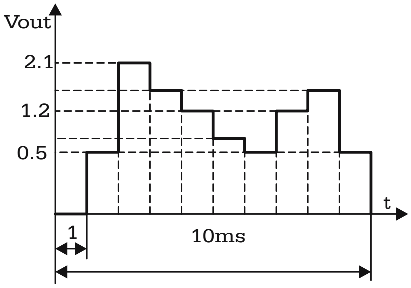


Fig. 7.7. Waveform of the signal referred in the exercise from paragraph 7.8.2

intervals a timer is required, TOC1 for example, which generates periodic interrupts. The timer interrupt service routine must update the port data. The initiation sequence must configure the PORTA in output mode, enable the TOC1 interrupt, and initialize the pointer XTAB with the address of the ROM table that contains the values of the samples.

```
INIT_DAC    LDD    TOC1
            ADDD   #2000       ;next interrupt in 1 ms
            STD    TOC1
            LDX    #DACTAB
            STX    XTAB        ;init pointer in DACTAB
            BSET   TFLG1,$80   ;clear OC1F if any
            BSET   TMSK1,$80   ;enable TOC1 interrupts
            LDAA   #$FF
            STAA   DDRA        ;PORTA all lines output
            RTS                ;end of initialization
```

The interrupt service routine must do the following:

- Erase the interrupt flag.
- Prepare the next interrupt from TOC1 to come in 1 ms.
- Write the value from DACTAB in PORTA
- Increment the XTAB pointer, and check if the end of table has been reached. If the end of the table is detected, XTAB must be reinitialized with the starting address.

```
TOC1_ISR    BSET   TFLG1,$80   ;clear interrupt flag
            LDD    TOC1
            ADDD   #2000       ;next interrupt in 1 ms
            STD    TOC1
            LDX    XTAB
            LDAA   0,X         ;get data from table
            STAA   PORTA       ;write it to DAC
            INX                ;increment pointer
            CPX    #ENDTAB     ;check for end of table
            BHS    TOC1X
            STX    XTAB        ;update pointer and exit
            RTI
TCO1X       LDX    #DACTAB     ;reload pointer if end of
                               ;table detected
            STX    XTAB
            RTI
```

The table with the DAC values for the specified curve is obtained by linear interpolation, considering that the value $V_{\text{out}} = V_{\text{ref}} = 2.5$ V corresponds to a binary input \$FF.

```
DACTAB     DB     $00
           DB     $33
           DB     $D6
           DB     $A3
           DB     $7A
           DB     $4C
           DB     $33
           DB     $7A
           DB     $A3
           DB     $4C
ENDTAB     EQU    *           ;end of table
```

Obviously, by modifying the data in DACTAB, it is possible to obtain any waveform for the output signal. The only limitation is imposed by the speed of the processor. For HC11, the 1-ms interval between two successive interrupts is close to the upper limit. The AVR microcontrollers are much faster.

8

Using the Internal EEPROM Memory

8.1 In this Chapter

This chapter contains a description of the EEPROM memory as implemented in the HC11 and AVR families of microcontrollers, as well as a description of the EEPROM control registers, and software examples for erasing and programming the EEPROM.

8.2 Overwiew of the EEPROM Subsystem

In many situations it is required that some program parameters, calibration tables, etc. are stored in nonvolatile memory, able to keep its contents indefinitely after the system is powered off.

The solution to this problem is to include in the structure of the microcontroller an EEPROM memory area (Electrically Erasable Programmable Read Only Memory). Most modern microcontrollers include between 128 bytes and 2 kilobytes of EEPROM.

For technological reasons, erasing and programming the EEPROM requires a 20 V V_{pp} voltage, obtained by means of a so-called *charge pump*. The current capability of this internal source is very low, and therefore the charge pump requires a time of around 10 milliseconds to stabilize. The following restrictions apply when accessing the EEPROM:

- Before programming, an EEPROM bit must be erased. The value of an erased bit is 1.
- After an erase or program operation, a 10-ms delay is required for the charge pump to stabilize.

8.3 The EEPROM Memory and the CONFIG Register of HC11

Depending on the model, the microcontrollers of the HC11 family have between 512 and 2048 EEPROM memory bytes, *mapped in the general memory map*. The starting address of the EEPROM memory area differs from one model to another, but, in some cases, the whole EEPROM block can be remapped, by means of the control bits [EE3:EE0] from the CONFIG register.

8.3.1 The Registers Controlling the EEPROM of HC11

HC11 has two registers involved in the control of the EEPROM. These are the PPROG register (EEPROM Programming Control Register) and the BPROT register (EEPROM Block Protect Register). The PPROG register has the following structure:

PPROG	7	6	5	4	3	2	1	0
	ODD	EVEN	–	BYTE	ROW	ERASE	EELAT	EEPGM
RESET	0	0	0	0	0	0	0	0

The ODD and EVEN bits are used in the *special test* operating mode and they will not be discussed here.

The BYTE, ROW and ERASE bits control the erase process of the EEPROM, according to Table 8.1.

Table 8.1. The effect of programming the bits BYTE, ROW, and ERASE

BYTE	ROW	ERASE	Erase mode
x	x	0	Normal read/program
0	0	1	All locations erased
0	1	1	A 16 bytes “row” is erased
1	0	1	One byte is erased
1	1	1	One byte is erased

- EELAT – EEPROM Latch Control bit

  EELAT = 0 EEPROM address and data buses are prepared for normal read operations.
  EELAT = 1 EEPROM address and data buses are prepared for write or erase operations. Write operations to EEPROM with EELAT = 1 cause the values of address and data to be retained in special latches.

- EEPGM – EEPROM Program Control bit.

  EEPGM = 1 starts the charge pump assuring the programming tension for the EEPROM. The EEPGM bit can be written only if EELAT = 1.

The BPROT register has the following structure:

BPROT	7	6	5	4	3	2	1	0
	–	–	–	PTCON	BPRT3	BPRT2	BPRT1	BPRT0
RESET	0	0	0	1	1	1	1	1

Table 8.2. Block addresses associated with BPRTi, valid for 68HC11F1

Bit name	Block protected
BPRT0	$xE00–$xE1F
BPRT1	$xE20–$xE5F
BPRT2	$xE60–$xEDF
BPRT3	$xEE0–$xFFF

- PTCON – Protect CONFIG register. When this bit is set to 1, the CONFIG register cannot be written or erased.
- BPRT3–BPRT0 – When these bits are 1, the protection of an EEPROM memory block associated with each bit is activated, as shown in Table 8.2.

The BPROT register can only be written during the first 64 E cycles after RESET. Out of RESET, all BPROT bits are set to 1, which means that the protection is activated. The programming examples presented in the next paragraph assume that all bits of interest in the BPROT register have been erased in the initialization sequence, executed immediately after RESET.

8.3.2 Software Routines to Erase and Write the EEPROM

8.3.2.1 Erasing a Single Byte of EEPROM

The E2BE (EEPROM Byte Erase) subroutine receives in X the address of the byte to be erased. The first step is to write in PPROG a control word, which specifies an erase operation (EELAT = 1 and ERASE = 1) at the byte level (BYTE = 1). The next step is to perform a write operation to the address of the byte to be erased.

When EELAT = 1, the address of the destination of any write operation is stored in special latches, and will be used in further erase or write operations to EEPROM. The actual erase process starts when the EEPGM bit is set, and lasts about 10 milliseconds. After this delay, the PPROG register must be cleared to return to normal operation mode. Here is the subroutine that executes an EEPROM byte erase.

```
E2BE       LDAB   #$16       ;BYTE=1, ERASE=1, EELAT=1
           STAB   PPROG
           STAB   0,X        ;write operation to latch
                             ;address
           LDAB   #$17       ;make EEPGM=1
           STAB   PPROG      ;start Vpp charge pump
           JSR    DLY10      ;wait 10 ms
           CLR    PPROG      ;stop Vpp and return to
                             read ;mode
           RTS               ;return to main program
```

8.3.2.2 Writing a Byte to EEPROM

The E2W (EEPROM write) subroutine, listed below, assumes that:

```
E2W        LDAB   #$02       ;BYTE=0, ERASE=0, EELAT=1
           STAB   PPROG
           STAA   0,X        ;write operation to latch the
                             ;address and data
           LDAB   #$03       ;make EEPGM=1
           STAB   PPROG      ;start Vpp charge pump
           JSR    DLY10      ;wait 10 ms
           CLR    PPROG      ;stop Vpp and return to read
                             ;mode
           RTS               ;return to main program
```

- The bit in BPROT associated with the destination address of the write operation is cleared, i.e. the destination is not write protected.
- The destination byte has been previously erased.
- The address of the destination byte is placed in X.
- The data byte to be written in the EEPROM is placed in A.

If ERASE = 1 and BYTE = 0, setting PPROG will initiate an erase sequence on the entire EEPROM memory, called BULK ERASE.

Reading EEPROM while a write or erase operation is in progress (EELAT = 1) will return erroneous data. There is no hardware mechanism to prevent this type of error.

8.3.3 The CONFIG Register

The CONFIG register consists of eight EEPROM cells, organized as a register located in the I/O register block of the MCU. The CONFIG register can be erased or programmed just like any other EEPROM location. The structure of the CONFIG register of 68HC11F1 is as follows:

CONFIG	7	6	5	4	3	2	1	0
	–	–	–	–	–	NOCOP	ROMON	EEON
RESET	0	0	0	0	0	x	x	x

- NOCOP = 1 disables the COP (Computer Operating Properly) watchdog,
- ROMON = 1 enables the internal ROM.
- EEON = 1 enables the internal EEPROM.

Some members of the HC11 family allow remapping of the EEPROM block to the beginning of any 4 K boundary in the memory map. To this purpose, the most significant four bits of the CONFIG registers, called [EE3:EE2:EE1:EE] are used to define the most significant four bits of the address of the EEPROM. For example, if [EE3:EE2:EE1:EE] = [0:1:0:1], then the starting address of the EEPROM is $5000.

> Bulk erase operations on the EEPROM do not affect the CONFIG register.
>
> A new value written to the CONFIG register becomes effective only after a subsequent RESET sequence.

8.4 The EEPROM Memory of the AVR Microcontrollers

There are significant differences in the way the EEPROM memory is implemented in AVR microcontrollers, compared to HC11. While in HC11, the EEPROM memory is directly visible in the memory map, and can be used as data memory or as program memory, for the AVRs the access to the EEPROM looks more like accessing data from a peripheral interface.

8.4.1 The Registers of the Interface with the EEPROM Memory

Four registers control the access to the EEPROM. These are named EEARH, EEARL, EEDR and EECR.

- EEARH – EEARL (EEPROM Address Register High/Low) form together a 16-bit register that implements the EEPROM address space.
- EEDR – EEPROM Data Register. This is used to access the EEPROM data, during the read and write operations.
- EECR – EEPROM Control Register contains the control bits for the write and read operations. EECR has the following structure:

EECR	7	6	5	4	3	2	1	0
	–	–	–	–	–	EEMWE	EEWE	EERE
RESET	0	0	0	0	0	0	0	0

- EEMWE – EEPROM Master Write Enable. This bit provides a protection mechanism of the EEPROM data, in case of program runaway. The EEMWE is set by software, but is automatically cleared by hardware, after four cycles of the main system clock. In this interval of four cycles, it is possible to initiate a write operation to the EEPROM, by writing 1 to EEWE. Attempts to write to EEWE when EEMWE = 0 have no effect. EEWE is cleared by hardware when the write operation completes, usually after 2.5–4 ms, depending on the value of the supply voltage. Therefore, it is recommended to poll EEWE, to determine the end of the write operation.
- EERE – EEPROM Read Enable. This bit selects the access type to the EEPROM.

EERE = 1 indicates a read operation, while EERE = 0 indicates a write operation.

8.4.2 Software Routines to Read and Write EEPROM

8.4.2.1 Reading a Byte from EEPROM

The following program sequence reads in R16 an EEPROM byte from the address indicated by register Z. It starts by testing EEWE to determine if the interface is ready. If EEWE = 0 the contents of Z are transferred to EEARH–EEARL, then a control word is written to EECR having EERE = 1 (read operation). The EEPROM data is now readable in EEDR.

```
EERead:
           SBIC   EECR,EEWE  ;make sure EEWE=0
           RJMP   EERead
           OUT    EEARH,ZH   ;write address in EEARH-EEARL
           OUT    EEARL,ZL
           LDI    R16,$01    ;EERE=1 -- read operation
           OUT    EECR, R16
           IN     R16,EEDR   ;read data in R16
           RET
```

8.4.2.2 Writing a Byte To EEPROM

The subroutine described below writes the byte in R16 to the EEPROM address specified by Z. The following sequence of operations is performed:

- Check EEWE to determine if the interface is ready.
- Write Z to EEARH:EEARL.
- Write R16 to EEDR.
- Set EEWME.
- Set EEWE to start the actual write operation.

Here is the program that executes these operations:

```
EEWrite:
           SBIC   EECR,EEWE  ;make sure EEWE=0
           RJMP   EEWrite
           OUT    EEARH,ZH   ;write address in EEARH-EEARL
           OUT    EEARL,ZL
           OUT    EEDR,R16   ;write data to EEDR
           CLI               ;disable interrupts
           LDI    R16,$04    ;EEMWE=1
           OUT    EECR, R16
           LDI    R16,$02    ;EEWE=1
           OUT    EECR,R16
           SEI               ;enable interrupts
           RET
```

Note that before launching the actual write command, the program disables all interrupts. The reason for this is that an interrupt occurring between the execution of steps 4 and 5 would delay the moment when EEWE is set beyond the limit of four cycles after EEMWE is set. Writing EEWE in step 5 would find EEWME cleared by hardware, and the whole write operation would fail.

9

HC11 Development Board

9.1 In this Chapter

This chapter contains the description of a simple development board based on the microcontroller 68HC11F1, and related software utilities, intended to allow the user to write and test software applications for this microcontroller.

9.2 Description of the Hardware Module

A *development system* for the study of a microcontroller comprises a hardware module and a set of software utilities (cross-assembler, compiler, etc). Along with the data sheets and application notes from the manufacturer, this allows the user to reach a clear understanding of the resources and capabilities of that specific microcontroller, and to write and test software applications for it.

The module presented here is one of the simplest possible. It is built around the Motorola 68HC11F1 microcontroller, operating with an external EEPROM memory AT28C256, easily reprogrammable around 10 000 times. Test programs can be written directly to AT28C256 using a standard commercial EPROM programmer.

The 68HC11F1 microcontroller does not have internal program memory – it is designed to work with external ROM in *expanded mode*. For this purpose, the microcontroller can provide up to four chip select signals for external memory or other I/O devices. The way these signals activate, their polarity and relative priority are software controlled. The schematic of the hardware module is presented in Fig. 9.1.

There are very few external components, besides the microcontroller. The program memory is connected directly to the address and data buses, and the chip select signal CSPROG is provided by the microcontroller, without the need of an external address decoder.

The external clock circuit is implemented using the crystal Q1 (8 MHz typical), the capacitors C5 and C6 (15–22 pF) and the resistor R1 (1–10 M). This type of oscillator circuit is typical of all HC11 family members.

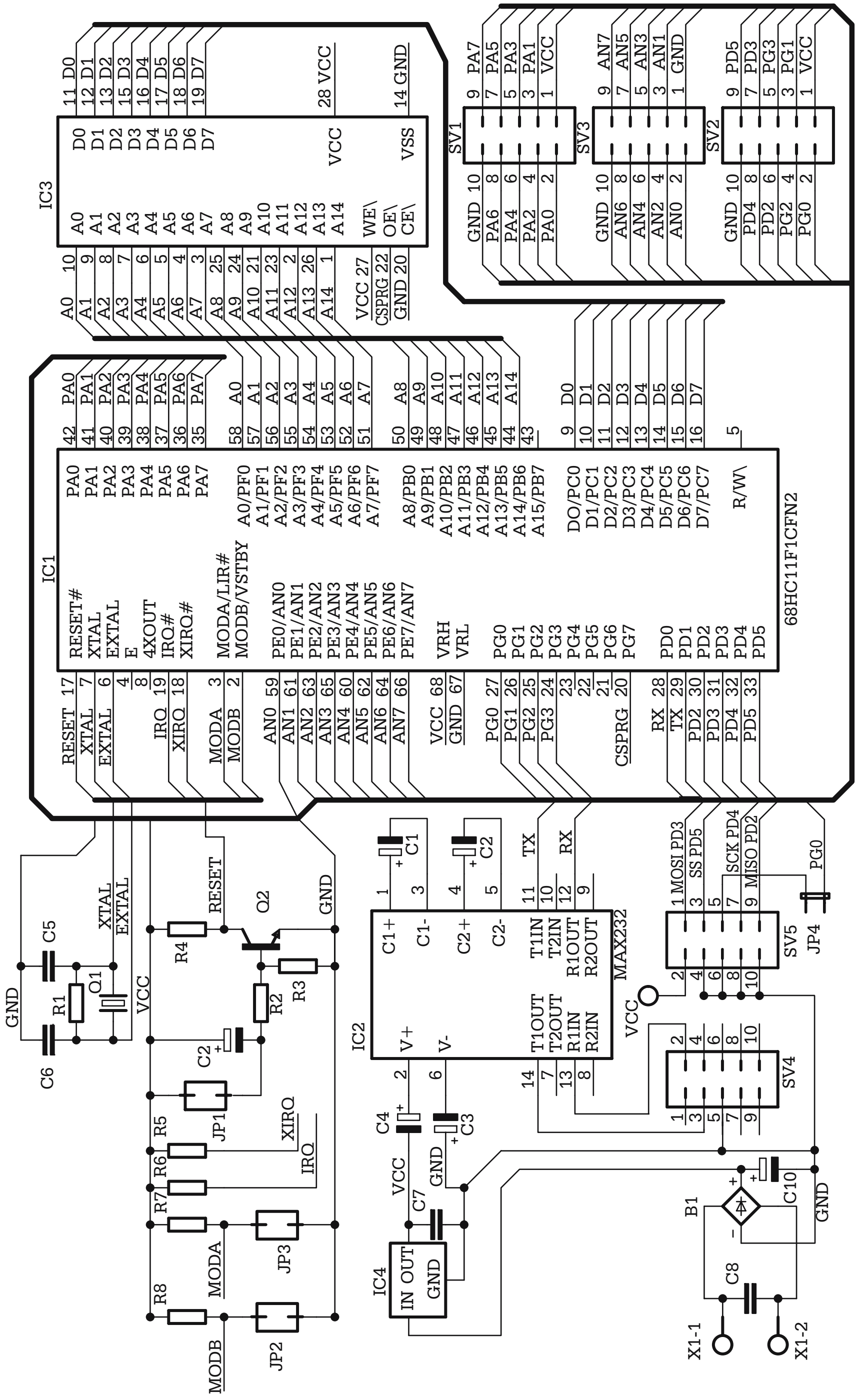


Fig. 9.1. Schematics of the HC11 development board

The RESET circuit comprises the transistor T1 along with resistors R2, R3, R4, capacitor C9 and connector JP1. At power-up, the capacitor C8 starts charging through R2, feeding current to the base of T1. This saturates the transistor for a time interval comparable to R2×C9, generating an active LOW pulse on the RESET line of the microcontroller. A similar process occurs when the terminals of JP1 are short-circuited, providing a means for manual reset of the module.

The IC2 circuit (MAX232) along with capacitors C1, C2, C3, and C4 implement the typical RS232 interface presented in Chap. 3.

The control signals MODA, MODB and the external interrupts IRQ and XIRQ are pulled up to V_{cc} using four 10 K resistors. The rest of the I/O lines of the microcontroller are brought to three header-type connectors SV1, SV2, SV3.

The module is powered from a DC or ac adapter, able to deliver 9–12 V at 250 mA. The input voltage is applied through the bridge rectifier B1 to the voltage regulator IC4 (7805) that generates the +5 V supply voltage.

9.3 Assembling and Testing the Module

All the projects described in this section come with detailed schematics and printed circuit board drawings available on the accompanying CD. The PCB drawings are created with Eagle 4.11 from CadSoft. A *freeware* version of this software is available for free download from Cadsoft, so that the user can view, edit, and print all the schematics and the PCB drawings of the projects described in this book.

The major advantage of this CAD software is that it is very easy to learn, and the freeware version, despite some limitations, still allows the user to design pretty complex microcontroller based circuits.

Most companies manufacturing custom printed circuit boards accept drawings created with Eagle 4.11, and the cost per square decimeter of printed circuit board is reasonable. However, in this particular case, the circuit is so simple that it can be easily realized on a prototyping board using the wire wrap technique. It is recommended to use appropriate IC sockets for all ICs, except IC4.

Refer to Fig. 9.2 for a possible component layout of the PCB.

Once you have the module assembled, follow this test sequence:

1. Apply power to the module, without having the ICs inserted in the sockets.
2. Check the +5 V voltage on every socket.
3. Check the RESET circuit by measuring the voltage in the collector of T1. When a short-circuit is applied to the terminals of JP1, the voltage on this point drops near 0 V, while without short-circuit it stays near the V_{cc} value.
4. Insert the ICs into the sockets and apply power to the module. With an oscilloscope check the E clock on pin 4 of the microcontroller. If you don't have an oscilloscope, use a voltmeter to measure the RESET signal on the collector of T1. If the oscillator fails to operate, the clock monitor circuit of the microcontroller will force this line to logic zero.

5. If everything is alright, install the cross-assembler, as described in the next paragraph and assemble a simple software example, like the one presented in this chapter in exercise X9.1.
6. Using a commercial EPROM programmer, program the AT28C256 memory with the binary file created by the cross-assembler, and check the expected behavior of the module.

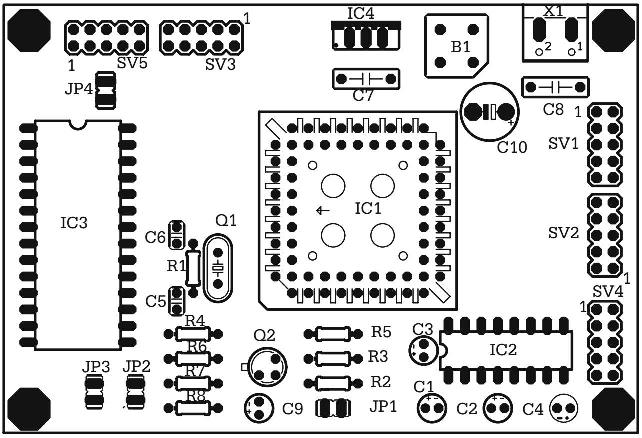


Fig. 9.2. Component layout of the PCB for the HC11 development board

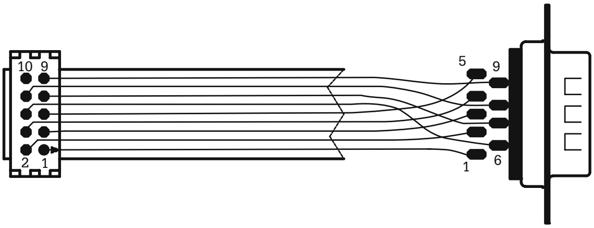


Fig. 9.3. Layout of the adapter cable for RS232

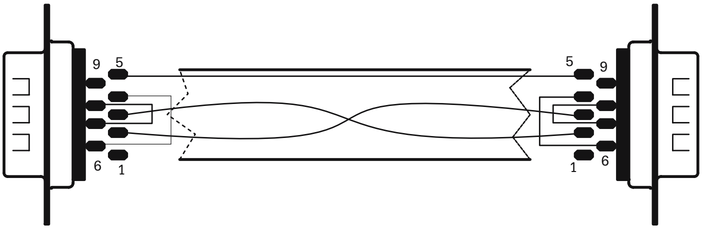


Fig. 9.4. Layout of the NULL MODEM cable

For ALL the projects described in this book, the RS232 interface is connected to a header-type connector. The adapter cable between the header connector and the NULL MODEM cable is presented in Fig. 9.3, and the NULL MODEM CABLE is shown in Fig. 9.4.

9.4 Description of the Software Components

The freeware cross-assembler offered by Motorola for the HC11 family, called AS11.EXE, is extremely simple and has many weak points. Among them, the most important are:

- It does not allow the INCLUDE directive, thus forcing the user to write all the source code in a single file. This can be extremely inconvenient. For example, the source code for the free monitor offered by Motorola (BUFFALO) has over 5000 lines of code. Obviously, it is virtually impossible to debug such a program, if anything goes wrong. On the other hand, if the assembler accepts INCLUDE, the source code can be organized in smaller, *reusable*, blocks of code, placed in distinct files, according to some user selected criteria, e.g. TIMER.ASM, SCI.ASM, OUTPUT.ASM, etc. All these software modules are invoked by a main module, using the INCLUDE directive.
- It does not allow macro definitions, thus severely limiting the freedom of the user to customize his/her programming environment.

For these reasons, we chose to use a different freeware cross-assembler, called ASHC11, created by *Peter Gargano*. Besides INCLUDE and macro definitions, this cross-assembler has many additional features. See Appendix B2 for details on how to download this free software.

Getting Started

Step 1. Installing the Cross-Assembler

ASHC11 is a DOS application and does not require any special installation procedure. To install it, create a folder on the hard drive with the name C:\ASHC11, and unzip the downloaded archive to this folder. Then create a DOS window, having the working directory in C:\ASHC11.

To do this, in Windows™ 9.x, press the right button of the mouse on the desktop, and, from the list of options presented, choose *New*, then *Shortcut*. In the space reserved to define the command line associated to the new shortcut, type "*Command.com*", then press the *Next* button, and *Finish*.

The effect of this operation is that a new shortcut appears on the desktop, named "MS-DOS™ Prompt". To complete the job, specify the working directory for this shortcut. To do this, select the new shortcut, then press the right button of the mouse and choose the option *Properties*, and then *Program*.

In the field labeled *Working*, type C:\ASHC11, and then press *OK*. Double clicking on the new shortcut opens a DOS window, where applications can be launched from the command line by simply typing their names, followed by [ENTER].

To simplify the use of the assembler, we created a batch file, called LASM.BAT, with the following contents:

```
ashc11 %1.asm -LIST=%1.lst
bincvt %1.s19 -OV
```

The first line invokes the assembler, and instructs it to create a list file, having the same name as the source file, and the extension .LST. The default extension for the source file is assumed to be .ASM.

The second line calls a conversion utility BINCVT, provided by the author of ASHC11 in the same archive with the assembler. BINCVT creates a binary file having the same name as the source, and the extension .BIN. This can be useful, since some EPROM programmers do not properly handle the S19 output files generated by the assembler.

The batch file must be invoked by specifying the name of the source file, without extension, like in the example below:

```
LASM STEP1
```

This causes the file STEP1.ASM to be open and assembled. The following files are created in this process:

- STEP1.S19 – this contains the actual executable code in Motorola S19 format.
- STEP1.BIN – contains the executable code in plain binary format.
- STEP1.LST – this is the list file.

The list file is a text file, containing detailed information on the source file, as well as on the result of the assembly process – the output code, and the address where this will be placed in the memory map. The list file also explains the variable definitions, and gives a short description of the errors that might occur during assembly.

ASHC11, like all MS-DOS™ based programs, admits only short file names in the format “8.3”, i.e. maximum eight characters for the file name, and maximum three characters for the extension. If you are using a Windows™ based text editor to create your source files, like *Notepad*, make sure you observe this rule when you save your files on disk.

If you are not familiar with MS-DOS™ programs, consult Appendix A15 for the description of a software utility that allow invoking the assembler from a Windows™ application. This utility also contains a text editor for creating and editing the assembler source files.

Step 2. Defining the Memory Sections for the Program

Consider the following simple source file: (STEP1.ASM)

```
TITLE Defining origins for DATA and CODE sections
           DATA
           ORG    $100       ;DATA starts at $100
VAR1       DS     1          ;reserve 1 byte for
                             a variable
           CODE
           ORG    $8000      ;CODE starts at $8000
RESET      EQU    *          ;Assign the value $8000 to
                             ;the label RESET
           LDAA   VAR1       ;some code here
           BRA    RESET
*Define the RESET vector here
           ORG    $FFFE
           DW     RESET      ;place $8000 at the
                             address of ;vector
           END
```

Though very simple, this code fragment contains some fundamental elements, to be considered in any program, written for any microcontroller. It shows how to define the starting address of the two main memory segments: the CODE segment (or section), where the executable code is placed, and the DATA segment, normally a RAM zone, reserved for variables. This is done with the ORG (Origin) directive.

The assembler associates a pointer to each predefined memory section (DATA, CODE). The ORG directive initializes the pointer of the current section to the value specified as a parameter to ORG.

Symbolic parameters for ORG are allowed, like in this example:

```
RESET      EQU    $8000
           CODE
           ORG    RESET
```

The assembler automatically increments the pointer associated with the section, after each operation that reserves space in that section. In the example below, three variables are defined:

```
           DATA
           ORG    $100
VAR1       DS     1
BUF1       DS     10
VAR2       DS     1
```

Here is the list file generated by ASHC11 for this fragment:

```
            =0000   1   DATA
0100        =0100   2   ORG     $100
0100                3   VAR1    DS      1
0101                4   BUF     DS      10
010b                6   VAR2    DS      1
```

For the variable VAR1, one byte is reserved, at the current address in the DATA section. The variable BUF1 takes the next 10 bytes between the addresses $101 and $10A, and VAR2 will take the memory location at the address $010B.

A similar process occurs for the CODE section. The assembler automatically determines how many bytes of code each instruction need, and increments the pointer accordingly.

Step 3. Creating Symbolic Names for Resources

The EQU (EQUation) directive associates a numeric value to a symbolic name like in this example:

```
REGBASE     EQU     $1000
SCDR        EQU     REGBASE+$002F
```

In this mode, all the resources of the microcontroller can have symbolic names assigned. These definitions can be saved in distinct files that can be invoked later, using the INCLUDE directive. The accompanying CD includes several definition files for the HC11 microcontrollers. They are named 68HC11E9.DEF, 68HC11F1.DEF, 68HC11KA.DEF.

STEP 4. USING MACROS

A MACRO is a block of code delimited by the directives MACRO and ENDM, associated with the label NAME:

```
NAME        MACRO   <param>, <param>
            ....
            ENDM
```

When the assembler encounters a previously defined MACRO name in the source file, it automatically inserts the whole block of code associated with that MACRO name. The examples below are intended to illustrate the utility of this feature:

```
PSHD    MACRO
        PSHA
        PSHB
        ENDM
```

```
PULD    MACRO
        PULB
        PULA
        ENDM
```

The next MACROs create the possibility of conditional program jumps anywhere in the 64 K addressable space.

```
JEQ     MACRO     ?dest
        LOCAL     @A
          BNE     @A
        JMP       ?dest
@A      EQU       *
        ENDM
```

```
JNE     MACRO     ?dest
        LOCAL     @A
        BEQ       @A
        JMP       ?dest
@A      EQU       *
        ENDM
```

Please note the following distinctive features of these MACROs:

1. MACROs allow the use of *parameters*, in this case the destination address of the jump.
2. *Local labels* are allowed. This avoids "duplicate symbol" errors, when the macro is used more than once in a software module. Note that not all assemblers allow local labels in MACROs.
3. The line
   @A EQU *
   indicates that the symbolic label @A receives the current value of the pointer of the CODE section.

An interesting application of this feature is shown in the following example:

```
VECTOR_SCI    MACRO
              LOCAL     @SCI_SVC
              CODE
@SCI_SVC      SET       *
              ORG       $FFD6
              DW        @SCI_SVC
              ORG       @SCI_SVC
              ENDM
```

This MACRO saves the current address in the CODE section in the variable @SCI_SVC, then stores this value to the addresses $FFD6-$FFD7, which contain the interrupt vector for the SCI system. After this operation the pointer of the section is restored to the value saved in the variable @SCI_SVC. Similar MACROs define the other interrupt vectors. See the file AS11.MAC on the accompanying CD, for the whole set of MACROs used in this book.

Step 5. Defining a General Structure for Software Applications

Consider the example found in the file *STEP5.ASM:*

```
           TITLE    MAIN MODULE
           INCLUDE  68HC11F1.DEF   ;definitions
           INCLUDE  AS11.MAC       ;macro definitions
           INCLUDE  MAP.ASM        ;memory map

           CODE
           VECTOR_RESET
RESET      EQU      *
           INCLUDE  INIT.ASM       ;initialization
MLOOP      EQU      *              ;start of main loop
           INCLUDE  TIMER.ASM      ;timer routines
           INCLUDE  SCI.ASM
* ............................
* other user modules come here
           JMP      MLOOP          ;the program is an
                                   ;infinite loop
           INCLUDE  GENLIB.ASM
                                   ;general purpose
                                   ;routines
           INCLUDE  TABLES.ASM     ;ROM tables
           END
```

The first line instructs the assembler to include in the source file the file 68HC11F1.DEF, which contains symbolic definitions for the resources of the microcontroller.

The second line invokes the library of MACRO definitions AS11.MAC.

The file MAP.ASM, invoked in the third line, contains the origins of the DATA and CODE sections. It is advisable to define here all the variables used by the program. Here is an example of structure for MAP.ASM.

```
           TITLE  Memory map & global variables

           DATA
           ORG    RAMBASE     ;RAMBASE is defined in
                              ;68HC11F1.DEF
VAR1       DS     1           ;some variables
VAR2       DS     1
VAR3       DS     1

           CODE
           ORG    ROMBASE     ;ROMBASE is defined in
                              ;68HC11F1.DEF
           END
```

Note that no actual executable code has been generated so far. The purpose of all these include files is to simplify and to organize the dialog with the assembler.

The MACRO named VECTOR_RESET defines the entry point in the main program after RESET. The first piece of code executed after RESET is the initialization sequence for registers associated with the hardware subsystems and for RAM variables.

The example below presents some typical initializations: the stack pointer, data direction register for an I/O port, and some variables:

```
             TITLE  INITIALIZATION SEQUENCE

             LDS    #RAMEND     ;Init SP
             LDAA   #$05        ;Enable CSPROG
             STAA   CSCTL       ;for a 32k memory
             LDAA   #$3C
             STAA   DDRD        ;PORTD 2-5 for output
             CLR    VAR1        ;other initializations
   *.......
             END
```

An important detail on this initialization sequence concerns the ability of 68HC11F1 to generate four selection signals, CSPROG, CSGEN, CSIO1, CSIO2, for external memory or I/O devices

These signals are software controlled by means of four special registers: CSCTL, CSGADR, CSGSIZ and CSSTRH.

The CSPROG line that selects the external program memory is controlled by the CSCTL register. The structure of CSCTL register is as follows:

CSCTL	7	6	5	4	3	2	1	0
	IO1EN	IO1PL	IO2EN	IO2PL	GCSPR	PCSEN	PSIZA	PSIZB
RESET	0	0	0	0	0	0	0	0

- IO1EN – Enable CSIO1.
  When set to 1, CSIO1 is enabled. When cleared to 0, CSIO1 is disabled and the associated pin, PORTG bit 5, is usable as a general-purpose I/O pin.
- IO1PL – Controls the polarity of CSIO1. When this bit is 0, CSIO1 is active LOW; when set to 1, CSIO1 is active HIGH.
  IO2EN and IO2PL act similarly on CSIO2.
- GCSPR – General-Purpose Chip Select Priority

  GCSPR =0 CSPROG has priority compared to CSGEN
  GCSPR = 1 CSGEN has priority compared to CSPROG

- PCSEN – Program Chip Select Enable

  When the microcontroller operates in expanded mode (i.e. with external bus) this bit is automatically set out of RESET.

PSIZA and PSIZB. These bits relate the selection signal with the size of the external memory, as shown in Table 9.1.

This explains why, in the initialization sequence, CSCTL is written with $05. This means that an external 32 K ROM is used, while the other three chip selects are disabled and the pins of PORTG are usable as general-purpose I/O pins.

Table 9.1. The effect of programming the bits [PSIZA:PSIZB] in CSCTL

PSIZA	PSIZB	Size (Kbytes)	Address range
0	0	64	\$0000–\$FFFF
0	1	32	\$8000–\$FFFF
1	0	16	\$C000–\$FFFF
1	1	8	\$E000–\$FFFF

After the initialization sequence, the program enters an infinite loop. In the example presented (STEP5.ASM) the program loop includes the modules TIMER.ASM and SCI.ASM, while GENLIB.ASM and TABLES.ASM remain outside the loop. See the examples in the next paragraph for a better understanding on how to organize your software.

Step 6. Creating Reusable Software Modules

This paragraph presents a detailed solution to create a set of software timers. A software timer is basically a predefined variable that can be initialized by the program. Once initialized these predefined "timers" are automatically decremented at precise time intervals. Thus, the timer will reach zero after a time $T = N \times K_t$, where N is the initialization value and K_t is the time quantum between two successive decrements. The time quantum is generated using the TOC2 (Timer Output Compare 2) interrupt, which is generated every 4 milliseconds. Based on this interrupt, precise 100 ms, 1-second, and 1-minute intervals are generated.

When a time quantum expires, the associated set of variables (software timers) is scanned, and if their value is greater than zero, they are decremented.

The software timers are defined in MAP.ASM, which becomes:

```
           TITLE Memory map & global variables

           DATA
           ORG    RAMBASE
TIRQ       DS     1          ;flag set by TOC2 ISR
T4MS0      DS     1          ;4 ms timers
T4MS1      DS     1
T4MS2      DS     1
T4MS3      DS     1
T4MS4      DS     1
T4MS5      DS     1
T4MS6      DS     1
T4MS7      DS     1
T100MS0    DS     1          ;100 ms timers
T100MS1    DS     1
T100MS2    DS     1
T100MS3    DS     1
T100MS4    DS     1
```

```
T100MS5    DS     1
T100MS6    DS     1
T100MS7    DS     1
T1S0       DS     1          ;1 second timers
T1S1       DS     1
T1S2       DS     1
T1S3       DS     1
T1S4       DS     1
T1S5       DS     1
T1S6       DS     1
T1S7       DS     1
T1M0       DS     1          ;1 minute timers
T1M1       DS     1
T1M2       DS     1
T1M3       DS     1
T1M4       DS     1
T1M5       DS     1
T1M6       DS     1
T1M7       DS     1
CNT100MS   DS     1          ;counters
CNT1S      DS     1
CNT1M      DS     1

           CODE
           ORG    ROMBASE
           END
```

The TOC2 initialization routine, the interrupt service routine, and the code that scans and decrements the software timers are placed in TIMER.ASM.

To improve the readability of the program, a MACRO, called DJEQ (Decrement or Jump if EQual to zero), has been defined. It has the following structure:

```
DJEQ   MACRO   ?dest
       LOCAL   @A
       TST     ?dest
       BEQ     @A
       DEC     ?dest
@A     EQU     *
       ENDM
```

DJEQ receives a parameter, which is the symbolic address of the destination. It checks the content of the destination, and if greater than zero, it decrements it by one. If the destination is zero, DJEQ returns leaving the destination unchanged.

Here is the full listing of TIMER.ASM:

```
           TITLE  MAIN TIMER
           CODE
           JMP    TMRMAIN
```

```
*the initialization routine is called from INIT.ASM
ITIMER     LDAA   #$40       ;clear OC2F if any
           STAA   TFLG1
           STAA   TMSK1      ;enable TOC2 interrupt
           LDX    #T4MS0     ;clear all timers
           LDAB   #32
IT10       CLR    0,X
           INX
           DECB
           BNE    IT10
           LDAA   #25        ;init counters
           STAA   CNT100MS
           LDAA   #10
           STAA   CNT1S
           LDAA   #60
           STAA   CNT1M
           RTS
* TOC2 interrupt service routine
           VECTOR_TOC2
TOC2ISR    LDAA   #$40
           STAA   TFLG1      ;clear OC2F flag
           LDD    TOC2
           ADDD   #8000      ;next interrupt in 4 ms
           STD    TOC2
           INC    TIRQ       ;true the flag
           RTI               ;return from interrupt

* main timer task
TMRMAIN    TST    TIRQ       ;check for previous
           JEQ    TMREXIT    ;interrupt
           CLR    TIRQ
           DJEQ   T4MS0      ;scan the 4 ms timers
           DJEQ   T4MS1
           DJEQ   T4MS2
           DJEQ   T4MS3
           DJEQ   T4MS4
           DJEQ   T4MS5
           DJEQ   T4MS6
           DJEQ   T4MS7
           DEC    CNT100MS
           BEQ    TMR10
           JMP    TMREXIT
TMR10      LDAA   #25        ;reload counter
           STAA   CNT100MS
           DJEQ   T100MS0    ;scan the 100 ms timers
           DJEQ   T100MS1
           DJEQ   T100MS2
           DJEQ   T100MS3
           DJEQ   T100MS4
```

```
            DJEQ   T100MS5
            DJEQ   T100MS6
            DJEQ   T100MS7
            DEC    CNT1S
            BEQ    TMR20
            JMP    TMREXIT
TMR20       LDAA   #10
            STAA   CNT1S
            DJEQ   T1S0         ;scan the 1s timers
            DJEQ   T1S1
            DJEQ   T1S2
            DJEQ   T1S3
            DJEQ   T1S4
            DJEQ   T1S5
            DJEQ   T1S6
            DJEQ   T1S7
            DEC    CNT1M
            BEQ    TMR30
TMR30       LDAA   #60
            STAA   CNT1M
            DJEQ   T1M0         ;scan the 1min timers
            DJEQ   T1M1
            DJEQ   T1M2
            DJEQ   T1M3
            DJEQ   T1M4
            DJEQ   T1M5
            DJEQ   T1M6
            DJEQ   T1M7
TMREXIT     EQU    *            ;continue with the next
            END                 ;module
```

9.5 Exercises

SX 9.1

Using the software timers described in this paragraph, write a program that toggles PORTA bit 7 every 1 second, and PORTA bit 6 every 2.5 seconds.

Solution

Out of RESET, all I/O lines are configured as inputs. Therefore, we must configure the selected bits of PORTA as outputs, by adding the following lines to INIT.ASM.

```
            LDAA   #$C0         ;select PORTA bits 7 and 6
            STAA   DDRA
```

The main program looks like this:

```
           INCLUDE68HC11F1.DEF
           INCLUDEAS11.MAC
           INCLUDEMAP.ASM
           CODE
           VECTOR_RESET
RESET      EQU    *
           INCLUDEINIT.ASM
MLOOP      EQU    *         ;main loop start
           TST    T100MS0
           BEQ    M10
           BRA    M20
M10        LDAA   #10       ;restart timer
           STAA   T100MS0
           LDAA   PORTA
           EORA   #$80      ;toggle PORTA bit 7
           STAA   PORTA
M20        TST    T100MS1   ;next timer
           BEQ    M30
           JMP    MLOOP
M30        LDAA   #25       ;restart second timer
           STAA   T100MS1
           LDAA   PORTA     ;toggle PORTA bit 6
           EORA   #$40
           STAA   PORTA
           JMP    MLOOP
           END
```

The program uses two timers with the quantum 100 milliseconds, T100MS0 and T100MS1, which are tested one by one in an endless loop. When a timer reaches zero, the associated I/O line is toggled, and the timer is reloaded with the desired value.

X 9.2

Write a program that reads all the analog inputs every 20 milliseconds, and updates a set of variables AN0–AN7.

X 9.3

Write a program that uses the SCI reception interrupt. Upon reception of an ASCII code for ‘A’ ($41) the program answers with the last value read from the analog input AN0. The byte is transmitted as two ASCII characters, corresponding to its hexadecimal representation.

X 9.4

Modify the schematics of the module described in this chapter, by adding an external 32 K RAM circuit, selected by CSGEN. Describe the initialization sequence in this case.

10

AVR Development Board

10.1 In this Chapter

This chapter describes a simple, yet flexible development board, based on AT90S8535, for the study of the AVR microcontrollers. This board can be used to test most of the AVR projects presented in this book.

10.2 The Hardware

The schematic of the development board is shown in Fig. 10.1. The circuit comprises the following functional blocks:

1. Microcontroller
2. Clock circuit
3. RESET circuit
4. Output buffers
5. ISP interface
6. RS232 interface
7. Power supply circuit.

The microcontroller is an AT90S8535-P in a DIP40 package. Note that this microcontroller is pin by pin, hardware compatible with other members of the analog series of AVR microcontrollers, like ATMega8535 and ATMega16. However, there are many differences between these microcontrollers. Consult the data sheets before making the replacement.

The external clock circuit uses an 8 MHz crystal, Q1, and the capacitors C10, C11 (12–47 pF). The RESET circuit consists of the resistor R1 (10 K), and the capacitor C1 (10 F/10 V). These values are not critical, because AT90S8535 contains internal signal conditioning circuits for the RESET signal.

The digital input lines are connected to PORTC, and are pulled up to V_{cc} with resistors. A group of LEDs has been included, to show the status of each input.

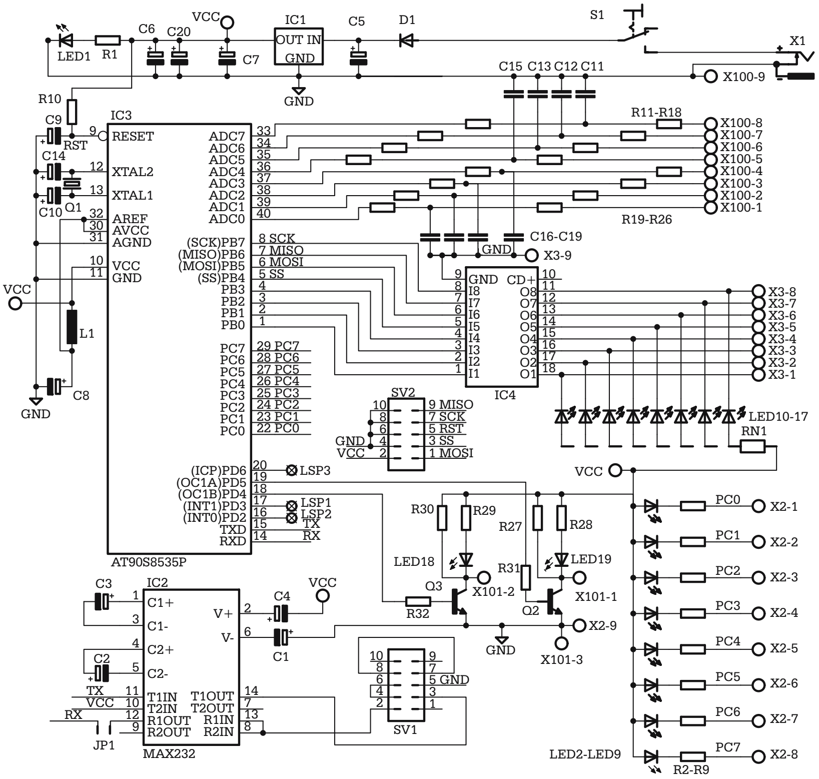


Fig. 10.1. Schematic of the AVR development board

The idle status of the input lines is HIGH. An input line is active when pulled to GND (for example by the contact of a relay). In this case, the associated LED will light, indicating the active input.

The analog input lines use PORTA, and are provided with RC filters for noise rejection. Note that these lines can be programmed to work as digital I/O lines. When using the lines of PORTA for digital output, reduce the values of the filter capacitors C11–C19, or remove them completely.

For digital output, the system uses the eight lines of PORTB of the microcontroller. They are buffered with IC4 – ULN2803, an array of eight open collector transistors able to sink up to 300 mA, at 30 V. This allows driving higher current loads, like relays, lamps, coils, etc. Each output line is also provided with a LED-resistor group to indicate its active status.

The output lines OC1A, OC1B of the MCU are buffered with two 2N2222 transistors, Q2 and Q3. They can be programmed to work either with timer 1 in output compare mode, or can be used as PWM outputs. LED18 and LED19 indicate the status of these lines.

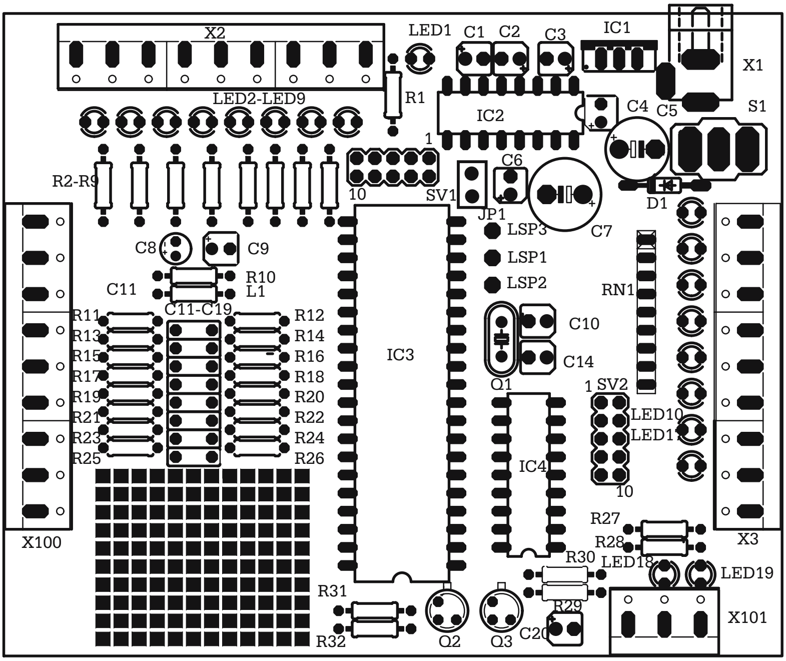


Fig. 10.2. PCB layout of the AVR development board

The input lines INT0, INT1, and INPUT CAPTURE of the MCU are not available on the external connectors. Instead, they are connected to test points and can be used for extensions of the board in the custom area.

The ISP connector SV2 is configured to make the board compatible with AVRISP, for programming the internal flash memory of the microcontroller. The ISP uses the MOSI, MISO, SCK and RESET lines of the microcontroller. The SS (Slave Select) signal has been added on the ISP connector, to allow the use of the ISP connector for a standard SPI interface.

The RS232 interface is a typical MAX232 implementation, and uses the connector SV1. Note that all the projects and examples in this book use the same circuits and connector layouts for RS232 and ISP.

The power supply circuit comprises the diode D1, the voltage regulator IC1, and filter capacitors C5, C6, C20. LED1 indicates when the circuit is powered.

Figure 10.2 shows the component layout of the printed circuit board for this module. Detailed execution drawings of the printed circuit board, as well as the complete component list, are available on the accompanying CD.

10.3 Testing the Circuit

Once you have the board assembled and visually checked for short-circuits, execute the following steps to test the circuit. It is recommended to use appropriate sockets for all the ICs, except IC1.

1. Power up the circuit without having the ICs inserted in their sockets.
2. Check for a value of +5 V of V_{cc} .
3. LED1 must be ON.
4. The RESET line (pin 9 of IC3) must be around +5 V.
5. Pins 10, 30, and 32 of IC3 must be at +5 V.
6. Pins 31 and 11 of IC3 must be at GND.

If everything is OK, remove the power and insert the ICs in their sockets.

Using the ISP interface, load one of the test programs described in the following paragraphs into the MCU internal flash memory and check the behavior of the module on the LEDs associated with the I/O lines.

10.4 The Software

Atmel Corporation offers an excellent, free programming and debugging tool to support AVRs, called AVRStudio. The latest version of AVRStudio is available for download at the Atmel Corporation web site.

Step 1. Installing the Avrstudio

To install AVRSTUDIO, unzip the downloaded archive in a new folder on your hard disk, then launch SETUP.EXE and follow the instructions on screen.

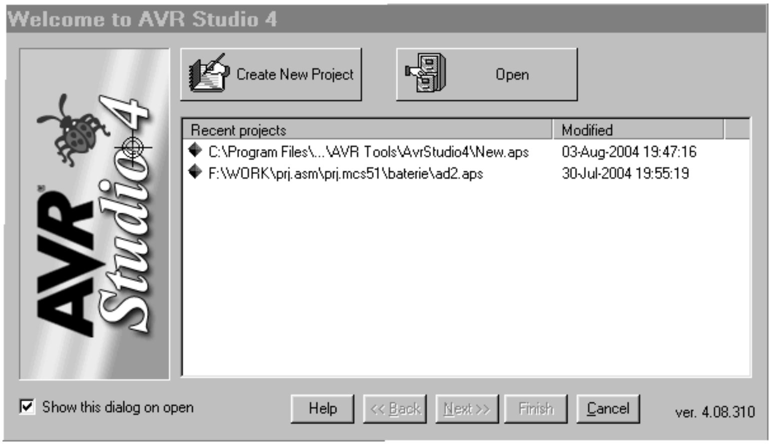
Once the program has installed, read the detailed help file included, for an overview of the capabilities of the program. This help file also contains a good description of the instruction set of AVR microcontrollers.

Step 2. Creating a New Project

When launched, AVRStudio4 prompts the user whether to create a new project, or open an existing one. Figure 10.3 shows a snapshot of the first dialog window of AVRStudio.

If you choose the option *Create new project*, the following information is required:

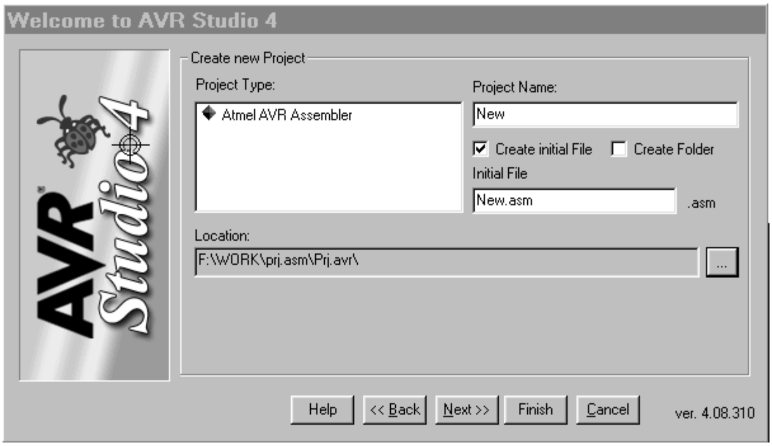
- the name for the new project,
- the name of the initial file,
- the working directory where the project will be placed.



Recent projects	Modified
◆ C:\Program Files\...\AVR Tools\AvrStudio4\New.aps	03-Aug-2004 19:47:16
◆ F:\WORK\prj.asm\prj.mcs51\baterie\ad2.aps	30-Jul-2004 19:55:19

☑ Show this dialog on open

Fig. 10.3. Snapshot of the first dialog window in AVRSTUDIO4



Project Type: ◆ Atmel AVR Assembler

Project Name: New

☑ Create initial File ☐ Create Folder

Initial File: New.asm .asm

Location: F:\WORK\prj.asm\Prj.avr\

Fig. 10.4. Snapshot of the dialog window for *Create new project* in AVRSTUDIO4

See Fig. 10.4 for a snapshot of the dialog window for *Create new project.*

Fill in the information required and press the *Next* button.

One final dialog window is presented, as shown in Fig. 10.5, where the user is prompted to choose the debugging platform and the device type. For this particular development board, choose AVR Simulator as the debugging platform, and AT90S8535 for the device, then press *Finish* to start writing the new program.

Step 3. Writing and Testing Simple Programs in AVRStudio4

Before starting, note a few syntax differences between the assembler included in AVRStudio and ASHC11, the assembler used for HC11.

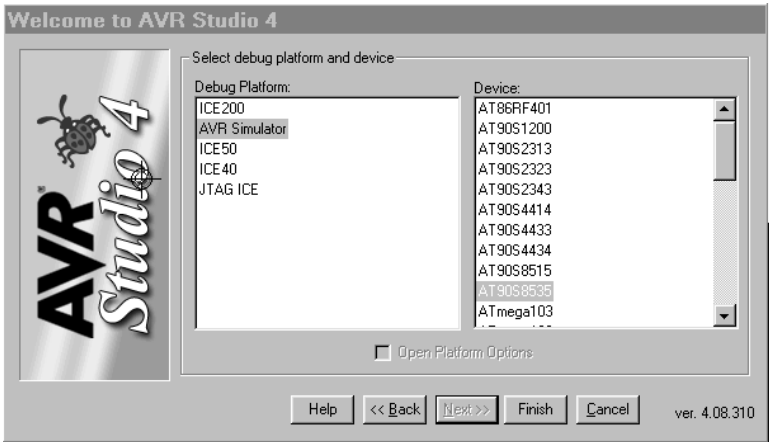


Fig. 10.5. Snapshot of the dialog window for selecting the debug platform and device

For example, labels must be terminated with colon ‘:’ when they are defined:

```
Loop:                   ;note the `:' in label definition
       .....
       Rjmp   Loop      ;do not use `:' when invoking
                        ;the label
```

Another important difference is that *all* assembler directives start with a period **‘.’** and the filenames specified for Include must be delimited by double quotes: for example

```
       .Include ``8535def.inc''
```

The directives that switch between DATA and CODE segments are now named .DSEG and .CSEG, respectively.

Here is an example of a fragment of code illustrating these distinctive features:

```
       .Include ``8515def.inc''
       .CSEG            ;select CODE segment
       Ldi    R16,0xFF;load R16 with
                        ;hex value $FF
       Out    Ddrb,R16;configure PORTB
                        ;as output port
Loop:
       In     R6,Pinc;read PORTC in R6
       Com    R6        ;complement R6
```

To assemble the program press [F7].The assembler creates several output files having the name chosen for the project, and the extensions .HEX (this is the executable code), .LST (this is the list file), and .MAP (this contains all the symbols defined in the program, and the invoked include files)

The list file also contains the error messages (if any), so this is where you have to look for explanations, in case of errors.

Once you have your program assembled without errors, you can invoke the simulator to run it by pressing *[Ctrl]+[F7]*.

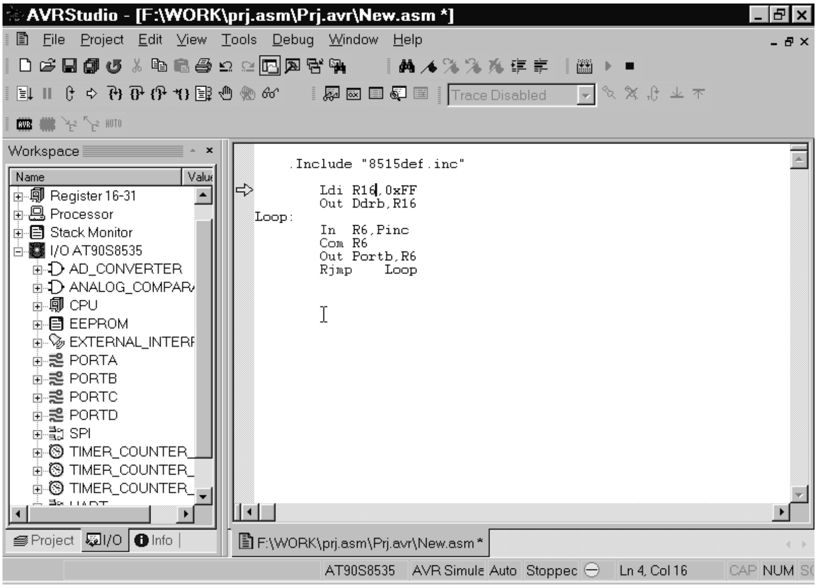
If you want to run the program step by step, press [F11]. At each step, the whole status of the microcontroller (program counter, registers, memory, I/O lines, etc.) is updated, and displayed, or *even modified* by the user.

See Fig. 10.6. for a snapshot of the simulator screen. Notice the left window, which displays the status of hardware resources for that particular step. In the right window, a cursor indicates the current instruction.

You can also choose to run the program normally ([F5]), to break the current execution (*[Ctrl]+[F5]*), to add breakpoints in the program (*[Ctrl]+[F9]*), etc. All these options are offered in the *Debug* pull-down menu.

The AVRStudio simulator is an exceptional tool that lets you perform complex and detailed test and debug procedures on your program, without having to download the program into the microcontroller's flash memory.

Explore for yourself the numerous features and tools available in AVRStudio, by launching the on-line help (Start>Programs>Atmel AVR Tools>AVR Tools Online Help).



```
        .Include "8515def.inc"

            Ldi R16,0xFF
            Out Ddrb,R16
    Loop:
            In  R6,Pinc
            Com R6
            Out Portb,R6
            Rjmp    Loop
```

Fig. 10.6. Snapshot of the AVRStudio simulator screen

Step 4. Loading the Executable Code into the Flash Memory

If the assembly process completes without errors, the assembler generates a file with the extension .HEX, which contains the actual executable code of the program. This code can be loaded into the microcontroller's flash memory through the ISP interface, following the instructions in Appendix A9.

Step 5. Choosing a Structure for Your Application

The principles on how to organize a software application, described in Chap. 9 for HC11, can be entirely followed when writing AVR programs. The assembler built in AVRStudio accepts MACROs as well as the .INCLUDE directive. Although the macro capabilities of the AVR assembler are weaker than those of the HC11 assembler, you can still use MACROs to simplify your work.

The following examples illustrate what MACROs can do:

```
.MACRO    LDXI
    ldi
    xh,high(@0)
    ldi
    xl,low(@0)
```

```
.MACRO     LDX
    lds    xh,@0
    lds    xl,@0+1
.ENDM
```

```
MACRO      TSX
    out    SPH,XH
    out    SPL,XL
.ENDM
```

```
.MACRO    TXS
    in    XH,SPH
    in    XL,SPL
.ENDM
```

Though very simple, the following example illustrates the concept on the general structure of a software application for microcontrollers. This program toggles the lines of port B every 500 milliseconds. Since each line of port B is connected to a LED, this program can be used to test the development board, without the need to measure any signal. Here is the listing of the main program module LEDMAIN.ASM.

```
           .include ‘‘8535def.inc’’
           .include ‘‘map.asm’’    ;application variables
           .cseg
           .org     0
reset:                             ;reset vector
           rjmp     init
           .org     9
vector_t0_ov:                      ;timer0 overflow vector
           rjmp     isr_t0_ov
init:
           .include ‘‘init.asm’’   ;all initializations
                                   here
            sei                   ;enable interrupts
            ldi      tmp1,time_s_timer0
            sts      s_timer0,tmp1 ;start soft timer
main_loop:
            lds      tmp1,s_timer0
            tst      tmp1          ;check if timer
                                  expired
            brne     main_loop
            ldi      tmp1,time_s_timer0
            sts      s_timer0,tmp0 ;restart timer
            rcall    toggle        ;toggle portb
            rjmp     main_loop
            .include ''timer.asm'' ;timer routines
            .include ''io.asm''    ;i/o routines
```

The definitions file 8535def.inc is provided by AVRStudio and contains a number of equation directives (.equ) that associate symbolic names to the addresses of all the ressources. Here is a fragment of the contents of this file:

```
.device AT90S8535
;***** I/O Register Definitions
.equ SREG   =$3f
.equ SPH    =$3e
.equ SPL    =$3d
....
.equ PORTA  =$1b
.equ DDRA   =$1a
.equ PINA   =$19
.equ PORTB  =$18
.equ DDRB   =$17
.equ PINB = $16
.....
```

The following Include file, MAP.ASM, contains application-specific variables and definitions. Below is the listing of MAP.ASM for this particular application:

```
.def rint=r1          ;just change the names
.def rsav=r2          ;of some registers
.def tmp1=r16
.def tmp2=r17
.def tmp3=r18
;define variables in data segment
.dseg
s_timer0:     .byte 1 ;software timer0
s_timer1:     .byte 1 ;software timer1
;constants used by timer
.equ prescaler_constant=5
.equ ft_div=177
.equ time_s_timer0=50
```

The CODE segment starts with the interrupt vector area. In this example, only two vectors are used: the reset vector, and the vector for the interrupt for timer0 overflow. The reset vector directs the program to the initialization sequence, and the timer0 overflow vector contains a jump instruction to the associated interrupt service routine, isr_t0_ov.

INIT.ASM contains the initialization sequence, which starts by loading the stack pointer SP with the last address of the internal RAM (Ramend), then calls the initialization routines for timer0, and I/O port B.

After the initialization sequence, the program enables the interrupts, loads the software timer s_timer0 with a constant (time_s_timer0), which defines the time between the moments when port B changes its status, then enters the main loop.

This software timer is the key element of the program. Basically, s_timer0 is an 8-bit RAM variable defined in MAP.ASM, and decremented every 10 milliseconds by the interrupt service routine for timer0 overflow. If the timer is loaded with the value N, it will reach zero after a time $T = N \times 10\,\text{ms}$.

In this example, N is the constant time_s_timer, defined in MAP.ASM with the value 50, which gives a total time of 500 ms between the moments when port B is toggled. When the software timer expires, port B changes its status, then the timer is reloaded, and the program continues in an endless loop.

The resource-specific routines are placed outside the main loop in the modules TIMER.ASM and IO.ASM. TIMER.ASM contains the initialization sequence for timer0, organized as a subroutine init_timer0, and the interrupt service routine that implements the software timer.

Here is the listing of TIMER.ASM:

```
init_timer0:
           ldi    tmp1,prescaler_constant
           out    tccr0,tmp1
           ldi    tmp1,ft_div
           mov    rint,tmp1
           out    tcnt0,tmp1
           ldi    tmp1,$01
           out    timsk,tmp1
           ret
```

This routine chooses the division factor for the prescaler, and the number of clocks before overflow, then enables the interrupt for timer0 overflow.

```
isr_t0_ov:
           out    tcnt0,rint
           in     rsav,sreg
           push   tmp1
           rcall  soft_timer
           pop    tmp1
           out    sreg,rsav
           ret
```

Note how this routine saves and restores the CPU status. The register rint (an alias for r1) is preloaded with the number of clocks before tcnt0 overflows, so that tcnt0 can be reloaded as fast as possible.

```
soft_timer:
           lds    tmp1,s_timer0
           tst    tmp1
           breq   sft1
           dec    tmp1
           sts    s_timer0,tmp1
sft1:
           lds    tmp1,s_timer1
           tst    tmp1

           breq   ex_sft
           dec    tmp1
           sts    s_timer1,tmp1
ex_sft:
           ret
```

The routine soft_timer implements two independent software timers, s_timer0 and s_timer1. In principle, the number of distinct timers that can be implemented using this technique is limited only by the time required to execute the interrupt service routine. If a larger number of software timers is required, use the technique described in Chap. 9 for HC11 to reduce the execution time of the interrupt service routine.

Step 6. Adding a New Task to an Existing Application

The structure of the software described in the previous paragraph can be used for almost any application. Adding a new task to an existing application involves the following steps:

1. Modify MAP.ASM to define more variables if needed.
2. Write a new software module if a new resource is required. It is recommended to keep all the routines associated with specific resources in separate files (e.g. TIMER.ASM, IO.ASM, UART.ASM, ADC.ASM, etc.)
3. Modify the main program so that it includes calls to subroutines that actually execute the new task.

In the following example, the program reports the status of port B as a string of two ASCII digits, followed by CR+LF, over the asynchronous serial communication line, when the ASCII code for ‘?’ is received on the serial line. Note that port B is toggled every 500 ms, as described in the previous example.

The UART initialization routine and the communication routines are placed in a separate file UART.ASM, invoked in the main module using the “.include” directive. A separate file, LIB.ASM, is used for miscellaneous library routines – in this case hex-to-ASCII conversion routines used to prepare data for serial communication.

Here is the listing of the main module UARTMAIN.ASM:

```
            .include ‘‘8535def.inc’’
            .include ‘‘map.asm’’
            .cseg
            .org     0
reset:
            rjmp     init
            .org     9
vector_t0_ov:
            rjmp     isr_t0_ov
init:
            .include ‘‘init.asm’’
            sei

            ldi      tmp1,time_s_timer0
            sts      s_timer0,tmp1
main_loop:
            rcall    get_uart
            brcc     main0
            cpi      tmp1,’?’
            brne     main0
            rcall    send
main0:
            lds      tmp1,s_timer0
            tst      tmp1
            brne     main_loop
            ldi      tmp1,time_s_timer0
            sts      s_timer0,tmp1
            rcall    toggle
            rjmp     main_loop
            ,include ‘‘timer.asm’’
            .include ‘‘uart.asm’’
            .include ‘‘io.asm’’
            .include ‘‘lib.asm’’
```

Note the two new subroutine calls included in the main loop: get_uart, and send.

Get_uart checks if a character is available in the receiver’s data register, reads the character in tmp1, and sets the carry bit to inform the main program. When no character is available, get_uart returns carry = 0.

```
get_uart:
            clc
            sbis   usr,rxc     ;check rxc status bit
            ret
            in     tmp1,udr    ;get character received
            sec                ;set carry
            ret                ;and return
```

Send prepares and sends over the communication line a string comprising two ASCII digits corresponding to the hexadecimal value of each nibble of port B, plus CR and LF.

```
send:
           rcall  hex_asc    ;convert tmp1 to ASCII
           rcall  put_uartw  ;send tmp1
           mov    tmp1,tmp2  ;next character in tmp2
           rcall  put_uartw  ;send next character
           ldi    tmp1,$0D   ;CR
           rcall  put_uartw
           ldi    tmp1,$0A   ;LF
           rcall  put_uartw
           ret
```

Put_uartw waits for the status bit UDRE to be set, then writes the content of tmp1 to UDR. It does not check for valid ASCII characters before sending.

```
put_uartw:
           sbis   usr,udre   ;check if TX
                             ;register empty
           rjmp   put_uartw  ;wait until ready to send
           out    udr,tmp1   ;send the byte
           ret
```

10.5 Exercises

SX 10.1

Use two software timers to make the upper and lower nibble of port B toggle at different time intervals.

Solution

The two software timers s_timer0 and s_timer1 described in the previous paragraphs must be loaded with different constants to obtain different time intervals for toggling the lines of port B. Here is how LEDMAIN.ASM should be modified for this:

```
init:
           .include ``init.asm''
           sei
           ldi      tmp1,time_s_timer0
           sts      s_timer0,tmp1      ;start s_timer0
           ldi      tmp1,time_s_timer1
           sts      s_timer1,tmp1      ;start s_timer1
main_loop:
           lds      tmp1,s_timer0
           tst      tmp1
           brne     main1
           ldi      tmp1,time_s_timer0 ;restart s_timer0
           sts      s_timer0,tmp1
           rcall    toggle1            ;act on port
main1:
           lds      tmp1,s_timer1      ;check s_timer1
           tst      tmp1
           brne     main_loop
           ldi      tmp1,time_s_timer1 ;restart s_timer1
           sts      s_timer1,tmp1
           rcall    toggle2            ;act on port
           rjmp     main_loop
```

Each of the subroutines toggle1 and toggle2 must affect only one nibble of port B, leaving the other nibble unchanged:

```
toggle1:
           in      tmp1,portb ;read port
           mov     tmp2,tmp1  ;save it to tmp2
           andi    tmp2,$0F   ;mask lower nibble
           com     tmp1
           andi    tmp1,$F0   ;mask upper nibble
           or      tmp1,tmp2  ;recompose
           out     portb,tmp1 ;write new value
           ret                ;to the port
```

11

8051 Development Board

11.1 In this Chapter

This chapter contains the description of a simple development board for the study of the 8051 family of microcontrollers.. Unlike the development boards dedicated to the HC11 and AVR families, presented in the previous chapters, this project allows the user to load and execute programs in an external RAM area, addressed to be visible both in the program memory and data memory address space.

11.2 Hardware

The schematic of the board is presented in Fig. 11.1, 11.2, and 11.3. Figure 11.1 shows the microcontroller IC10, the bus demultiplexer IC7, the RS232 interface IC12, the RESET and clock circuits, and the ISP connector SV1.

Note the presence of the two NAND gates IC6C and IC6D, which implement the logic function AND between the signals RD\ (Read) and PSEN\ (Program Store Enable), both active LOW, and generate the signal RDPSEN. RDPSEN is active LOW when either RD\ or PSEN\ is LOW. Connecting this signal to the RD\ input of an external RAM circuit allows the use of this RAM to store program as well as data.

The microcontroller can be any of 8032, 8051, 8052, AT89C51, AT89C52, etc. in a DIP40 package. The input EA\ (External Access) of the MCU is connected to the jumper JP1 to allow the use of the internal ROM, if this is available. Connect the jumper so that EA\ = 1 to use the internal ROM. When the external ROM is used, EA\ must be connected to GND.

The clock circuit comprises the crystal Q1 (14.74 MHz) and the capacitors C6, C7 (22 pF). C5 (10 μF) and R4 (10 K) implement the RESET circuit. The pushbutton S1 is provided to allow manual reset of the circuit.

The latch SN74HC573 (IC7), controlled by the signal ALE (Address Latch Enable), is used to demultiplex the external data bus, by storing the lower half (A0–A&) of the address bus.

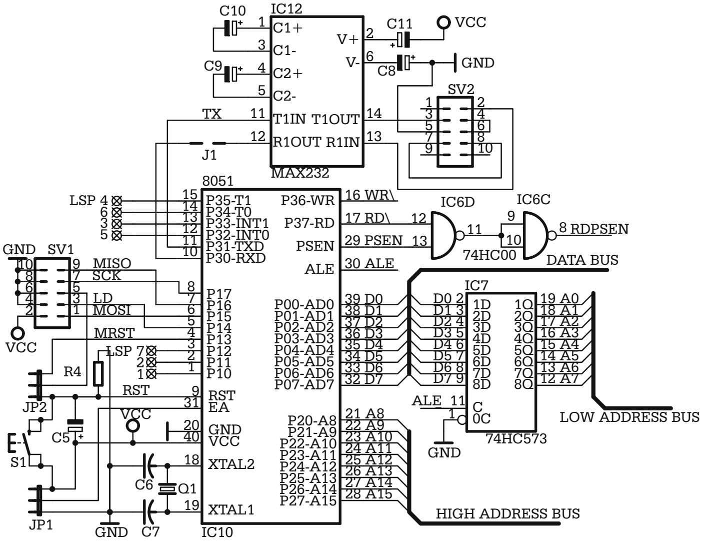


Fig. 11.1. MCU, bus demultiplexer, RESET, clock, and RS232

The ISP (In System Programming) connector SV1 has been included to allow the use of microcontrollers like AT89LS51, AT89S52 that are provided with this feature. In this case, the jumper JP2 must be connected to bring the signal RST to SV1 pin 5.

The memory circuits and the address decoder are presented in Fig. 11.2. IC5 (AT28C64) is an 8-kilobytes EEPROM, having the chip select input CS\ connected to the signal CS0000, generated by the address decoder IC9 (7445), and the output enable input OE driven by the signal PSEN, generated by the MCU.

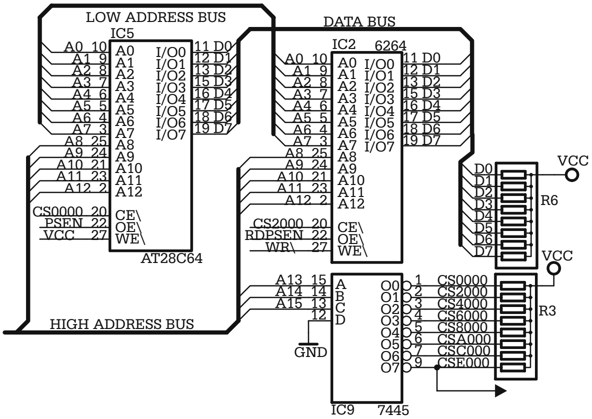


Fig. 11.2. External memory and address decoder

IC2 is a generic 8-kilobytes RAM circuit (6264) controlled by the selection signal CS2000, the RDPSEN signal prepared with the gates IC6, and the write control signal WR, generated by the MCU.

Note the pull-up resistor network R6 connected to the data bus.

The address decoder IC9 (7445) is used here to generate eight active LOW output signals, CS0000 to CSE000, each corresponding to an 8-kilobytes address range. Only three of these outputs are actually used in this circuit: CS0000, which decodes the address range between $0000 and $1FFF, used to select the external ROM; CS2000, corresponding to the address range $2000–$3FFF, used for the external RAM, and CSE000 ($E000–$FFFF), used for the external I/O ports.

Figure 11.3 shows the circuits that implement the external input and output ports and the address decoders for these ports. The actual output port is IC3 (SN74HC574), controlled by the signal OUTE000 (active HIGH), and the input port is implemented with IC11 (SN74HC573), controlled by the signal INE000, (active LOW).

The address decoder for the external I/O ports is implemented with the circuit IC8 (74HC138), along with the NAND gates IC6A and IC6B, which decode as I/O addresses any address combination of the type: 111x xxxx xxxx xx00. Note that the two registers have the same address, but the selection signal INE000 is active only for read operations, while OUTE000 is active for write operations.

A MOVX instruction at the address $E000 will cause the selection of either the input or output port, depending on the status of the RD\ signal.

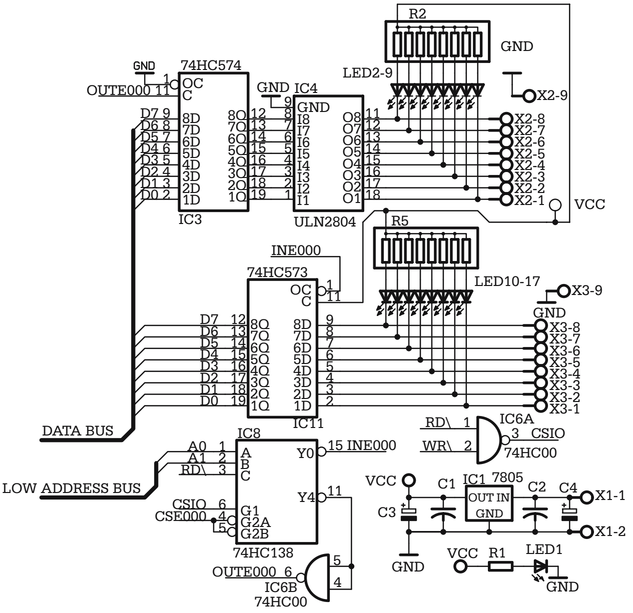


Fig. 11.3. I/O ports and power supply

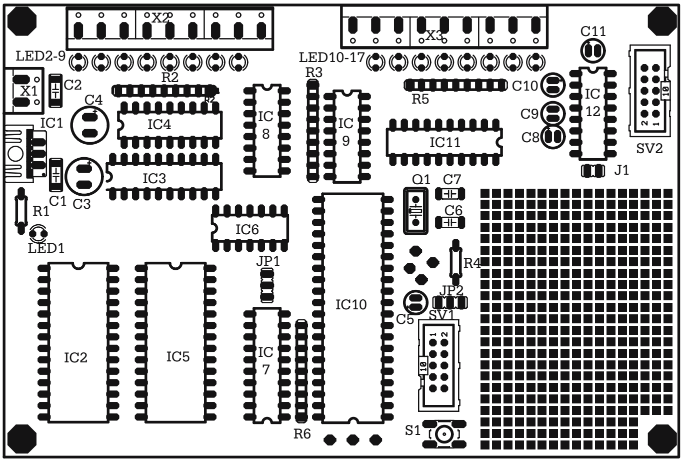


Fig. 11.4. Component layout for the PCB of the 8051 development board

The output port is buffered with IC4 (ULN2804). LED2–LED9 and the resistor network R2 have been included to indicate the status of the output lines at any moment. Similarly, LED10–LED17 and R5 are used to indicate the status of the input lines.

Figure 11.3 also shows the voltage regulator IC1 (7805) which generates the power supply voltage V_{cc} for the board. LED1 indicates the power present. The external power is applied on X1-1, X1-2, with the polarity indicated, and must be in the range +12 V to +16 V/250 mA.

The component layout of the PCB for this schematic is shown in Fig. 11.4.

11.3 The Software

11.3.1 Installing the Cross-Assembler

From the many freeware 8051 cross-assemblers available, we chose ASEM-51, created by W.W. Heinz, because it is fully compatible with the Intel syntax, allows nested include directives, and macros, and is very well documented. This tool is available for free download at http://plit.de/asem-51/.

To install the tools for creating and testing 8051 software, perform the following steps:

1. Locate the subfolder ASM51 of the folder SOFTWARE on the accompanying CD, and copy it to your hard drive, with all its content. This contains all the software examples described in this chapter.
2. Download the archive ASEM51 from http://plit.de/asem-51/ and unzip all the files to a temporary folder.
3. Copy the executable ASEMW.EXE to the hard disk in the folder ASM51.

4. Follow the instructions from Chap. 9 to create a new DOS window, having the working directory C:\ASM51, and a shortcut on your desktop linking to it.

The assembler is invoked from the command line by typing *ASEMW sourcefile.*

The extension of the source file is assumed to be .A51. If another extension is used, this must be specified in the command line, e.g. *ASEMW sourcefile.asm*.

If you are not familiar with MS-DOS™ programs, consult Appendix A15 for the description of a software utility that allow invoking the assembler from a Windows™ application. This utility also contains a text editor for creating and editing the assembler source files.

The assembler generates two output files, both having the same name as the source file, and the extensions .LST and .HEX. The list file, *sourcefile.lst* contains the detailed listing of the program and information on the errors occurred during the assembly process. The hex file, *sourcefile.hex* contains the actual executable code in *Intel Hex format*. The ASEM-51 package also contains a utility program to convert this type of file to plain binary files, called *HEXBINW.EXE*, but this is seldom necessary because most EPROM programmers directly handle Intel hex files.

The accompanying CD contains several examples of 8051 assembler source files, which can be used to verify that the assembler is correctly installed.

11.3.2 Writing and Testing Simple 8051 Programs

ASEM-51 accepts INCLUDE directives. This means that the user programs can be organized according to the structure described in the previous chapters for HC11 and AVR microcontrollers.

The simplest program (inout.asm) reads the status of the external input port and writes this value to the external output port. This is a good and simple method to test the hardware, and to get in touch with the software tools. Here is the listing of this program:

```
           org    0000h      ;reset sends here
           mov    Dptr,#0e000h
main_loop:
           movx   a,@dptr    ;read input port to A
           movx   @dptr,a    ;write A to output port
           sjmp   Main_loop  ;in an endless loop
           end
```

The next example is a bit more complicated. It uses the timer0 overflow interrupt to implement a user programmable software timer, which is used to determine the time intervals between the moments when the output port toggles its status. Compare this example with those presented in Chap. 9 and 10 and note the similarities and the differences. Besides the syntax differences (like $include instead of .include,) another significant difference is that, in this case, the symbolic names of the special function registers are automatically recognized by the assembler, and it is not necessary to define them in a special include file.

```
            $include (map.asm)      ;map
            Cseg                    ;code segment
            Org      Startaddr      ;start address
Rst:
            Ajmp     Init           ;go to init
            Org      rst+0bh        ;timer0 ovf interrupt
Int_t0_ov:                          ;timer0 vector
            Ajmp     isr_t0_ov
Init:
            $include (init.asm)     ;all initializations
            Mov      s_timer1,#time_st1;start software timer1
Mainloop:
            Mov      a,s_timer1     ;read timer value
            Jnz      Mainloop       ;loop until expires
            Mov      s_timer1,#time_st1;restart timer1
            Mov      a,port_v       ;get the value for
                                    port
            Cpl      A              ;complement it
            Mov      port_v,a       ;save new value
            Movx     @dptr,a        ;out to port
            Sjmp     Mainloop       ;loop forever
            $include (timer.asm)    ;timer0 routines
            End
```

MAP.ASM contains, as usual, the application specific variables and constants.

```
            Time_st1 equ 50     ;constant to timer1
            Startaddr equ 0000h ;start address
            Dseg                ;data segment
            Org     28h
s_timer1:   ds      1           ;software timer1
port_v:     ds      1           ;out port value
icnt1:      ds      1           ;interrupt counter
```

time_st1 is the value that, multiplied by the timer quantum, determines the actual time until the software timer expires. In this example, time_st1 is 50 and the timer quantum is 10 ms, which leads to 500 ms total time between the moment when the software timer is loaded, and the moment when it reaches zero.

The actual software timer is a RAM location s_timer1, and port_v is another RAM location used to store the value to write in the external output port. It is initialized with the value 055h. Finally icnt1 is an 8-bit software counter initialized with the value 100. Here is the whole initialization sequence located in the file INIT.ASM:

```
            acall   init_timer0;init timer0 system
            mov     port_v,#55h;start with 055h
            mov     icnt1,#100 ;100x0.1mS=10 ms
            mov     dptr,#0e000h;dptr points to port
```

The subroutine init_timer0, called here, is located in the file TIMER.ASM.

Note that the initialization sequence does not contain initialization of the stack pointer SP. There is no need for this because the SP of 8051 is hardware initialized with the value 07h.

The module TIMER.ASM contains the subroutine init_timer0, and the interrupt service routine for timer0 overflow:

```
init_timer0:
           mov    tmod,#02    ;timer0 in mode 2
           mov    th0,#131    ;256-131=125
                              ;125x0.8us=0.1ms
           mov    tl0,#131
           setb   et0         ;timer0 interrupt enable
           setb   ea          ;global interrupt enable
           setb   tr0         ;start timer0
           ret
```

Remember that in operating mode 2, timer0 is reloaded by hardware, at every overflow, with the content of th0. For the given value of the frequency of the oscillator, th0 is initialized so that the overflow interrupts from timer0 occur every 0.1 ms.

The interrupt service routine is listed below:

```
Isr_t0_ov:
           push   psw         ;save psw
           push   acc         ;save accumulator
           djnz   icnt1,exit_st;decrement & exit if zero
           mov    icnt1,#100
St1:
           mov    a,s_timer1  ;get timer1 value
           jz     exit_st     ;exit if zero
           dec    a           ;else decrement it
           mov    s_timer1,a
Exit_st:
           pop    acc         ;restore accumulator
           pop    psw         ;restore psw
           reti               ;return from interrupt
```

The quantum of the software timer is determined by the interval between two consecutive overflow interrupts from timer0, and by the value loaded in icnt1. In this example, the interval between interrupts is 0.1 ms, and the value loaded in icnt1 is 100, and the resulting timer quantum is 10 ms.

The software timer s_timer1 is decremented by the interrupt service routine every 10 ms, only if its value at the moment of the interrupt is non-zero. The main program must load the desired number of quanta in s_timer1, then check the moment when s_timer1 reaches zero. In this example, the software timer expires after 50 × 10 ms = 500 ms.

11.3.3 Loading and Executing Programs in the External Ram Memory

Consider the memory of the development board described in this chapter, presented in Fig. 11.2.

Note that the external RAM is visible both in the address space of the program memory, and in the address space of data memory.

This allows, in principle, the execution of a program loaded in the external RAM. The problem is that the RAM memory is volatile, and the program must be loaded into the memory via the serial communication line after each power down.

The solution for this is to place in ROM a special program, which allows the user to enter into a “dialogue” with the board, and to launch at least two “commands”: one to load a program in RAM, and the other to execute the program. Other possible commands are for displaying and/or modifying the contents of the code or data memory.

The accompanying CD contains the file MONITOR.HEX, which is the executable code of a program that does all this. This very simple monitor program uses the RS232 interface to connect the development board to a computer running a standard terminal program, like HyperTerminal. The link cable needed is a null-modem cable, and the communication parameters are 9600, N, 8, 1, i.e. 9600 baud, 8-data bits, no parity bit, one stop bit.

When executed, the monitor program starts by sending the prompter ‘>’ to the terminal, then awaits a command string. All the command strings must be ended by CR. Table 11.1. contains the list of valid commands accepted by this monitor.

The command strings consist of one or two letters, followed by a two-digit (xx) or four-digit (xxxx) hexadecimal parameter signifying a memory address referred to by the command. No delimiters are required between the letter that defines the command and the parameter. For example, the effect of the command G2000 is an unconditional jump to the address 2000h.

There are two commands that do not require any parameter: ‘?’ and ‘S’. The command ‘?’ displays a help screen, and ‘S’ displays a list of entry point addresses for several ROM resident subroutines, available for the user by means of the acall instruction (absolute address call).

The ‘L’ commands (LH or LB) start the process of loading a program into the external RAM of the board. No parameter is required for LH because the program is

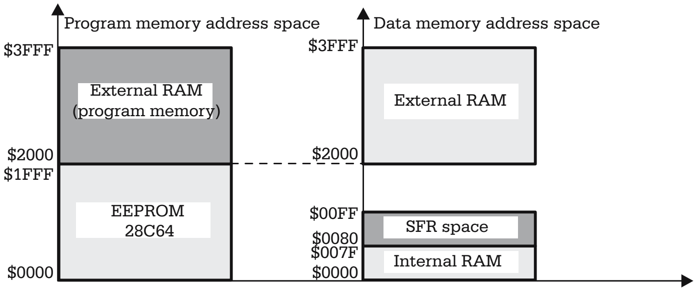


Fig. 11.5. Memory map for the 8051 development board

Table 11.1. List of valid commands of the monitor program

Command syntax	Effect
?	Display a help screen on the terminal
LH	Load user program in RAM, via RS232, as Intel HEX file
LBxxxx,yyyy	Load binary file starting address xxxx. yyyy is the byte count
Gxxxx	Execute program starting the address xxxx (four hex digits)
Dxx	Dump content of internal RAM starting the address xx
XDxxxx	Dump content of external RAM starting xxxx
XSxxxx	Substitute content external RAM starting xxxx
Cxxxx	Dump code memory starting xxxx
Ixxxx	Set pseudovectors for a program starting at xxxx
S	Display a list of entry points for some monitor subroutines

presented in Intel hex format, which contains information about the address where the code must be stored.

After receiving the command ‘LH’, the monitor displays a new prompter ‘:’ and waits indefinitely until the transmission process begins. When the transmission ends, the monitor returns to the main prompter, and waits for a new command.

The command LB acts similarly, but two parameters must be supplied, separated by a comma, each consisting in four hex digits. The first parameter indicates the starting address; the second is the number of bytes expected.

The terminal program used for communication must allow transmission of user specified text or binary files. HyperTerminal has this feature.

See also the utility software ASMEDIT.EXE on the accompanying CD, described in Appendix A15. This contains a terminal program that allow serial communication between a computer and the development board.

The command ‘Ixxxx’ requires special attention because it introduces a new concept: *the pseudovectors*. As mentioned before, the interrupt vectors are hardwired and link each interrupt to a ROM address, containing the interrupt service routine or a jump to this routine. Apparently, this seems to be a major limitation when writing programs aimed to run in RAM, because all the interrupt vectors point to ROM addresses. In fact, there is a simple solution to this problem. The fix is to reserve two RAM locations, called pseudovectors, for each interrupt and initialize these locations with the actual address of the interrupt service routine, before the interrupts are enabled. At the address of the vector, in ROM, place a piece of code that reads the contents of the pseudovector and loads it into the program counter. The following example lists the ROM code for the timer0 interrupt:

```
           Org    0bh
Int_tm0:
           Push   76h
           Push   77h
           Ret
```

The initialization sequence must write to the RAM locations 76h–77h the address of the actual interrupt service routine for timer0.

The ‘Ixxxx’ command assumes that the entry points for the interrupt service routines are located at addresses of the form:

$$\text{ISR\_entry} = \text{start} + \text{vector}$$

Where start is the start address of the RAM program, and vector is the address of the hardwired vector associated with the respective interrupt (e.g. 0bh for timer0 interrupt, or 23h for UART events, etc.).

In fact, the commands ‘LH’ and ‘LB’ automatically set the values of the pseudovectors according to this rule, starting with the specified start address. The command ‘Ixxxx’ is useful in case of program runaway, or when several programs are loaded in RAM at different addresses, and must be tested separately.

11.4 Exercises

SX11.1

Modify the schematic of the memory section of the development board, presented in Fig. 11.2 by adding a new 32K RAM chip (62 256) located in the address range $4000-$BFFF.

Solution

The solution is presented in Fig. 11.6. Note the simple way to obtain the selection signal CS4000, which uses the fact that 7445 has open collector outputs.

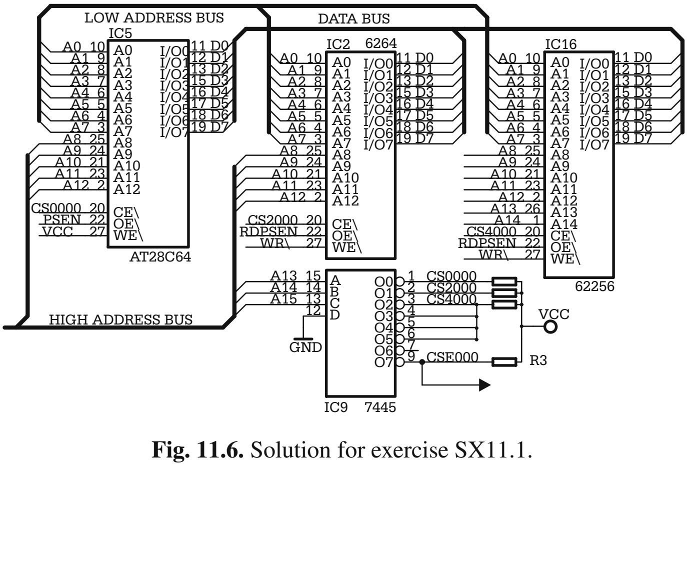


Fig. 11.6. Solution for exercise SX11.1.

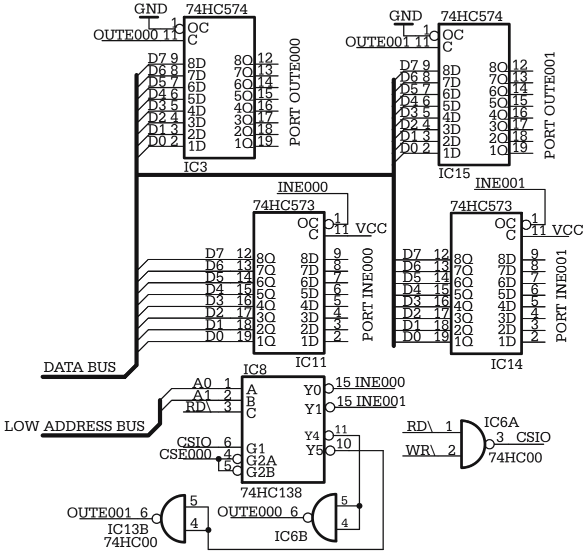


Fig. 11.7. Solution for exercise SX11.2.

SX11.2

Modify the schematic of the I/O section of the development board, presented in Fig. 11.3, by adding one more input port and one more output port, located at the address $E001.

Solution

The solution is shown in Fig. 11.7.

12

Digital Voltmeter with RS232 Interface

12.1 In this Chapter

This chapter contains a didactic example on how to use the AVR development board described in Chap. 10 as an 8-channel digital voltmeter that can be interrogated over the RS232 interface. The characters received from the serial line are interpreted as the address of the analog channel to be read and reported.

12.2 The Hardware

The voltage levels expected by the A/D converter are in the range $[0, V_{\mathrm{ref}}]$ and V_{ref} is connected to V_{cc}. The analog inputs are presented as positive voltage signals to the connector X100, pins 1–8, referred to the ground line X100-9, then filtered by the RC filters implemented by R11–R26 and C11–C19, and applied to the analog inputs ADC0–ADC7 of the MCU.

The output of the ADC system, corresponding to V_{in} is a 10-bit binary value:

$$\mathrm{ADC\_value} = 1023 \times \mathrm{V_{in}/V_{ref.}} \tag{12.1}$$

Since $V_{\mathrm{ref}} = V_{\mathrm{cc}} = +5$ V, an input voltage $V_{\mathrm{in}} = 3.1$ V, for example, is converted to decimal 634, which corresponds to the hexadecimal value $27A.

The example presented in this chapter uses only the serial communication interface RS232 to report the values read by the ADC, but optionally a local display, like the one presented in Appendix A14, can be connected to the board on the synchronous peripheral interface SPI.

12.3 The Software

The software application is structured according to the principles described in Chap. 10. After the initialization sequence, the program enters an infinite loop, where it reads all the analog inputs of the ADC, and stores the results in a RAM buffer. If an ASCII character in the range ‘0’ to ‘7’ is received on the asynchronous serial interface, the two bytes needed to represent the 10-bit value of the corresponding analog input are converted to four ASCII digits and transmitted on the serial communication line.

Here is the listing of the main module VMMAIN.ASM:

```
.include       ''8535def.inc''
.include       ''map.asm''   ;local variables
       .cseg
       .org   0
reset:
       rjmp   init
       .org   $10
init:
.include       ''init.asm''  ;calls to init routines
       sei
main_loop:
       rcall  get_adc       ;read analog inputs
       rcall  get_cmd       ;check for data from UART
       brcc   main_loop
       rcall  exec_cmd      ;answer query over
                            ;serial line
       rjmp   main_loop
.include       ''uart.asm''  ;UART specific routines
.include       ''lib.asm''   ;misc routines
.include       ''adc.asm''   ;ADC specific routines
```

Note that the main program loop only contains calls to three subroutines. Get_adc reads all the analog inputs, get_cmd checks the UART receiver data register for an ASCII character between ‘0’ and ‘7’, and returns the CARRY flag set if one of these characters has been received. Finally, exec_cmd extracts the value of the analog line indicated by the command character, converts it to four ASCII digits, and sends them to the serial line.

All other modules contain either variable definitions (MAP.ASM), calls to the initialization routines (INIT.ASM), peripheral specific routines (UART.ASM, SDC.ASM) or miscellaneous data conversion or execution routines (LIB.ASM).

MAP.ASM contains a number of definitions for constants, and RAM variables:

```
.def    rint=r1          ;some aliases for regs
.def    rsav=r2
.def    tmp1=r16
.def    tmp2=r17
.def    tmp3=r18
.def    op1l=r19
```

```
         .equ     prescaler_adc=7 ;ADC clock=CK/128
         .equ     adc_max_ch=7    ;max # of ADC channels
         .equ     f_baud=51       ;9600 baud
         .dseg                    ;RAM variables
buf_adc: .byte    16              ;buffer for ADC data
         .equ     f_baud=51       ;9600 baud
```

INIT.ASM is a block of code containing the initialization of the stack pointer SP, and calls to the initialization routines associated with the peripheral interfaces used by the application. The resource-specific initialization routines are located in the corresponding module: init_uart in UART.ASM and init_adc in ADC.ASM. The whole block of code contained in INIT.ASM is inserted in the main program at the point specified by the .include directive.

```
init_sp:
         ldi      tmp1,high(ramend)
         out      sph,tmp1
         ldi      tmp1,low(ramend)
         out      spl,tmp1
         rcall    init_uart
         rcall    init_adc
end_init:
```

The UART initialization routine, init_uart. has been described in the previous chapters. Below is the listing of the subroutine for the initialization of the ADC subsystem:

```
init_adc:
         ldi      tmp1,0             ;channel 0 first
         out      admux,tmp1
         ldi      tmp1,prescaler_adc ;ADC clock=CK/128
         out      adcsr,tmp1
         sbi      adcsr,aden         ;ADC enabled
         sbi      adcsr,adsc         ;start first
                                     ;conversion
         ret
```

This code selects channel number 0 by writing to admux, then selects the ADC clock by programming the prescaler to divide the system clock by 128, and enables the ADC system by setting the bit ADEN (A/D converter enable) in ADCSR. Finally, it starts the conversion on the selected channel, by setting the bit ADSC (Start A/D conversion) in ADCSR.

The subroutine get_adc checks the bit ADIF in ADCSR to determine if the last conversion is completed. If ADIF = 1, then the ADC result value (read as two bytes) is stored in a RAM buffer buf_adc, using the channel number as index in this buffer, then a new conversion is started on the next channel:

```
get_adc:
   sbis   adcsr,adif         ;exit if ADIF=0
   ret
   in     tmp1,admux         ;else, get channel number
   andi   tmp1,0b00000111
   ldi    xl,low(buf_adc)    ;x points to start
                             ;of buffer
   ldi    xh,high(buf_adc)
   clr    tmp2
   push   tmp1               ;save channel number
   add    tmp1,tmp1          ;two bytes foe each channel
   add    xl,tmp1            ;add offset to x
   adc    xh,tmp2
   in     tmp1,adcl          ;get ADC conversion result
   in     tmp2,adch
   st     x+,tmp1            ;and store it in buffer
   st     x,tmp2
   pop    tmp1               ;restore channel number
   inc    tmp1               ;increment
   cpi    tmp1,adc_max_ch+1 ;check if last channel
   brlo   getadc1
   clr    tmp1               ;if so, return to channel 0
getadc1:
   out    admux,tmp1         ;select new channel
   sbi    adcsr,adsc         ;start a new conversion
   ret
```

The RS232 communication is performed by means of two subroutines called in the main program loop: get_cmd and exec_cmd, both located in the module LIB.ASM.

Get_cmd checks the UART receiver data register for an ASCII code between ‘0’ and ‘7’. If no character is received, or the ASCII code is outside the expected range, get_cmd returns CARRY = 0, otherwise it returns the valid code in the register r16 (tmp1) and CARRY = 1.

```
get_cmd:
   rcall  get_uart        ;check if character received
   brcs   getcmd1         ;exit if none
   ret
getcmd1:
   clc
   cpi    tmp1,$30        ;$30 is the ASCII code for ‘0’
   brlo   exit_gcmd
   cpi    tmp1,$38        ;$38 is ASCII ‘8’
   brsh   exit_gcmd
   sec                    ;set carry
exit_gcmd:
   ret
```

Exec_cmd sends on the serial line a data packet, having the following structure:

N : AAAA CR LF

where N is the ASCII code of the channel number, ':' is used as delimiter, AAAA is the last converted value in hexadecimal of the selected analog channel, and CR, LF are the ASCII codes for 'carriage return' ($0D) and 'line feed' ($0A). This is required to allow the use of any ASCII terminal emulator, such as Windows™ HyperTerminal, to test the program.

Note that exec_cmd calls hex_asc – a subroutine that converts the content of tmp1 to two ASCII characters, returned in tmp1 and tmp2 corresponding to the nibbles of the input value.

Therefore, the string output to the terminal by exec_cmd in case of reading on channel 3 the binary value 1100100111b is: 3:0C27

Here is the listing of exec_cmd:

```
exec_cmd:
    rcall  put_uartw        ;display channel number
    rcall  asc_hex          ;convert it to hex
    ldi    xl,low(buf_adc)  ;x points to start of buf
    ldi    xh,high(buf_adc)
    clr    tmp2
    add    tmp1,tmp1        ;compute offset in buf_adc
    add    xl,tmp1
    adc    xh,tmp2
    adiw   xh:xl,2
    ldi    tmp1,':'         ;send ':' as separator
    rcall  put_uartw
    ld     tmp1,-x          ;get high byte from buffer
    rcall  hex_asc          ;convert to ascii
    rcall  put_uartw        ;send first character
    mov    tmp1,tmp2
    rcall  put_uartw        ;send second chr
    ld     tmp1,-x          ;get low byte
    rcall  hex_asc          ;convert it
    rcall  put_uartw        ;send it
    mov    tmp1,tmp2
    rcall  put_uartw
    ldi    tmp1,$0d         ;CR
    rcall  put_uartw
    ldi    tmp1,$0a         ;LF
    rcall  put_uartw
    ret
```

Note that the input for exec_cmd is the channel number in ASCII, as received from the serial line. Exec_cmd starts by echoing this value. Then, the ASCII value of the channel number is converted to its corresponding hexadecimal value by the subroutine asc_hex. The resulting binary value is used to compute the offset in the buffer buf_adc, to retrieve the result of the last conversion performed on the selected analog channel.

The accompanying CD contains a terminal emulator, Voltm.exe, which automatically interrogates the voltmeter for the values of all eight analog channels, and displays the values on the screen. The default Comm port is COM2, but this can be changed by the user. The Start/Stop button initiates and interrupts the communication process. The program runs under Windows™ 9x.

12.4 Exercises

X 12.1

Write a subroutine CBCD16 that converts a 16-bit binary number, lower than 9999, received as input in the registers tmp1:tmp2, to four BCD digits placed in the same registers.

X12.2

Modify the software so that the data sent on the serial line contains the decimal representation of the analog value read from the requested channel.

X12.3

Modify the software so that the decimal value of channel 0 is also displayed on the SPI display described in Appendix A14.

13

Simple RS485 Network with Microcontrollers

13.1 In this Chapter

This chapter contains the description of a device capable of reading the status of a number of digital inputs, or the values of a number of analog inputs, to store this information, and to report it when it receives a specific interrogation from a master device. It is implemented using an AVR microcontroller, and uses a RS485 line to communicate with the master. The aim of this chapter is to introduce the basic concepts of distributed data acquisition systems.

13.2 The Hardware

The simple data acquisition modules described in this chapter are meant to be connected as slaves in a RS485 network, as shown in Fig. 13.1.

The master device in this network is a personal computer (PC), which uses one of the asynchronous communication ports (COM1/COM2) connected to the network through a RS232-to-RS485 interface converter.

Two different slave devices are described for this application: one, called SLD, is designed to read and report the status of three digital inputs, such as relay contacts; the other, called SLA, reads the values of three analog inputs in the range 0–2.56 V.

It is possible, in principle, to design and add more slave devices, with different functions, provided that they communicate according to the same protocol as the

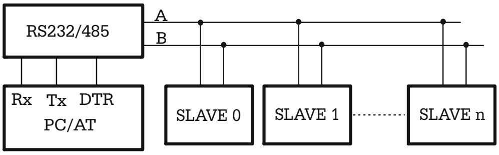


Fig. 13.1. Block diagram of the network described in this chapter

others. According to the recommendations of the RS485 standard, the total number of slaves connected to the same bus is limited to 32.

13.2.1 The RS232-to-RS485 Converter

This circuit converts the ±12 V voltage levels required by the RS232 interface of the PC used as a master device in the network, to differential signals compatible with the RS485 communication bus. The schematic of the circuit is presented in Fig. 13.2.

The actual RS485 interface circuit is IC3 –SN75176, described in Chap. 3. This converts TTL levels to differential signals, and its line driver can be controlled to enter a high-impedance status by means of the input DE (Driver Enable, active HIGH). This allows the implementation of a two-wire, half-duplex differential communication bus.

IC4 –MAX232 converts TTL to RS232 voltage levels and vice-versa. The inverting gate IC2A (74HC04) is used to provide proper polarity for DE, so that the interface signal DTR (Data Terminal Ready), available on pin 4 of the nine-pin interface connector, can be used to control the direction of data transfer on the RS485 bus.

The external power supply $V_{PP} = +12\,\mathrm{V}$ is applied on connector X1. V_{PP} is reduced to +5 V by means of the voltage regulator IC1 –LM7805, to supply local circuits, but is also made available on the connector X2, along with the differential data lines, in order to provide power to the slave devices. For this particular application V_{PP} must be able to deliver at least 200 mA.

Note the resistors R1, R2, connected to V_{CC} and GND. Their purpose is to maintain the voltage levels on the data lines A and B to a steady potential when all the line drivers of the devices connected to the RS485 bus are in the high-impedance status.

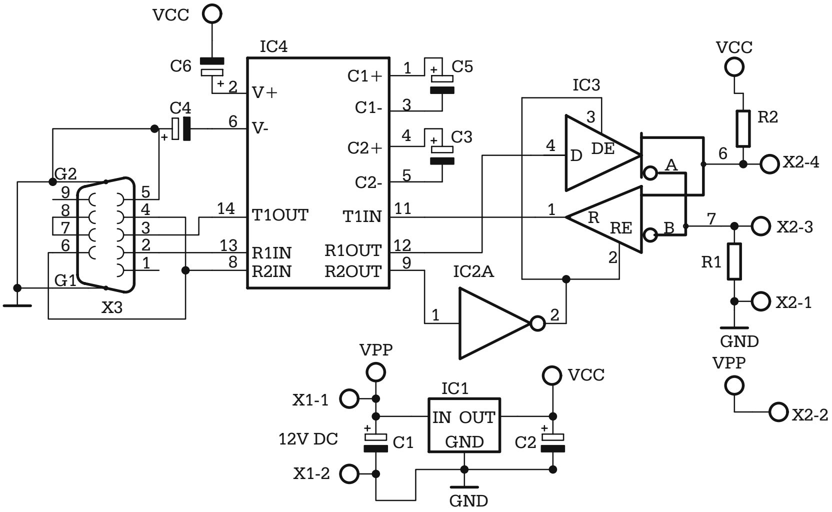


Fig. 13.2. Schematic of the RS232-to-RS485 converter

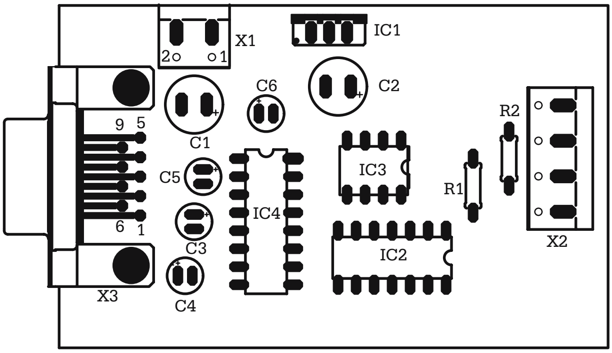


Fig. 13.3. PCB layout for the RS232-to-RS485 converter

Figure 13.3 shows a possible layout for the printed circuit board for this circuit.

13.2.2 The Digital Input Module

This circuit shown in Fig. 13.4 is one of the simplest microcontroller structures possible.

The circuit comprises the microcontroller AVR AT90S2313, with very few external components: the RS485 interface circuit IC3 (SN75176), the voltage regulator IC1 (LM7805), and the oscillator and RESET circuits.

The digital inputs are read on the PB0–PB2 I/O lines, which must be software configured to use the internal pull-up resistors, and another I/O line, PD2, enables the line drivers of the interface circuit SN75176.

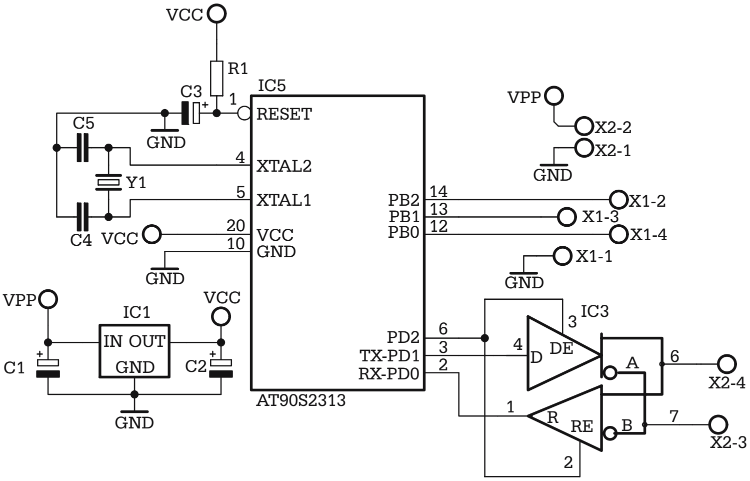


Fig. 13.4. Schematic of the SLD module

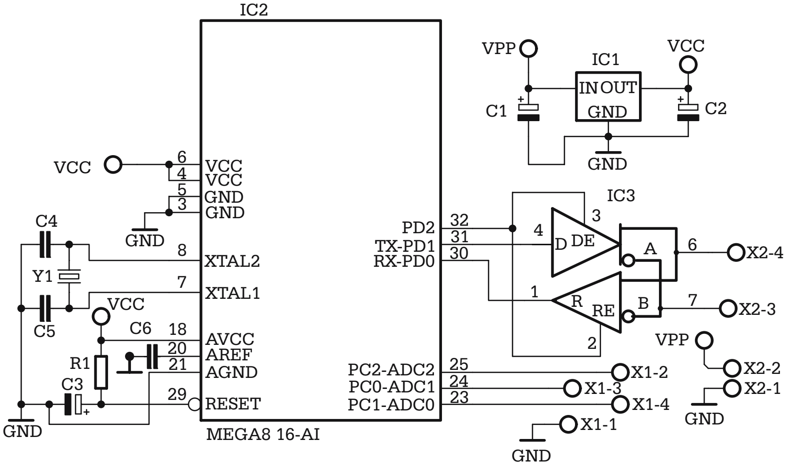


Fig. 13.5. Schematic of the analog input module SLA.

13.2.3 The Analog Input Module

The schematic of the analog input module SLA is presented in Fig. 13.5.

This circuit uses a different microcontroller, ATMEGA8-16AI (IC2), which includes an ADC converter, and internal analog reference. For better protection from communication errors the circuit uses an external oscillator (Y1, C3, C4).

The analog inputs of the MCU, ADC0–ADC2, are directly connected to the external connector X1, without additional protection or conditioning circuits. The internal reference voltage is decoupled with the external capacitor C6.

The RS485 interface is identical to the one used by the digital input module SLD, described in the previous paragraph. The data direction on the bus is controlled by the I/O line PD2 of the microcontroller, which is software controlled so that, in normal operation, the local line driver is disabled, and the receiver circuit of IC3 (SN75176) is enabled. Any slave device is allowed to take control of the communication bus only when it receives a specific query from the master.

The power supply voltage for the circuits V_{CC} is prepared using the voltage regulator IC1 (LM7805), and the filter capacitors C1, C2, starting from the external voltage V_{PP}. Normally, V_{PP} is common for all slaves.

13.2.4 Using the AVR Development Board to Emulate Thel SLD and SLA Modules

The AVR development bard described in Chap. 10 can be used to emulate the functions of the SLD and SLA modules and to test the software for the implementation of the network protocol. A simple hardware modification is required for this, as shown in Fig. 13.6.

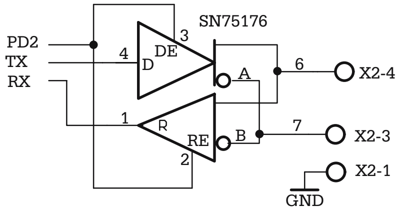


Fig. 13.6. Adding the RS485 interface circuit to the AVR development board

Table 13.1. Connections between the RS485 interface circuit and the MCU

Signal name	MCU pin	SN75176	X2
DE	16	2, 4	–
TX	15	3	–
RX	14	1	–
GND	11	5	X2-1
VCC	10	8	–
A	–	6	X2-4
B	–	7	X2-3

The purpose of the modification is to add a SN75176 circuit, which implements the RS485 interface, and the X2 connector in the custom area of the development board. These components must be connected to the MCU according to Table 13.1.

With the RS485 interface, the development board is capable of emulating all the functions of the SLD and SLA modules described above.

13.3 The Software

The principles of a simple master–slave network protocol have been described in Chap. 3. The following paragraphs contain an example of the implementation of a simple microcontroller network based on these principles.

13.3.1 Description of the Communication Protocol

The devices connected to the RS485 bus communicate according to the following set of rules:

- Communication data consists of fixed-length packets, having a predetermined structure.
- All the information in the data packets is ASCII encoded.
- All data transfers are initiated by the master device, which sends interrogation or command packets to specific slave devices.
- All slave devices are identified by a unique address.

The data packets have the following structure:

Header	Opcode	Address	Data field	CRC	EOT

Since all data transmitted is represented in ASCII, the header and end of transmission marker (EOT) can be defined by single ASCII characters, without the risk of misinterpreting other characters in the data field as false packet delimiters. For this particular example, the header is the ASCII character STX (Start of Text –$02), and the end of transmission marker is the ASCII character for EOT (End of Text –$04).

The opcode and the address fields are one character (ASCII) each. The dimension of the data field is related to the number of inputs or outputs of the slave devices referred to by the packet. In this example, the digital input module has only three inputs, and therefore three ASCII characters are enough to encode the status of all inputs.

Similarly, the 10-bit value of the result of an analog-to-digital conversion, which ranges between 0 and $3FF, can also be represented using three ASCII characters, corresponding to the three hex digits of the ADC result. This means that a three-byte data field covers the requirements of both the SLD and SLA modules.

The CRC field is, in this example, a simple checksum consisting of the two-byte ASCII representation of the sum of the bytes in the packet starting with the opcode, and ending with the last byte of the data field.

Note that this is not a real cyclic redundancy control sum, but it serves the purpose of this simple example.

With these specifications, the detailed structure of the data packets corresponding to this protocol becomes:

STX	OPC	ADR	D1	D2	D3	CRCH	CRCL	EOT

D1, D2, D3 are the three bytes of the data field, and CRCH–CRCL are the two ASCII characters corresponding to the two nibbles of the CRC checksum.

The packets sent by the master device have opcodes in the range ‘0’–‘9’ ($30–$39), as described in Table 13.2.

Table 13.2. List of valid opcodes for data packets sent by MASTER

Opcode	Command
‘0’ ($30)	Reserved
‘1’ ($31)	Reserved
‘2’ ($32)	Read digital inputs from the SLD module specified by ADR
‘3’ ($33)	Read analog input 0 from the SLA module specified by ADR
‘4’ ($34)	Read analog input 1 from the SLA module specified by ADR
‘5’ ($35)	Read analog input 2 from the SLA module specified by ADR
‘6’ ($36)	Reserved
‘7’ ($37)	Reserved
‘8’ ($38)	Reserved
‘9’ ($39)	Reserved

The packets returned by the slaves can only have the opcode ACK (Acknowledge – $06), when the command has been received and executed correctly, or NAK (Negative Acknowledge –$15), for packets received with the wrong CRC, or an invalid opcode.

Here is an example of a data packet containing a command to read digital inputs (opcode $32) from the slave module SLD having the address '1':

$02	$32	$31	$30	$30	$30	$46	$33	$04

Note that the data field of the packets sent by the master is filled with dummy '0' characters. The reason for this is to maintain the same structure for all packets, and to allow adding new commands. The CRC field is two-bytes long, $46 ('F'), $33 ('3'), which corresponds to $F3 = $32 + $31 + $30 + $30 + $30.

The answer from the slave device that reports the status of all digital inputs to logic 1 is a data packet having the opcode ACK ($06), like this:

$02	$06	$31	$31	$31	$31	$43	$41	$04

13.3.2 The Software for the SLD Module

The software is structured according to the general principles described in the previous chapters. The main module, called SLDMAIN.ASM, is listed below:

```
        .include  "8535def.inc" ;general definitions
        .include  "macro.asm"   ;macro definitions
        .include  "map.asm"     ;specific variables
        .cseg
        .org      0
reset:
        rjmp      init
init:
        .include  "init.asm"    ;all initializations
main_loop:
        rcall     get_io        ;read digital input
        rcall     send          ;send char if any
        rcall     get_cmd       ;check rec. buffer
        brcc      main_loop
        rcall     make_pk       ;prepare the answer
        rjmp      main_loop
        .include  "uart.asm"    ;uart routines
        .include  "lib.asm"     ;general routines
        .include  "io.asm"      ;i/o routines
```

MAP.ASM contains definitions of the application-specific variables and a number of equations:

```
.dseg               ;data segment
                    starts at $60
.org                $60
Buf_io:  .byte      1             ;store the status of
                                  the input
Buf_rx:  .byte      9             ;Rx buffer
Buf_tx:  .byte      9             ;Tx buffer
Stat_rx: .byte      1             ;Tx status
Stat_tx: .byte      1             ;Rx status
Ack_nak: .byte      1             ;error flag
         .def       tmp1=r16
         .def       tmp2=r17
         .def       tmp3=r18
         .def       op1l=r19
```

INIT.ASM contains the usual initialization routines for the resources involved in the application. The uart is programmed for 9600 baud, the lines 0, 1, and 2 of port C are configured as inputs, with internal pull-up resistors, and port D, bit 2 is configured as output.

The core of the application is the subroutine get_cmd, located in the module LIB.ASM. Get_cmd receives the data packets from the uart, checks its structure and CRC, and sets the carry flag and the variable Ack_nak to inform the main program about the result of the analysis. If no command is received, get_cmd returns carry = 0. Valid commands are signaled by carry = 1, and Ack_nak = ACK. Invalid commands (i.e. commands with invalid opcode, or with the wrong CRC) are indicated by carry = 1, and Ack_nak = NAK. Valid commands having different slave address are ignored (carry = 0).

Get_cmd uses two macros, aimed to improve the readability of the program, both defined in MAP.ASM.

The macro Get_adr expects two parameters: the first parameter is an 8-bit value, the second is a 16-bit value. Get_adr adds the two parameters and stores the result in register X. This is useful to create a pointer in a table, starting from the address of the first location of the table, and an 8-bit offset. Here is the definition of the Get_adr macro:

```
.macro get_adr                 ; two parameters @0 and @1
            ldi    xl,low(@1)  ; X<-@1
            ldi    xh,high(@1)
            clr    tmp2
            add    xl,@0
            adc    xh,tmp2     ;xh+0+carry !!
.endm
```

Example: `get_adr    op1l, buf_rx`

The second macro, Mk_crc, computes the CRC on a buffer indicated by the first 16-bit parameter taking into consideration a number of bytes indicated by the second parameter. Here is the definition of Mk_crc:

```
.macro             mk_crc     ;two parameters @0 ands @1
           ldi     xl,low(@0)
           ldi     xh,high(@0)
           ldi     tmp2,@1
           rcall   crc
.endm
```

Example: `mk_crc   Buf_rx, pack_len`

Get_cmd starts by calling get_uart, which returns carry = 1 if a character has been received from the serial line. The character received is available in tmp1. If the offset in buf_rx, stored in the variable stat_rx, is zero then get_cmd expects the character STX, which is the marker of the beginning of a data packet. Any other character is ignored. The characters received are stored in the buffer buf_rx, and their count is checked against the maximum length of the packet, pack_len (9). If the last character is not EOT, the whole packet is rejected.

If the packet received has the expected structure, get_cmd checks the opcode and the CRC and sets the variable Ack_nak accordingly, then exits with carry = 1. When carry = 1, the main program calls the subroutine make_pk, which prepares the transmission buffer buf_tx. The content of buf_tx is sent to the serial line, one character at a time until the counter stat_tx is decremented to zero.

See the accompanying CD for the full listing of all the program modules involved in this application.

13.3.3 The Software for the MASTER Device

The accompanying CD contains a small executable, called MASTER485.EXE, which generates and sends data packets according to the communication protocol described in this chapter. The program is able to interrogate up to four slaves, having the addresses ‘1’, ‘2’, ‘3’, and ‘4’.

The slaves with the addresses ‘1’ and ‘2’ are assumed to be digital input modules, SLD, and the slaves with the addresses ‘3’ and ‘4’ are assumed to be SLAs.

The program can operate in manual mode, when the user is expected to click on the buttons *Query1–Query4* to generate query packets for the slaves with the corresponding addresses, or automatically when the user presses the *Start* button.

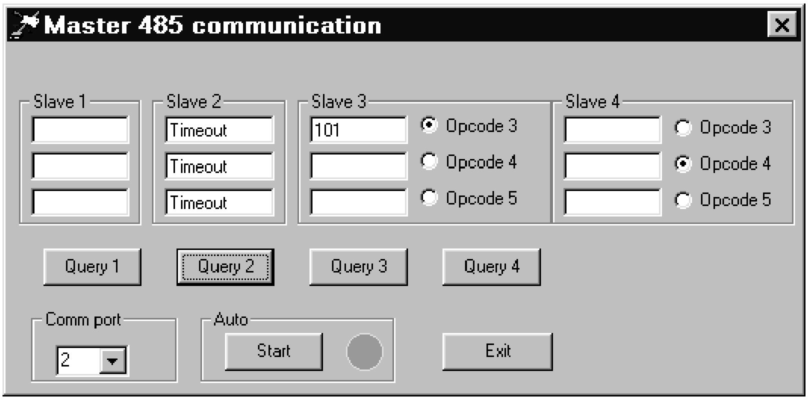
The default communication port is COM2, but the user can select any port in the range COM1–COM4. The communication parameters are 9600, N, 8, 1, and the DTR signal is used to control the RS485 line driver.

When interrogating one of the SLD modules, the opcode is automatically selected to be ‘2’. For the SLA modules, the user can select one of the opcodes ‘3’, ‘4’, or ‘5’.

The data fields are updated with the values returned by the slaves, or with one of the following error messages:

Timeout – when the slave has failed to answer within a specified time interval.
NAK – when the slave returned a NAK type packet.
CRC Error – when the packet received has the wrong CRC.

A snapshot of the main screen presented by this program is shown in Fig. 13.7.



Master 485 communication

Slave 1

Slave 2
Timeout
Timeout
Timeout

Slave 3
101
● Opcode 3
○ Opcode 4
○ Opcode 5

Slave 4
○ Opcode 3
● Opcode 4
○ Opcode 5

Comm port
2

Auto

Fig. 13.7. Snapshot for MASTER485.EXE

13.4 Exercises

X13.1

Write the software for the analog input module SLA.

X13.2

Based on the examples presented in this chapter, write a software application so that the AVR development board acts like two distinct slaves, one SLA and one SLD, having distinct addresses.

14

PI Temperature Controller

14.1 In this Chapter

This chapter is an introduction to the basic principles of control systems. It also contains a description of a didactic implementation of a PI temperature controller that uses the HC11 development board described in Chap. 9.

14.2 Basic Concepts

A control system is a system comprising physical and decisional elements, designed to control (to interferewith, to influence, to modify) a process.

In the classic example of a switch that controls an electric heater, the human operator has both decision and execution functions. A system where the intervention of a human operator is required is called *manual control*. If in this example the human operator is replaced by a time relay that switches the heater on and off at predetermined time intervals, the system becomes an automatic sequential system.

This system does not check whether the controlled heater actually produces heat, and the temperature of the environment does not influence the time relay. When the interaction between the control system and the controlled process is unidirectional, the control system is called an *open-loop control system* (see Fig. 14.1) .

Open-loop control systems are often associated with manual control. Most automatic control systems have at least one active feedback loop which allows the system to evaluate the response of the controlled process and adjust the control action, so that the value of a controlled variable is maintained close to a set-point value. The general block diagram of a closed-loop control system is shown in Fig. 14.2.

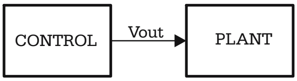


Fig. 14.1. Block diagram of an open-loop control system

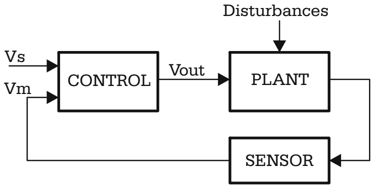


Fig. 14.2. General structure of a closed-loop control system

In this configuration, the sensor measures the value V_{m} of the process variable (which can be temperature, pressure, speed, flow, pH, etc.) and submits it to the control unit, which compares it to the desired value, or set-point value V_{s}, and adjusts the output V_{out} to reduce the error $e = V_{\mathrm{s}} - V_{\mathrm{m}}$.

Figure 14.3 describes the simplest control algorithm, where the controlled process variable is the temperature. The output of the control circuit V_{out} turns the heating element on, when the measured temperature T_{m} is lower than the set-point temperature T_{s}, and off, when $T_{\mathrm{m}} > T_{\mathrm{s}}$.

In practice, the turn-on (T1) and turn-off (T2) threshold temperatures are deliberately made to differ by a small amount (called hysteresis), to prevent switching of the heating element rapidly and unnecessarily, when the measured temperature T_{m} is close to the set-point value T_{s} (refer to Fig. 14.4). Most domestic thermostats use this control algorithm.

The problem with the on–off control systems is that the fluctuations of the controlled process variable are often too large to be acceptable. A better solution, from this point of view, is *proportional control*. In this case, the output of the control circuit V_{out} adjusts the power applied to the heater in proportion to the error signal

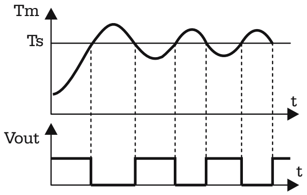


Fig. 14.3. Waveforms for the on–off control

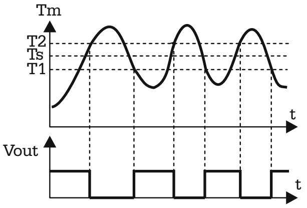


Fig. 14.4. On–Off control with hysteresis

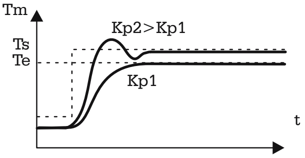


Fig. 14.5. Waveforms for proportional control

$e(t) = T_\mathrm{s} - T_\mathrm{m}$:

$$V_\mathrm{out} = K_\mathrm{p} \times e(t)\,. \tag{14.1}$$

K_p is called the *proportional gain* of the control circuit. The response of this system to a step change of the set-point temperature is shown in Fig. 14.5.

For moderate values of K_p, and in the absence of disturbances, the system reaches a steady state temperature $T_\mathrm{e} < T_\mathrm{s}$, when the energy brought into the system by the heater compensates the energy losses. The *steady state error* ($T_\mathrm{s} - T_\mathrm{e}$) can be reduced by increasing K_p, but higher values for K_p cause overshoot, or oscillations of the controlled temperature around the set-point value. In fact, for very large values of K_p, the proportional control acts like the on–off control described above. The reason for that is that the heating can only supply a limited power V_max, and it cannot sink power when $T_\mathrm{m} > T_\mathrm{s}$.

Therefore, when K_p is very large, $V_\mathrm{out} = V_\mathrm{max}$, even for small values of the error $e(t)$, as shown in Fig. 14.6. The interval of values for $e(t)$ where V_out takes values in proportion to $e(t)$ is called the *proportional band* (PB).

The expression (14.1) reflects only the status of the system *at the present time*. To solve the problem of the steady state error, it is required to adjust Vout by adding a term that reflects the evolution of the error in the past. Mathematically, this is expressed by the *integral of the error* over a period of time. The output of a *proportional-integral* (PI) control system is described by:

$$V_\mathrm{out} = K_\mathrm{p} \times e(t) + K_\mathrm{i} \times \int_0^t e(t)dt \tag{14.2}$$

The constant K_i is called the *integral gain*. Often, K_i is represented as $K_\mathrm{i} = \frac{1}{T_\mathrm{i}}$, where T_i is called *integral time* constant of the control system. The response of this system to a step change of the set-point temperature is shown in Fig. 14.7.

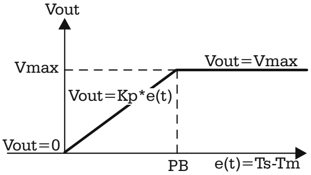


Fig. 14.6. The output of a proportional control system

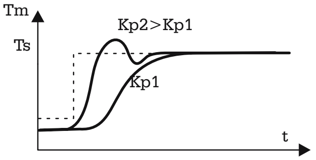


Fig. 14.7. Response of a PI control system to a step change in the error

The effect of the integral term is to adjust the output of the control circuit, until the time-averaged error is zero. Note that if the error $e(t)$ is large for a long period, for example after a large change of the set-point, or at start-up, then the value of the integral of the error becomes too large, and causes overshoot that takes a long time to recover. One method to avoid this problem (which is called *integral wind-up)* is to inhibit integral action while the error is outside the proportional band.

The integral action solves the problem of the steady state error, but not the problem of overshoot. In principle, by keeping K_p at relatively low values it is possible to avoid the overshoot, at the expense of a slow response of the system, as shown in Fig. 14.8, curve 1.

To reduce the response time, it is required to adjust V_out by adding a new term to the expression (14.2). This term should be proportional to the speed of variation of the error, $e(t)$, which is the first derivative of $e(t)$. The expression for V_out becomes:

$$V_\text{out} = K_\text{p} \times e(t) + K_\text{i} \times \int_0^t e(t)dt + K_\text{d} \times \frac{de(t)}{dt} . \tag{14.3}$$

This is the general expression of the output of a *proportional-integral-derivative* control system (PID). K_d is called the *derivative gain*, or *damping factor*. The derivative term only affects V_out when the error has a fast variation, for example when the set-point value is changed. In steady state, the value of the derivative term is zero. The effect of the derivative term on the response of the system is presented in Fig. 14.8, curve 2.

While the proportional term in the expression for V_out describes the present status of the system, and the integrative term is related to the past evolution of the system, the derivative contains information about the tendency of evolution of the system in the future. Negative values of the derivative mean that the error is decreasing, i.e. the controlled parameter tends to approach the set-point value, and therefore the control

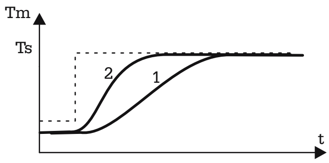


Fig. 14.8. Two possible responses to a step change in the error

action V_{out} is reduced. This explains the damping effect of the derivative component on the oscillations of the response of the system around the set-point value.

Temperature control systems are in most cases inherently slow-response systems, and therefore the derivative component is seldom useful for temperature control. Moreover, in noisy environments, some noise spikes can be misinterpreted as step changes of the error, and induce instability in the whole system. For these reasons, the example of the temperature controller described in this chapter is a PI type controller.

14.3 Hardware Implementation of a Microcontroller-Based Temperature Controller

The hardware implementation of the temperature controller uses the HC11 development board described in Chap. 9, along with an expansion board containing the circuits for interfacing the temperature sensor and the heater. The schematic of the interface with the temperature sensor is presented in Fig. 14.9.

The actual temperature sensor is an RTD (Resistance Temperature Detector), connected to J1. The RTD used in this design example has a nominal resistance of 100 ohms at 0 °C, and a rate of variation with the temperature of +0.4 ohms/°C. To sense the variation of the resistance, a constant current of 2.5 mA is supplied by the voltage-to-current converter made with the operational amplifier IC2B, the transistor Q1 and the resistor R3. The current generated by the converter is $I0 = \frac{V_{\text{REF}}-V_{\text{A}}}{R3}$, where V_{A} is a fraction of V_{REF}, adjustable by means of the potentiometer R2. V_{REF} is a 2.5 V reference voltage generated by the special circuit MAX872. IC2A is a voltage follower intended to separate the reference generator from the load. The variations of the voltage on the RTD sensor are sensed and amplified by the differential amplifier IC3, and applied to the analog input AN1 of the MCU. The gain of IC3 is 10, so that the voltage on the analog input AN1 has a variation of 10 mV/°C. By tuning the value of the reference voltage V_{REF} of the A/D converter at 2.55 V, each 10 mV quantum corresponds to 1 °C temperature; thus the value read by the A/D converter directly represents the temperature.

The circuit ICL7660 (IC5) creates the negative voltage (−5 V) required to supply the operational amplifiers.

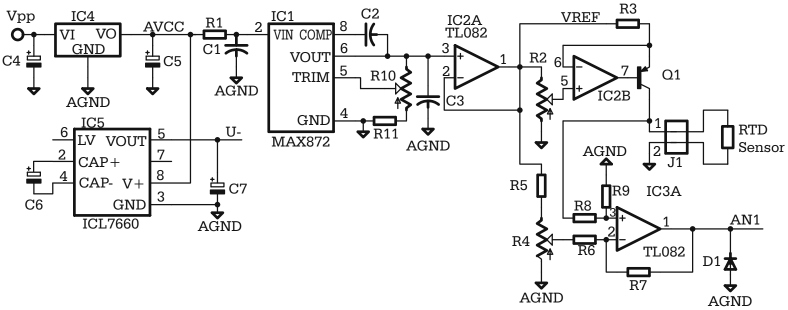


Fig. 14.9. The interface with the temperature sensor

The set-point temperature, and the control parameters K_p and T_i are read by the MCU as analog inputs, by means of three external potentiometers R12, R13, R14, that divide the reference voltage V_{REF}. The cursors of the potentiometers are connected to the AN0, AN2, and AN3 analog inputs of the MCU. (Refer to Fig. 14.10).

A rotary encoder, S1, connects to ground the four digital inputs PA1–PA4 according to the binary code of the digit selected. This information is used by software to determine which parameter is selected for display. The system uses a four-digit, seven-segment, multiplexed display, connected to the SPI, as described in Appendix A14. Two more digital inputs, PA5, PA6, are connected to external push buttons for the operator START and STOP commands.

Figure 14.11 shows the zero crossing detector circuit that generates a square wave out of the sine wave of the ac power. The output of this circuit is connected to PA0, which is also used as the input for the input capture 3 timer. The edges of this signal generate interrupt requests used by the software to synchronize the pulses that control the output triac. The same transformer that provides the ac voltage used by the zero crossing detector also provides the ac input for the bridge rectifier B1, which generates V_{pp}.

The heater is driven by the triac T2, controlled to open at precise time moments by the pulses generated by software on PA7, and transmitted to the gate of the triac by means of the transistor T1 and the pulse transformer L1, as shown in Fig. 14.12.

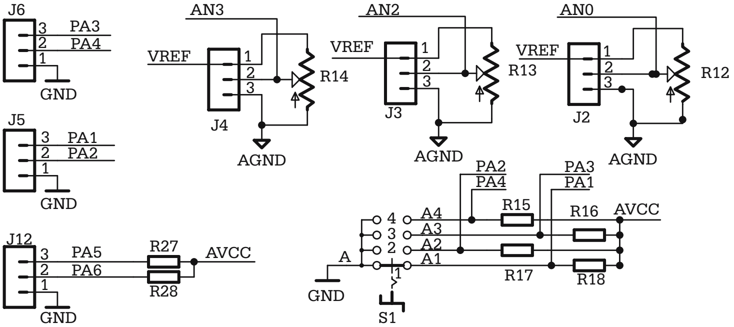


Fig. 14.10.

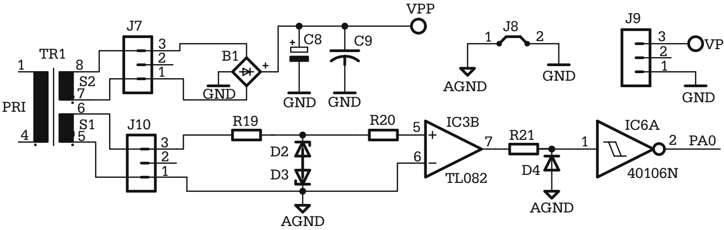


Fig. 14.11. Schematic of the zero crossing detector

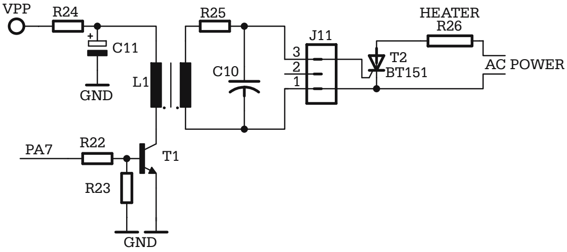


Fig. 14.12. The gate control circuit of the triac

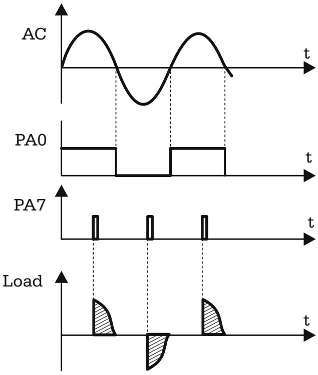


Fig. 14.13. Waveforms explaining the operation of the gate control circuit

Figure 14.13 explains how the power transferred to the heater is controlled by controlling the delay between the moment when the ac voltage crosses zero and the moment when a pulse is generated by software to open the triac.

The actual output of the control circuit is a time delay: the shorter the delay, the longer time the triac stays in conduction, and the more power is transferred to the load.

14.4 Software Implementation of a PI Temperature Controller

The structure of the software for this application follows the general structure described in Chap. 9. The MAIN.ASM module includes a number of other ASM files, each associated with a particular function, or *task*. Here is the listing of the MAIN module:

```
TITLE PI CONTROLLER MAIN
INCLUDE   68HC11F1.DEF
INCLUDE   AS11.MAC
INCLUDE   MAP.ASM
```

```
        CODE
        VECTOR_RESET              ;RESET entry point
RESET   EQU      *
        INCLUDE  INIT.ASM         ;initializations
MLOOP   EQU      *
        INCLUDE  TIMER.ASM        ;software timers
        INCLUDE  INPUT.ASM        ;digital&analog
        INCLUDE  INTEGRAL.ASM
        INCLUDE  CONTROL.ASM
        INCLUDE  DISPLAY.ASM
        JMP      MLOOP
        INCLUDE  OUTPUT.ASM
        END
```

The first two include files, 68HC11F1.DEF and AS11.MAC, are present unchanged in all HC11 applications described in this book. They contain the symbolic definitions of the I/O registers, and a set of useful macro definitions.

MAP.ASM defines the memory map, by specifying the beginning of the DATA and CODE segments, and also contains the definitions of *all* RAM variables used by the program. Although MAP.ASM contains application-specific code, a similar file must be present in all applications, along with INIT.ASM, which contains the initialization routines for all the resources used by the application. In this particular project, INIT.ASM has the following contents:

```
TITLE INITIALIZATION MODULE
CODE
SEI                     ;disable all interrupts
                        ;during init sequence
LDAA     #$0A
STAA     HPRIO          ;TIC3 has the highest
                        ;priority
LDAA     #$05           ;Enable CSPROG
STAA     CSCTL          ;for a 32k memory
LDS      #RAMEND        ;init stack pointer
LDAA     #$80           ;init I/O ports
STAA     DDRA
LDAA     #$20
STAA     DDRD
JSR      ITIMER         ;init TOC2
JSR      IADC           ;init ADC
JSR      ITIC3          ;init TIC3
JSR      ISPI           ;init SPI
JSR      INITVAR        ;init variables
CLI                     ;enable interrupts
END
```

Note that the actual initialization sequences, associated with specific resources, are organized as subroutines (ITIMER, ISPI, IADC), and are located in the soft-ware modules dealing with the respective resources.

The rest of the modules are included in an endless program loop. TIMER.ASM defines a set of software timers, as described in Chap. 9. INPUT.ASM reads the analog input lines AN0–AN3, connected to the temperature sensor and three external potentiometers, and updates the variables TS, TM, KP and TI. For scaling purposes, the 8-bit value read from AN2 is truncated to the four most significant bits, so that KP is limited to take values in the range [0–15]. Similarly, TI is reduced to take values in the range [0–63]. The digital input lines PA0–PA6 are read to update the variables SWSTATUS, which encodes the status of the rotary switch, and QSTART, a Boolean variable set to $FF when the START button is pressed, and cleared by pressing the STOP button. The subroutine CERR (Compute Error) is called here and updates the variable ERR – a 16-bit signed integer, defined as the difference (TS − TM).

The module INTEGRAL.ASM approximates the term:

$$\frac{1}{T_i} \times \int_0^{T_i} e(t)dt\,.$$

Using the notation $E_i = e(i \times T_0)$, then the integral can be approximated by the sum:

$$\sum_{i=1}^{N}(T_0 \times E_i) = T_0 \times \sum_{i=1}^{N} E_i\,. \tag{14.4}$$

If $T_i = N \times T_0$, then:

$$\frac{1}{T_i} \times \int_0^{T_i} e(t)dt = \frac{\sum\limits_{i=1}^{N} E_i}{N}\,. \tag{14.5}$$

This means that the integral term in the expression (14.3) can be approximated by the average of the last N values of the variable ERR = TS − TM. The number N of samples considered for computing the integral is determined by the variable TI. T0 is a constant, equal to 100 ms, implemented by means of a software timer.

The values of the samples E_i are stored in a buffer IBUF. The size of IBUF must be dimensioned so that it can store the maximum value of samples, as indicated by the variable TI. Since TI can have one of 64 possible values, and the error ERR is a 16-bit integer, then IBUF must be128 bytes in length.

Two additional bytes are reserved for a pointer in IBUF, named XIBUF. IBUF is organized as a circular buffer. Each time the timer that defines T_0 expires, the computed value of ERR is stored in the buffer at the location indicated by XIBUF, and then the pointer is incremented by 2. When the end of buffer is reached, the pointer is reloaded with the starting address of IBUF.

Only the last TI values in IBUF are used to compute the average. The result is stored in the variable INTEGRAL This is used by the module CONTROL.ASM, which computes VOUT = KP ∗ (ERR + INTEGRAL). VOUT is adjusted to be an 8-bit value in the range [0–255].

The computed value of VOUT is used in OUTPUT.ASM to determine the moment at wich to generate the pulse to open the triac that drives the heater. This is done in the interrupt service routine of TIC3. When an edge on IC3 (PA0) occurs, the TIC3 interrupt service routine prepares a value for TOC1, based on the value of VOUT, then enables the TOC1 interrupt.

To avoid time-consuming calculations in the interrupt service routine, VOUT is used as an offset in a table (TOCTAB) that contains for each possible value of VOUT, a 16-bit value that is added to the current TCNT and written to TOC1. These cause a TOC1 interrupt to be generated after a number of E cycles equal to the value extracted from TOCTAB.

The TOC1 interrupt service routine simply generates a pulse on PA7, then disables further TOC1 interrupts. A new interrupt will only be generated when an edge is sensed on IC3 and QSTART = $FF.

Here is what the interrupt service routines for TIC3 and TOC1 look like:

```
        VECTOR_TIC3
        LDAA     TFLG1      ;clear interrupt flag
        ORAA     #$01
        STAA     TFLG1
        LDX      #TOCTAB
        LDD      VOUT
        ABX                 ;add VOUT as offset
        LDD      0,X        ;get data from table
        ADDD     TCNT
        STD      TOC1       ;prepare TOC1 interrupt
        LDAA     #$80       ;enable TOC1 interrupts
        ANDA     QSTART     ;if QSTART=$FF
        BEQ      TIC310
        ORAA     TMSK1
        STAA     TMSK1
        RTI
TIC310  LDAA     TMSK1      ;disable TOC1 interrupts
        ANDA     #$7F       ;if QSTART=0
        STAA     TMSK1
        RTI

        VECTOR_TOC1
        LDAA     TFLG1
        ORAA     #$80       ;clear interrupt flag
        STAA     TFLG1
        LDAA     #$80       ;pulse on PA7
        STAA     PORTA
        NOP                 ;wait a few microseconds
        NOP
        NOP
        NOP
        NOP
```

```
LDAA      TMSK1       ;disable TOC1
ANDA      #$7F
STAA      TMSK1
CLRA
STAA      PORTA
RTI
```

The module DISPLAY.ASM contains the software routines needed to control the four-digit seven-segments display unit described in Appendix A14.

15

Fuzzy Logic Temperature Controller

15.1 In this Chapter

This chapter contains an introduction to the principles of fuzzy control and the description of a simple temperature controller based on these principles.

15.2 The Principles of Fuzzy Control

Tuning a PID controller, i.e. adjusting the values of K_p, T_i, and K_d, to obtain the desired response of the controlled process, can be a difficult task. The mathematical solution to this problem requires that the transfer function of the controlled process is known. In practice, real-world systems are seldom described by a simple and obvious transfer function. It would be much more convenient to define the behavior of the control system through simple sentences like this: "If the temperature is higher than the set-point and rising fast, then the output of the controller must be very low".

Fuzzy controllers work this way. Basically, a *fuzzy controller* is a control system that operates according to fuzzy logic. In fuzzy logic, the sentence "the element x is member of the set A" can be true to a degree of y%, which is equivalent to "the element x has a *y% degree of membership* to the set A". For example, when speaking about temperatures, the domain from 0 to 100 °C can be divided in two subsets: cold, and warm. An object having a temperature of 15 °C could be considered to be 60% cold and 40% warm.

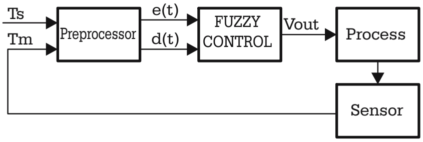


Fig. 15.1. Block diagram of a fuzzy controller

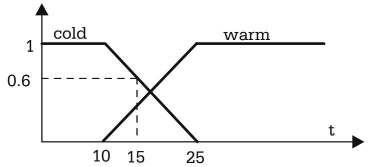


Fig. 15.2. Example of membership functions

This means that it is possible to associate to any fuzzy set A, a function $f\colon A \to [0, 1]$ called the *membership function*. Figure 15.2 shows a possible shape of the membership functions associated with the fuzzy sets "cold" and "warm" defined in the above example.

In practice, it is not very useful to know the absolute temperature of the controlled system – it is more important to know how warm the controlled system is *with respect to the set-point temperature*. In other words, it is more convenient to use the calculated error $e(t) = T_s - T_m$, instead of the measured temperature Tm.

The domain of variation of e(t) can be divided in three fuzzy subsets: a subset N for the negative values, a subset Z for the values near zero, and another subset P for positive values. Figure 15.3 shows the graphic shape of the three membership functions associated with these fuzzy subsets of values of $e(t)$.

But, as mentioned in the previous chapter, the information provided by the error function $e(t)$ describes only the present status of the system, and it is not enough for efficient control action. The information about the tendency of evolution of the system is contained in the derivative of the function $e(t)$, $d(t) = \frac{de(t)}{dt}$. $d(t)$ is also called the *error dot*. The possible values of $d(t)$ can be described as having a certain degree of membership to fuzzy subsets N_D, Z_D, P_D similar to those related to $e(t)$.

The domain of variation of the output can also be divided in three fuzzy subsets of values: Low (L), Medium (M), and High (H). To make things simpler, these subsets are *singletons,* i.e. a single numeric value is assigned to represent the whole range of low values of the output, another single value represents the medium range of values, etc. For example, if $V_{\text{out}} \in [0, 255]$, it is possible to define the singleton subsets L, M, H as $L = \{15\}$, $M = \{125\}$, and $H = \{250\}$.

Having the fuzzy subsets associated with each input and the output defined, it becomes possible to describe the behavior of the system through a number of sentences (or "rules") similar to this: "If the error is positive and the derivative of the

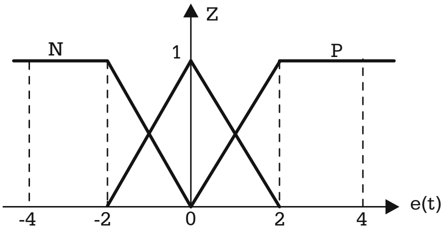


Fig. 15.3. Membership functions for the subdomains N, Z, and P of $e(t)$

error (the error dot) is positive, then the output is high.” This translates to: “If the temperature of the controlled system is lower than the set-point temperature, and the difference between the two values tends to become bigger, then the output must be high.” Which is pretty close to the natural human language.

The general structure of these rules is: “If (A and B), then C”, where A, B, and C are logic sentences referring to the inputs $e(t)$, $d(t)$ and the output $V_{\text{out}}(t)$, respectively.

(A and B) is called *the antecedent* of the proposition, and C is the *consequent*.

Note that in fuzzy logic the truth value of a proposition is not 0 or 1, but a number in the interval [0, 1]. *A, B, C and the proposition “If (A and B) then C” are all fuzzy logic propositions.*

The total number of rules that describe the behavior of the system is $K = N\,M$, where N is the number of fuzzy subsets associated with the input $e(t)$, and M is the number of subsets associated with $d(t)$. In the example presented in this chapter, $N = 3$ and $M = 3$, which leads to a total of $K = 9$ rules.

For maximum clarity, the nine rules can be presented in two tables (Tables 15.1 and 15.2): one containing the values for the antecedent, the other for consequent.

The antecedent table and the consequent table define the *rule matrix*. The whole system of rules defines the *rule base*.

Knowing the membership functions for $x = e(t)$ and $y = d(t)$, it is possible to determine at any moment the truth value of the sentences A_i and B_i corresponding to the antecedent of each rule. In fuzzy logic, if (A_i), (B_i) are the truth values of the sentences A_i, B_i, then the truth value of the sentence A_i and B_i is $z_i = (A_i \text{ and } B_i) = \min((A_i), (B_i))$.

One possible way to obtain the *crisp output* V_{out} is by combining all the rules in the rule matrix, according to the following formula:

$$V_{\text{out}} = \frac{\sum_{i=1}^{K} zi \times Si}{\sum_{i=1}^{K} zi} \tag{15.1}$$

Table 15.1. The antecedent table

$x = e(t)$ $y = de(t)/dt$	N	Z	P
N	$x = N$ and $y = N$	$x = Z$ and $y = N$	$x = P$ and $y = N$
Z	$x = N$ and $y = Z$	$x = Z$ and $y = Z$	$x = P$ and $y = Z$
P	$x = N$ and $y = P$	$x = Z$ and $y = P$	$x = P$ and $y = P$

Table 15.2. The consequent table

$z = V_{\text{out}}(t)$	N	Z	P
N	L	L	M
Z	L	L	M
P	L	M	H

Where $z_i = \min((A_i), (B_i))$, S_i is the corresponding singleton value of the fuzzy output, and K is the total number of rules in the rule base.

The process of computing the crisp value of the output of the fuzzy controller is called *defuzzyfication*.

Example

Consider a fuzzy temperature controller having the membership functions for $e(t) = T_s - T_m$ defined in Fig. 15.4, and for $d(t) = de(t)/dt$ defined in Fig. 15.5. The singleton values of the output are Low = 10, Medium = 125, and High = 250. Compute the crisp output for a time moment when $e(t) = 3$ °C, and $d(t) = 1$ °C/s.

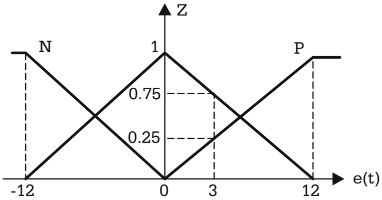


Fig. 15.4. Sample membership functions for $e(t)$

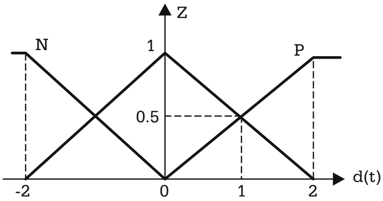


Fig. 15.5. Sample membership functions for $d(t)$

Solution

The first step in solving this problem is to create the antecedent table, starting from the data provided in Figs. 15.4 and 15.5 (see Table 15.3). In this table, the notation $x = N$ indicates the truth value of the position "$x = N$" ("x is negative").

Each cell of Table 15.3 contains two numeric values, the first of which represents the degree of membership of $e(t)$ to the sets N, Z, and P (extracted from Fig. 15.4), and the second represents the degree of membership of $d(t)$ to the sets N, Z, and P (as results from Fig. 15.5).

By selecting the minimum of the two values, the antecedent table becomes Table 15.4 and, after filling the values of singletons for each situation, the consequent table becomes Table 15.5.

Table 15.3. Sample antecedent table

$x = e(t)$ $y = de(t)/dt$	N	Z	P
N	$(x = N) = 0$ $(y = N) = 0$	$(x = Z) = 0.75$ $(y = N) = 0$	$(x = P) = 0.25$ $(y = N) = 0$
Z	$(x = N) = 0$ $(y = Z) = 0.5$	$(x = Z) = 0.75$ $(y = Z) = 0.5$	$(x = P) = 0.25$ $(y = Z) = 0.5$
P	$(x = N) = 0$ $(y = P) = 0.5$	$(x = Z) = 0.75$ $(y = P) = 0.5$	$(x = P) = 0.25$ $(y = P) = 0.5$

Table 15.4. Sample antecedent table

$x = e(t)$ $y = de(t)/dt$	N	Z	P
N	0	0	0
Z	0	0.5	0.25
P	0	0.5	0.25

Table 15.5. Sample consequent table

$z = Vout(t)$	N	Z	P
N	10	10	125
Z	10	10	125
P	10	125	250

It is now possible to compute the crisp value of V_{out}, according to the formula (15.1), which is a weighted average of the values in the consequent table, with the weights extracted from the corresponding cells of the antecedent table.

$$V_{\text{out}} = \frac{0.5 \times 10 + 0.25 \times 125 + 0.5 \times 125 + 0.25 \times 250}{0.5 + 0.25 + 0.5 + 0.75} = 107\,.$$

15.3 A Microcontroller Implementation of a Fuzzy Controller

From a hardware perspective, nothing distinguishes a fuzzy controller from a microcontroller-based PID controller. It’s the software that makes the difference. The project described in this chapter uses exactly the same hardware as that used for the PI temperature controller described in the previous chapter.

The key element in designing the software for this application is the way of representing the membership functions, The simplest solution to do this is to use the values of $e(t)$ and $d(t)$ as the offset in ROM tables associated with every function. In this case, obtaining the degree of membership for any value of the variable ERR $= e(t)$ is as simple as this:

```
* HC11 code
       LDX    #MFTAB   ;load X with the starting address
       LDAB   ERR      ;load current value of e(t) in B
       ABX             ;adjust index
       LDAA   0,X      ;get the value from table
       ....
       ....
MFTAB  DB     $00
       DB     $01
       ....
```

This fragment of code assumes that ERR is an 8-bit unsigned integer. In practice, things are a bit more complicated because $e(t)$ and $d(t)$ can take values in the range $[-255, 255]$. To cover this range, the variables associated with $e(t)$ and $d(t)$, called ERR and DERIV, must be 16-bit signed integers.

To avoid working with negative offsets in a table, a simple solution is to split the table associated with a membership function in two distinct tables: one for the positive values of the variable, the other for the negative values. Here is an example of code that uses this artifice:

```
       LDD   ERR     ;ERR is now a 16-bit signed INT
       BMI   NEGVAL  ;check sign of D
       LDX   #MFTAB1 ;one table for positive values
       ABX           ;id D>=0, B contains the offset
       LDAA  0,X     ;get value from table
       ....
NEGVAL LDX   #MFTAB2 ;table for negative values
       COMA          ;compute 2's complement
       COMB          ;of the negative value in D
       ADDD  #1      ;B contains now abs(ERR)
       ABX
       LDAA  0,X     ;get the value from table
       ....
MFTAB1 DB    $00     ;table for positive values
       DB    $01
       ....
       ....
MFTAB2 DB    $00     ;table for negative values
       ....
       ....
```

Besides the obvious simplicity of the program, defining membership functions as tables has another important advantage: it allows the use of any shape for the membership functions without the need to change anything in the code, only the tables.

The general structure of the software application, defined in MAIN.ASM, is very similar to the implementation of the PI temperature controller, described in the previous chapter.

```
       INCLUDE   68HC11F1.DEF
       INCLUDE   AS11.MAC
       INCLUDE   MAP.ASM
       CODE
       VECTOR_RESET
RESET  EQU       *
       INCLUDE   INIT.ASM
```

```
MLOOP  EQU       *
       INCLUDE   TIMER.ASM
       INCLUDE   INPUT.ASM
       INCLUDE   DERIV.ASM
       INCLUDE   FUZZY.ASM
       INCLUDE   DISPLAY.ASM
       JMP       MLOOP
       INCLUDE   OUTPUT.ASM
       INCLUDE   MFTAB.ASM
       END
```

INPUT.ASM contains the code to read the status of the digital input lines (the START and STOP buttons, and the rotary encoder used to select the variable to display) and the values of the analog inputs AN0, AN1, AN2, assigned to the variables TS – set-point temperature, TM – measured temperature, and TD – the time interval used to compute the error dot. The variable ERR is updated here with the values of the difference TS – TM by the subroutine CERR, listed below:

```
CERR   CLRA
       LDAB  TM
       STD   TEMP     ;TEMP is a 16-bit storage
       LDAB  TS
       SUBD  TEMP     ;D=TS-TM 16-bit signed INT
       STD   ERR      ;update ERR
       RTS
```

The rest of the preprocessing task is performed in the module DERIV.ASM, which calculates the derivative of the error, by comparing the current value of ERR with the previous value, stored in the variable PREVERR. The time interval between two successive samples is obtained using a software timer, and it is made adjustable by multiplying the timer quantum with the value of the variable TD, which stores the analog value of the potential on the cursor of the potentiometer R13. The effect is that the variable DERIV = ERR – PREVERR is updated at time intervals of 100 ms*TD.

The six membership functions associated with the subdomains *N*, *Z*, and *P* of *e*(*t*) and *d*(*t*) are defined as tables in module MFTAB.ASM. This also contains the nine-byte consequent table SGTAB.

The actual fuzzy inference is performed in FUZZY.ASM, by the subroutine CAT (Create Antecedent Table), which updates the contents of the 9-byte RAM table ATAB, according to the rule matrix described above. Finally, the subroutine CRISP computes the crisp output of the controller, and updates the variable VOUT, as the weighted average of the values in SGTAB, with the weights extracted from the antecedent table ATAB.

The value of VOUT is used in OUTPUT.ASM to determine the moments when the triac that drives the heater must be open, as described in the previous chapter.

Exercises

SX 15.1

What is the influence of the parameter TD on the overall behavior of the fuzzy controller described above?

Solution

TD determines the time between the moments when the variables ERR and PREVERR are updated. The values of the error dot $d(t) = e(t_2) - e(t_1) = e(t_1 + TD \times k) - e(t_1)$ are directly related to the parameter TD and to the response time of the controlled system. When TD is too small, t_2 is very close to t_1, $d(t)$ tends to zero, and the process is misinterpreted as being at equilibrium. Therefore, the control system exerts low-to-moderate action on the process, even if the error is positive and high.

On the contrary, when TD is too large, the controller tends to react drastically even at small variations of the temperature, and the system tends to become unstable. In practice, TD should be set to high values only for slow response processes.

SX 15.2

What is the effect of changing the shape of the membership functions associated with the subdomains N, Z, P of $e(t)$ to the shape presented in Fig. 15.6?

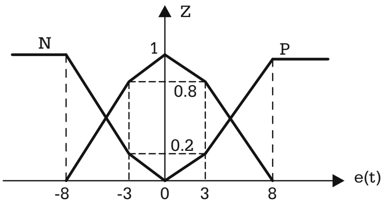


Fig. 15.6. Membership functions for exercise SX15.2

Solution

The slope of a function $f(x)$ contains information about the speed of variation of the function. For high values of the slope, even small variations of the argument x produce important variations in $f(x)$. In this particular example, the controller senses the variations of $e(t)$ as being more significant when the slope of the membership functions is higher, and therefore V_{out} will change more drastically when T_m is far from T_s, than in the vicinity of the equilibrium point.

Note the similarity between the slope of the membership functions associated with $e(t)$ and the proportional gain K_p of a PID controller. Using membership functions with the shape presented in Fig. 15.2 is equivalent to dynamically adjusting the parameter K_p of a PID controller. The risk of overshoot is reduced in this case, compared to the case of triangular membership functions.

16

Remote Relay Controller over Telephone Lines

16.1 In this Chapter

This chapter contains the description of a device that interfaces a microcontroller to a telephone line, and is capable of receiving and decoding a series of DTMF signals, and to execute commands received this way.

16.2 Description of the Hardware Solution

Since telephone lines are available almost anywhere, it is interesting to design a device that, when connected to a phone line, is able to answer a call, and to recognize and execute a series of commands generated directly from the keypad of the telephone set that initiated the call.

This device operates according to the following set of rules:

- The human operator initiates a call, using a regular telephone set.
- The relay controller senses and counts the ring tones generated by the central office.
- After a specified number of rings, the relay controller answers the call and transmits a specific audible tone over the telephone line to acknowledge its presence.
- Using the telephone keypad, the operator generates a series of DTMF (Dual Tone Multi Frequency) tones, having a determined structure, which are received and decoded by the controller. The DTMF tones are audio signals, consisting of a combination of two sine waves, having precise frequencies in the range 697–1633 Hz. Each key of the telephone set keypad is associated with a distinct combination of frequencies, so that the receiving device can easily identify the key pressed.
- The relay controller sends back to the operator distinct audible tones after the reception of correct, or erroneous DTMF series.
- After 30 seconds of inactivity, i.e. if no DTMF tone is received for 30 seconds, the controller closes the line, and terminates the session.

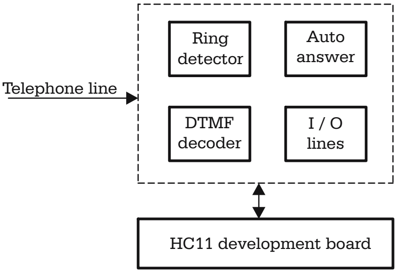


Fig. 16.1. Block diagram of the remote relay controller

The block diagram of such a device, called *remote relay controller*, is presented in Fig. 16.1.

The circuit is designed to use the HC11 development board, described in Chap. 9, extended with a telephone line interface, and the actual relay interface.

The telephone line interface comprises the following functional blocks:

- The ring detector. This is the circuit that recognizes the ring tone, and informs the microcontroller about an incoming call.
- The auto-answer circuit. This allows the microcontroller to open the line and answer the call, after a specified number of rings.
- The DTMF decoder. This decodes DTMF tones received over the telephone line and reports the codes to the microcontroller, along with a STROBE signal. The DTMF tones are generated by the telephone set that initiates the call.

The schematics of the ring detector and auto-answer circuits are presented in Fig. 16.2.

The device emulates the behavior of a standard telephone set. While in the on-hook status, the capacitor C1 separates the interface from the telephone line, but allows the ring signal, which is an ac signal with amplitude 48–50 V, and frequency of approximately 30 Hz, to go through, towards the rectifier formed by D1, D2 and C2.

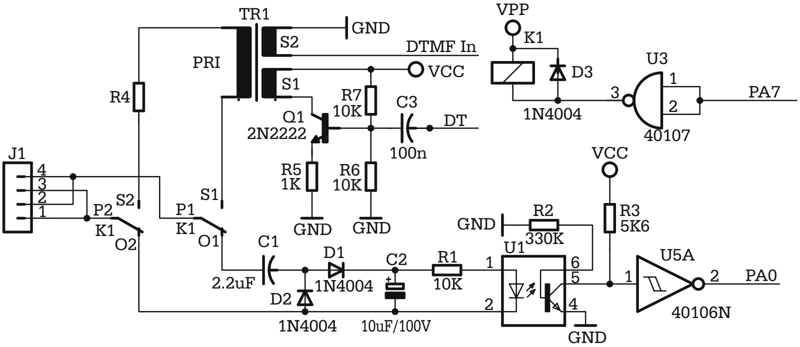


Fig. 16.2. Ring detector and auto-answer circuits

The resulting dc voltage activates the LED of the optocoupler U1, which results in the saturation of the output transistor. The output of the optocoupler is conditioned and inverted by the Schmitt trigger gate U5A, and then applied to the PA0 (TIC3) input of the microcontroller. The rising edge of the signal on PA0 generates a TIC3 interrupt, thus informing the microcontroller about the detected ring.

After counting a programmed number of rings, the MCU activates the relay K1, by writing 1 to PA7, and opens the telephone line. In the off-hook status, the telephone line is connected to the primary circuit of the transformer TR1, in series with the resistor R4 (560 ohms), which gives a total load impedance of around 600 ohms, as required by all modern central offices.

Besides galvanic isolation, the transformer TR1 has two additional functions: it transfers the audio signal DTMF_IN from the telephone line to the input of the DTMF decoder, and, on the other hand, transmits the audible dialog tones DT, generated by the interface, to the telephone line, by means of the transistor Q1.

The actual DTMF decoder is shown in Fig. 16.3.

The input for the decoder circuit CM8870 (U2) is the analog signal DTMF_IN, which is the analog signal present on the telephone line, separated by the transformer TR1, without further processing. When this signal contains a valid combination of tones, corresponding to the DTMF standard, the circuit presents on the output lines Q1–Q4 the 4-bit binary code of the digit associated with the DTMF tone, along with a strobe signal STD, active HIGH. STD stays HIGH as long as the DTMF tone is stable at the input of the circuit (min 100 ms).

The combinations of frequencies associated with each digit are presented in Table 16.1, and the corresponding binary codes are shown in Table 16.2.

The strobe signal is connected to the line PA1/IC2 of the MCU, and generates a TIC2 interrupt on the rising edge. The data bits Q1–Q4 (Q4 is the most significant bit) are connected to the port D (PD2–PD5) of the microcontroller.

Note that the binary codes associated with each digit are not the hexadecimal codes for these digits. Table 16.2 lists the binary codes corresponding to the frequency pairs in Table 16.1.

Figure 16.4 shows the schematic of the circuit that generates the audible dialog tones DT, transmitted by the controller over the telephone line in response to

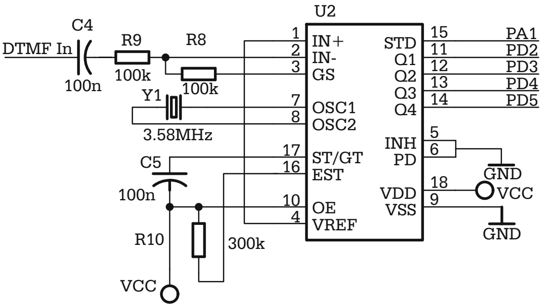


Fig. 16.3. Schematic of the DTMF decoder circuit

Table 16.1. DTMF combinations of frequencies for each digit

	Digit				Low frequency (Hz)
Digit	1	2	3	A	697
	4	5	6	B	770
	7	8	9	C	852
	*	0	#	D	941
High frequency (Hz)	1209	1336	1477	1633	

Table 16.2. The binary codes associated with the digits of the telephone set keypad

Digit	Q4	Q3	Q2	Q1
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1
0	1	0	1	0
.*	1	0	1	1
#	1	1	0	0
A	1	1	0	1
B	1	1	1	0
C	1	1	1	1
D	0	0	0	0

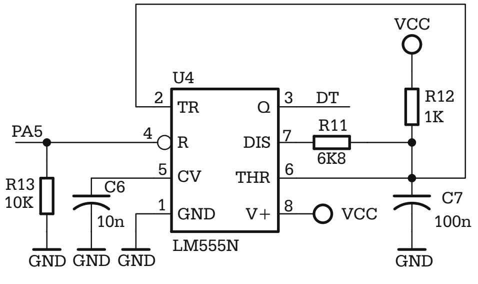


Fig. 16.4. Schematic of the multivibrator that generates the dialog tones

the DTMF commands received. This is a classic multivibrator, built with NE555, calibrated to generate a square wave signal with frequency around 1 kHz.

The circuit is controlled by the output line PA5 of the microcontroller. When PA5 = 1 the multivibrator is enabled. The duration of the audible beep transmitted is controlled by software by controlling PA5. One long beep (1 s) is used to indicate that the last DTMF command is invalid, while three short beeps (0.3 s) indicate a valid command.

16.3 Description of the Software

The structure of the software application, as defined in MAIN.ASM, is entirely similar to the structure of the other HC11 applications described in this book:

```
             INCLUDE    68HC11F1.DEF
             INCLUDE    AS11.MAC
             INCLUDE    MAP.ASM
             CODE
             VECTOR_RESET
RESET        EQU        *
             INCLUDE    INIT.ASM
MLOOP        EQU        *
             INCLUDE    TIMER.ASM
             INCLUDE    RING.ASM
             INCLUDE    DTMF.ASM
             INCLUDE    BEEP.ASM
             JMP        MLOOP
             END
```

The application-specific modules are RING.ASM, which controls the ring detector and the auto-answer circuits, DTMF.ASM, containing the interrupt service routine that handles the data from CM8870, and executes the commands, and BEEP.ASM, which contains the routines for controlling the generation of the dialog tones.

The central element of RING.ASM is the interrupt service routine for TIC3, called on the rising edge of the output of the ring detector circuit. This increments the variable RCNT (Ring Counter) and sets the variable QRING to $FF, to inform the main program that a ring has been detected.

The logic diagram of the task RING.ASM is presented in Fig. 16.5. Besides the variable RCNT, which is compared to the EEPROM constant MAXRING to determine the moment when the controller must open the telephone line, RING.ASM uses two software timers.

The first timer controls the time interval between two successive rings detected. If this interval is longer than 3 seconds, the detection sequence is aborted and RCNT is cleared. The other timer provides a software mechanism to terminate the connection and close the line when no DTMF tones are detected for more than 30 seconds.

The interesting aspect of this piece of software is that it is implemented as a finite state machine. The ring detection and auto-answer machine has three distinct states, encoded by the variable RSTATUS.

State 0 is the idle state, when no ring has been detected yet, and the line is closed. State 1 corresponds to the situation when $1 < \text{RCNT} < \text{MAXRING}$, and state 2 is the open line state, when the controller receives and executes DTMF commands.

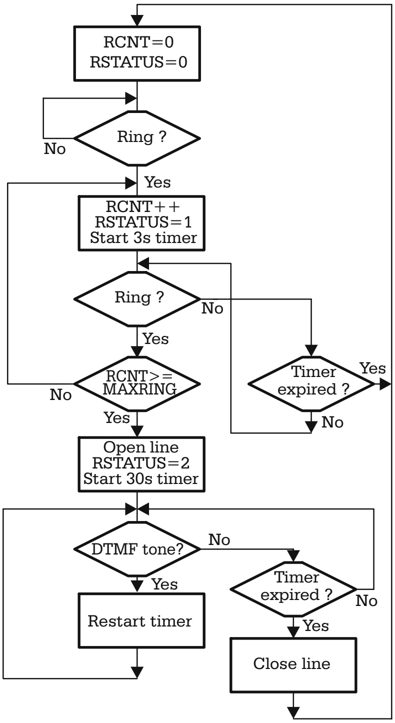


Fig. 16.5. Logic diagram of the ring counter and auto-answer program

Each time the program enters this task, it is directed to a different section, according to the value of the variable RSTATUS. The value of RSTATUS is used as an offset in a table containing the starting addresses of the routines associated with the corresponding states:

```
            LDX    #JTAB      ;X points to jump table
            LDAB   RSTATUS
            LSLB              ;multiply by 2 !
            ABX               ;adjust X
            LDX    0,X        ;get the address to jump
                              to
            JMP    0,X        ;and jump there
            ...
            ...
RNG00       ...               ;jump here when RSTATUS=0
            ...
RNG10       ...               ;jump here when RSTATUS=1
            ...
RNG20       ...               ;jump here when RSTATUS=2
            ...
```

```
JTAB        DW     RNG00        ;2 bytes for each address
            DW     RNG10
            DW     RNG20
```

The transitions from one state to another occur when global variables, controlled by the interrupt service routines or by other tasks, change their values. Here is an example showing how this particular state machine changes state from 0 to 1:

```
RNG00       TST    QRING        ;QRING is set by TIC3 ISR
            BEQ    END_RING     ;if no ring, continue
                                ;to next task
            LDAA   #$01
            STAA   RSTATUS      ;prepare transition
                                ;to next state
            LDAA   #3
            STAA   T1S0         ;start 3 sec. timer
            CLR    QRING        ;clear QRING
                                ;after using it
            JMP    END_RING     ;execute the rest of tasks
                                ;before entering state1
```

This simple technique can be used to implement pretty complex finite state machines, with dozens of states.

The module DTMF.ASM handles data provided by the interrupt service routine for TIC2, the interrupt generated by the data strobe of CM8870. This routine places the DTMF data in a four-byte buffer DTBUF. When the end of buffer is reached, or when the binary code associated with the key ‘#’ is detected, the flag DBFULL is set to true to inform the program that the data buffer is full, and ready to be interpreted.

The following commands are accepted:

- *10# – Change relay status to OFF
- *11# – Change relay status to ON
- *2*x*# – Change the number of rings before the call is answered to the value *x*, ranging from 2 to 9. This parameter is stored in the EEPROM location MAXRING.
- *30# – Close the line and terminate the session.

The content of the buffer is checked against this structure, and if a valid command is recognized, it is executed immediately, and three short beeps are transmitted to the telephone line to acknowledge the command. One long beep indicates invalid data detected in the buffer.

Here is the listing of the interrupt service routine for TIC2:

```
            VECTOR_TIC2
            LDAA   TFLG1
            ORAA   #$02
            STAA   TFLG1        ;clear interrupt flag
            LDAA   #30
            STAA   T1S1         ;restart off-hook timer
            LDAA   PORTD        ;get DTMF code
            LSRA                ;shift to the lower nibble
            LSRA
            CMPA    #$0A        ;$0A is the code for '0'
            BNE     TIC210
            CLRA                ;change it to 0
TIC210      LDX     XDTBUF      ;get pointer in DTBUF
            STAA    0,X         ;put data in buffer
            INX                 ;update pointer
            CPX     #ENDDBUF    ;check for end of buffer
            BHS     TIC220
            CMPA    #$0C        ;DTMF code for '#'
            BEQ     TIC220
            RTI
TIC220      LDX     #DTBUF      ;point to start of buffer
            STX     XDTBUF
            LDAA    #$FF        ;true the flag DBFULL
            STAA    DBFULL
            RTI
```

The information prepared in DTBUF by the interrupt service routine is interpreted in the task DTMF.ASM. The following code fragment shows how the relay is controlled according to the contents of the reception buffer:

```
            TST     DBFULL      ;if buffer not full,
                                ;exit task
            JEQ     END_DTMF
            LDX     #DTBUF
            LDAA    0,X
            CMPA    #$0B        ;check if first code
                                ;is '*'
            BNE     DT99        ;error if not
            LDAA    1,X         ;check next code
            CMPA    #$01        ;check all valid opcodes
            BEQ     DT10
            CMPA    #$02
            BEQ     DT20
            CMPA    #$03
            BEQ     DT30
            BRA     DT99        ;any other value
                                ;is invalid
DT10        LDAA    2,X         ;relay ON/OFF command
            BEQ     RELOFF
            CMPA    #$01
            BEQ     RELON
            BRA     DT99        ;any other value is
                                invalid
RELON       LDAA    PORTA       ;relay ON
            ORAA    #$40
            STAA    PORTA
            BRA     DT90        ;generate dialog tone
```

```
RELOFF      LDAA   PORTA      ;relay OFF
            ANDA   #$BF
            STAA   PORTA
            BRA    DT90       ;generate dialog tone
```

The module BEEP.ASM that controls the generation of the dialog tones is structured as a finite state machine, in a way similar to the module RING.ASM. This time, the variable that directs the program to the subsections dedicated to each state is called BSTATUS.

There are two predefined sequences of states, associated with the two types of dialog tones required. The program transits from one state to another when a timer expires, and, after going through all the states in sequence, it returns to the idle state 0.

To start a sequence, another active task must write a value in BSTATUS, according to the initial state in the sequence. In this particular case, the initial states for the two sequences are defined as follows:

- BSTATUS = 1 starts the sequence that generates one long beep (1 sec).
- BSTATUS = 3 start the sequence that generates three 0.3-seconds beeps, separated by a pause of 0.1 seconds.
  DTMF.ASM writes directly to BSTATUS to generate the two possible dialog tones:

```
DT90        LDAA   #$03       ;valid command executed
            STAA   BSTATUS
DT91        LDX    #DTBUF     ;reset pointer to DTBUF
            STX    XDTBUF
            CLR    DBFULL
            BRA    END_DTMF
DT99        LDAA   #$01       ;invalid command
            STAA   BSTATUS
            BRA    DT91
```

Appendices

A.1 Glossary of Terms

ADC

Acronym for Analog-to-Digital Converter. A circuit that converts the amplitude of an analog signal into a numeric value, presented as a binary number.
See also: Resolution, S/H, SAR

Address

An unique code that identifies a resource in a microcontroller system.
See also: Address bus, Address space

Address bus

Physical connection between the CPU and the resources accessed, carrying the address information. The number of lines of the address bus determines the maximum number of resources that can be accessed.
See also: Address, Address space

Address space

The range of addresses associated with a resource, or a group of resources.
See also: Address, Address bus

Architecture

In a computer system, this term designates the structure and organization of the resources which determines the way resources are accessed by the CPU.
See also: Harvard, Von Neumann data memory, Program memory, Address bus, Data bus

ASCII

Acronym for American Standard Code for Information Interchange. This is a 7-bit binary code created to facilitate data communication between computers and peripheral equipment created by different manufacturers.

Asynchronous communication

Communication technique that allows data exchange without the need of a common synchronization clock for the transmitter and receiver(s).
See also: SCI, UART, Baud rate

Baud rate

Data transmission rate in serial communications, measured in bits per second.
See also: Asynchronous communication, SCI, UART

Buffer

Hardware: A device intended to adapt the electrical characteristics of a MCU I/O line to a load, or signal source.
Software: Temporary data storage intended to compensate the differences in communication rates between processes or devices involved in a computer system.

Bus

Physical connection between the CPU and the other resources of a computer system that carries data or addresses.
See also: Address bus, Data bus, Architecture, Harvard, Von Neumann

CPU

Acronym for Central Processing Unit. The core of any computer system that does all data processing, and controls the access to the data and address buses of the system. The CPU includes the instruction decoder, and an arithmetic and logic unit (ALU).
See also: Architecture, Data bus, Address bus, Peripheral interface, Instruction, Program

CRC

Acronym for Cyclic Redundancy Check. A special checksum included in data packets for error detection.
See also: Asynchronous communication, Parity bit

Current loop

Older technique of encoding information on serial communication lines. Typical values for the switched currents are 20 mA for an “IDLE,” “MARK,” or “1” condition and 0 mA for a “START,” “SPACE,” or “0” condition. This type of connection has high noise immunity and built-in ground loop isolation. The disadvantage is that the data rate is usually limited to less than 19.2 kbps.
See also: Asynchronous communication, RS232, RS422, RS485

DAC

Acronym for Digital-to-Analog Converter. A circuit that generates an analog signal having amplitude proportional with the value of a binary number received as input.
See also: ADC, Resolution

Data Bus

The bus (physical connection between the CPU and other resources of a computer system) that carries data. Note that some computer system can have more than one data buses.
See also: Bus, Address bus, Architecture, Harvard, Von Neumann

Data direction register

A register, associated with an input/output port, used to configure each individual line of the port as an input or output line.
See also: I/O line, I/O port

Data memory

Read/write memory area used to store variables in a computer system.
See also: Architecture, Memory, RAM, ROM

Differential communication

Technique of communication based on the use of two conductors to transmit each digital signal, the logic levels being defined by the relative difference of potential between the two conductors. The line driver of the transmitter controls the voltage levels on the two conductors, so that the difference of potential between them is positive for logic 1 and negative for logic 0.
See also: Current loop, RS232, RS422, RS485

EEPROM

Acronym for Electrically Erasable Programmable Read Only Memory. A special type of non-volatile memory that can be erased by exposing it, in certain conditions, to an electrical voltage higher than the normal operating voltage.
See also: RAM, ROM, Flash memory

Flash memory

Type of erasable non-volatile memory, very similar to EEPROM. The main difference between the two types is that EEPROM must be written one byte at a time, while the flash memory allows writing of blocks of data, which leads to faster write operations.
See also: RAM, ROM, EEPROM

Framing error

Error condition, specific for asynchronous serial communications, occurring when an invalid *stop bit* is detected.
See also: Asynchronous communication, Start bit, Stop bit.

Harvard

Computer architecture characterized by the presence of different buses for program memory and data memory. This leads to higher processing speed.
See also: Von Neumann

I2C bus

Acronym for Inter IC bus. This type of bus consists of two signal conductors named SDA (Serial Data) and SCL (Serial Clock) that interconnect at least two devices.
See also: Bus

Intel hex format

Special text file format, introduced by Intel Corporation, to encode the executable code generated by cross-assemblers or cross-compilers. Unlike binary files, the Intel hex files contain information about the address where the code must be loaded in the program memory and special checksums for error detection when the file is transmitted over serial communication lines.
See also: Motorola S file format

Interrupt

A mechanism that allows an external event to temporarily put on hold the normal execution of the program, forcing the execution of a specific subroutine. Once the interrupt service subroutine completes, the main program continues from the point where it was interrupted.
See also: Interrupt vector, Interrupt service routine

Interrupt service routine

The interrupt service routine (ISR) is a program sequence similar to a subroutine, executed every time the associated interrupt occurs. Unlike "regular" subroutines, the ISR ends with a special instruction called Return from Interrupt (RTI, RETI). The starting address of the interrupt service routine is defined by the interrupt vector.
See also: Interrupt, Interrupt vector

Interrupt vector

Reserved memory area, associated with each interrupt source, that contains either the address of the interrupt service routine, or a jump instruction to this routine.
See also: Interrupt, Interrupt service routine

Instruction

Basic element of a program, which encodes an elementary operation. A program is a logic sequence of instructions. The CPU reads the code of each instruction from the program memory, decodes and executes it, then fetches the next instruction in sequence.
See also: Program memory

I/O line

Input/output line. A mechanism for a computer system to interact with the outside world. An input line is used to sense the electrical level of an external signal, and an output line modifies its electrical status as directed by the program. In many cases, the I/O lines can be configured by the program to act as inputs or outputs.
See also: I/O port, Data direction register, Peripheral interface

I/O port

A group of I/O lines that can be accessed simultaneously. The number of lines of an I/O port depends on the width of the data bus, i. e. the number of data bits that can be transferred in a certain time on the data bus.
See also: I/O line, Data bus

ISP

Acronym for In-System Programming. A technique to program the internal program memory of some microcontrollers, by means of a specially designed interface.
See also: Program memory

Master–slave protocol

Communication protocol in which one MASTER device controls one or more SLAVE devices.

MCU

Acronym for Microcontroller Unit. Designates a single-chip computer, a structure that integrates a CPU, certain amounts of program memory and data memory, and a number of peripheral interfaces, all in one chip.
See also: CPU, ROM, RAM, Memory, Architecture, Peripheral interface

Memory

A device designed to store data. The simplest memory cell is the D-type flip–flop. In a microcontroller system, the memory is organized as an array of locations, each having a distinct address. The number of bits of each location is determined by the width of the data bus.
See also: Data memory, Program memory, RAM, ROM, Architecture

Memory map

A list showing how the addresses of an address space are assigned to resources.
See also: Address space, Architecture, Memory

Motorola S File format

Special text file format, introduced by Motorola, to encode the executable code generated by cross-assemblers or cross-compilers.
One distinctive feature of these files is that all the lines in this type of file start with the letter S. Unlike binary files, the Motorola S files contain information about the address where the code must be loaded in the program memory, and special checksums for error detection when the file is transmitted over serial communication lines.
See also: Intel hex file format

Network

A structure consisting in a number of devices interconnected in a way that allows data exchange between them, according to a set of rules, known as communication protocol.
See also: Protocol, Master–slave protocol

Overrun error

In a communication system, this error occurs at the receiver when a new character is received before the previous character has been handled.
See also: Framing error

Parity bit

Error detection technique consisting in adding an extra bit to each group of bits transmitted, so that the group always has an odd or even number of 1's. Usually the parity is checked at the byte level.
See also: CRC, Framing error, Overrun error

Peripheral interface

A circuit designed to sense/drive a number of I/O lines in order to provide a computer system with the capability to communicate with external equipment.
See also: I/O line, I/O port.

Prescaller

Programmable counter used by some peripheral interfaces to divide the system clock to provide a variable frequency clock to the interface.
See also: Peripheral interface

Program memory

Non-volatile memory used to store program information. The program memory is connected to the CPU through address and data buses.
See also: Memory, ROM, Flash memory, Address bus, Data bus, Architecture

Protocol

In a communication system or network, the protocol is a set of rules defining the structure of the data packets transmitted, and the conditions of access to the communication bus for each device in the system.
See also: Master–slave protocol

PWM

Acronym for Pulse Width Modulation. Some microcontrollers contain a special interface able to generate PWM signals with programmable frequency and duty cycle.
See also: Timer

RAM

Acronym for Random Access Memory. Typical read/write memory used to store variables (data memory). RAM memory is volatile, i. e. it loses its contents at power-down.
See also: Memory, Data memory, ROM, EEPROM, Flash memory

Register

Memory location, identified by a unique address, associated with particular control or status information regarding a peripheral interface or a part or function of the CPU.
See also: Memor, Peripheral interface

Resolution

The number of bits of an ADC or DAC circuit, which directly determines the number of quantization levels of the analog signal; e. g. a 10-bit ADC can distinguish 2^{10} values of amplitude of the analog input.
See also: ADC, DAC

ROM

Acronym for Read Only Memory. Non-volatile memory used to store program or data constants.
See also: Memory, Data memory, Program memory, RAM, EEPROM, Flash memory.

RS232

Recommended Standard of EIA organization, for point-to-point serial communication, where signals are carried as single voltages, referred to a common ground. The voltage levels associated with the logic levels 0 and 1 are as follows:

- For logic 0, the voltage on the communication line must be in the range +6 V to 15 V.
- For logic 1, the voltage must be in the range −6 V to −15 V.

See also: Differential communication, RS422, RS485

RS422

Recommended Standard for point-to-point differential communication, where two conductors are used to transmit each digital signal, the logic levels being defined by the relative difference of potential between the two conductors.
See also: Differential communication, RS485

RS485

Recommended Standard for multipoint differential communication. RS485 allows up to 32 transmitters to be connected to the same twisted pair of conductors. Therefore, each RS485 transmitter must be able to be virtually disconnected from the bus, by driving its output into a high-impedance status.
See also: Differential communication, RS422

SAR

Successive Approximation Register. Technique used by some ADC circuits to obtain the digital value of the analog input, by successively comparing the analog input with an analog value obtained by converting to analog the contents of the register that stores the results of comparisons.
See also: ADC, DAC

SCI

Serial Communication Interface. Peripheral interface designed to allow asynchronous serial communication between computer systems and peripheral equipment.
See also: Asynchronous communication, UART, Start bit, Stop bit, Framing error

S&H circuit

Acronym for sample & hold circuit. A circuit used by some ADC circuits to hold the instantaneous values of the analog input, for the duration of the conversion process.
See also: ADC

SPI

Synchronous Peripheral Interface. Particular master–slave communication interface, where the master device generates a common clock used by the transmitter and receiver to serially shift data in both directions.
See also: Master–slave communication, Synchronous communication

Stack

A data structure having Last In First Out (LIFO) access type. In practice, in most cases, the stack is a reserved area of the system RAM memory, addressed by a special register, called the Stack Pointer (SP).
See also: Stack Pointer

Stack pointer

A register used to address the stack. The stack pointer points to the top of the stack. Depending on the implementation, the top of the stack can be defined as the first empty location in the stack, or as the most recently pushed data.
See also: Stack

Start bit

In asynchronous serial communication, before the actual data transmission begins, the transmission line is pulled to the polarity opposite to the polarity of the idle line, for a duration of a bit time. This technique is destined to provide a means to synchronize the receiver.
See also: Asynchronous communication, Stop bit

Stop bit

In asynchronous serial communication, after the transmission of the data bits and any parity bit, the polarity of the transmission line is returned to the polarity of the idle line for the duration of a bit time.
See also: Asynchronous communication, Start bit

Synchronous communication

Communication technique that requires a common clock signal for the transmitter and receiver(s) in order to coordinate (synchronize) their activity.
See also: Asynchronous communication, SCI, SPI

UART

Acronym for Universal Asynchronous Receiver Transmitter. Interface circuit for asynchronous communication.
See also: Asynchronous communication, SCI

Von Neumann

Computer architecture characterized by common address and data buses for program memory and data memory.
See also: Architecture, Harvard

A.2 Description of the Registers of 68HC11F1

	Bit 7	6	5	4	3	2	1	Bit 0
PORTA	PA7	PA6	PA5	PA4	PA3	PA2	PA1	PA0
DDRA	DDA7	DDA6	DDA5	DDA4	DDA3	DDA2	DDA1	DDA0
PORTG	PG7	PG6	PG5	PG4	PG3	PG2	PG1	PG0
DDRG	DDG7	DDG6	DDG5	DDG4	DDG3	DDG2	DDG1	DDG0
PORTB	PB7	PB6	PB5	PB4	PB3	PB2	PB1	PO
PORTF	PF7	PF6	PF5	PF4	PF3	PF2	PF1	PO
PORTC	PC7	PC6	PC5	PC4	PC3	PC2	PC1	PC0
DDRC	DDC7	DDC6	DDC5	DDC4	DDC3	DDC2	DDC1	DDC0
PORTD	0	0	PD5	PD4	PD3	PD2	PD1	PO
DDRD	0	0	DDD5	DDD4	DDD3	DDD2	DDD1	DDD0
PORTE	PE7	PE6	PE5	PE4	PE3	PE2	PE1	PO
CFORC	FOC1	FOC2	FOC3	FOC4	FOC5	0	0	0
OC1M	OC1M7	OC1M6	OC1M5	OC1M4	OC1M3	0	0	0
OC1D	OC1D7	OC1D6	OC1D5	OC1D4	OC1D3	0	0	0
TCNTH	Bit 15	14	13	12	11	10	9	Bit 8
TCNTL	Bit 7	6	5	4	3	2	1	Bit 0
TIC1H	Bit 15	14	13	12	11	10	9	Bit 8
TIC1L	Bit 7	6	5	4	3	2	1	Bit 0
TIC2H	Bit 15	14	13	12	11	10	9	Bit 8
TIC2L	Bit 7	6	5	4	3	2	1	Bit 0
TIC3H	Bit 15	14	13	12	11	10	9	Bit 8
TIC3L	Bit 7	6	5	4	3	2	1	Bit 0
TOC1H	Bit 15	14	13	12	11	10	9	Bit 8
TOC1L	Bit 7	6	5	4	3	2	1	Bit 0
TOC2H	Bit 15	14	13	12	11	10	9	Bit 8
TOC2L	Bit 7	6	5	4	3	2	1	Bit 0
TOC3H	Bit 15	14	13	12	11	10	9	Bit 8
TOC3L	Bit 7	6	5	4	3	2	1	Bit 0
TOC4H	Bit 15	14	13	12	11	10	9	Bit 8
TOC4L	Bit 7	6	5	4	3	2	1	Bit 0
TI4/O5H	Bit 15	14	13	12	11	10	9	Bit 8
TI4/O5L	Bit 7	6	5	4	3	2	1	Bit 0
TCTL1	OM2	OL2	OM3	OL3	OM4	OL4	OM5	OM5
TCTL2	EDG4B	EDG4A	EDG1B	EDG1A	EDG2B	EDG2A	EDG3B	EDG3A
TMSK1	OC1I	OC2I	OC3I	OC4I	I4/O5	IC1I	IC2I	IC3I
TFLG1	OC1F	OC2F	OC3F	OC4F	I4/O5F	IC1F	IC2F	IC3F
TMSK2	TOI	RTII	PAOVI	PAII	0	0	PR1	PR0
TFLG2	TOF	RTIF	PAOVF	PAIF	0	0	0	0
PACTL	0	PAEN	PAMOD	PEDGE	0	I4/O5	RTR1	RTR0
PACNT	Bit 7	6	5	4	3	2	1	Bit0
SPCH	Bit 7	6	5	4	3	2	1	Bit0
SPSR	SPIE	SPE	DWOM	MSTR	CPOL	CPHA	SPR1	CPO
SPDR	SPIF	WCOL	0	MODF	0	0	0	Bit0
BAUD	Bit 7	6	5	4	3	2	1	Bit0

	Bit 7	6	5	4	3	2	1	Bit 0
SCCR1	TCLR	0	SCP1	SCP0	RCKB	SCR2	SCR1	SCR0
SCCR2	R8	T8	0	M	WAKE	0	0	0
SCSR	TIE	TCIE	RIE	ILIE	TE	RE	RWU	SBK
SCDR	TDRE	TC	RDRF	IDLE	OR	NF	FE	0
ADCTL	Bit 7	6	5	4	3	2	1	Bit0
ADR1	CCF	0	SCAN	MULT	CD	CC	CB	CA
ADR2	Bit 7	6	5	4	3	2	1	Bit0
ADR3	Bit 7	6	5	4	3	2	1	Bit0
ADR4	Bit 7	6	5	4	3	2	1	Bit0
BROT	0	0	0	PTCON	BPRT3	BPRT2	BPRT1	BPRT0
OPT2	GWOM	CWOM	CLK4X	0	0	0	0	0
OPTION	ADPU	CSEL	IRQE	DLY	CME	FCME	CR1	CR0
COPRST	Bit 7	6	5	4	3	2	1	Bit O
PPROG	ODD	EVEN	0	BYTE	ROW	ERASE	EELAT	EEPGM
HPRIO	RBOOT	SMOD	MDA	IRV	PSEL3	PSEL2	PSEL1	PSEL0
INIT	RAM3	RAM2	RAM1	RAM0	REG3	REG2	REG1	REG0
TEST1	TILOP	0	OCCR	CBYP	DISR	FCM	FCOP	0
CONFIG	EE3‘	EE2	EE1	EEO	1	NOCOP	1	EEON
CSSTRH	IO1SA	IO1SB	IO2SA	IO2SB	GSTHA	GSTHB	PSTHA	PSTHB
CSCTL	IO1EN	IO1PL	IO2EN	IO2PL	GCSPR	PCSEN	PSIZA	PSIZB
CSGADR	GA15	GA14	GA13	GA12	GA11	GA10	0	0
CSGSIZ	IO1AV	IO2AV	0	GNPOL	GAVLD	GSIZA	GSIZB	GSIZC

Note

The register block of 68HC11F1 is located by default in the address range $1000–$105F. The whole block can be relocated by writing the control bits [REG3–REG0] in the INIT register. Consult the data sheets for details on the registers of other HC11 family members.

A.3 HC11 Instruction Set

Mnemonic	Operation	Description	Flags
ABA	Add Accumulators	$A + B \rightarrow A$	H,N,Z,V,C
ABX	Add B to X	$IX + (00:B) \rightarrow IX$	
ABY	Add B to Y	$IY + (00:B) \rightarrow IY$	
ADCA (opr)	Add with Carry to A	$A + M + C \rightarrow A$	H,N,Z,V,C
ADCB (opr)	Add with Carry to B	$B + M + C \rightarrow B$	H,N,Z,V,C
ADDA (opr)	Add Memory to A	$A + M \rightarrow A$	H,N,Z,V,C
ADDB (opr)	Add Memory toB	$B + M \rightarrow B$	H,N,Z,V,C
ADDD (opr)	Add 16-Bit to D	$D + (M:M+1) \rightarrow D$	N,Z,V,C
ANDA (opr)	AND A with Memory	$A \cap M \rightarrow A$	N,Z,V
ANDB (opr)	AND B with Memory	$B \cap M \rightarrow B$	N,Z,V
ASL (opr)	Arithmetic Shift Left	$C \leftarrow b7 \leftarrow b6 \ldots\ b0 \leftarrow 0$	N,Z,V,C
ASLA	Arithmetic Shift Left A	$C \leftarrow b7 \leftarrow b6 \ldots\ b0 \leftarrow 0$	N,Z,V,C
ASLB	Arithmetic Shift Left B	$C \leftarrow b7 \leftarrow b6 \ldots\ b0 \leftarrow 0$	N,Z,V,C
ASLD	Arithmetic Shift Left D	$C \leftarrow b15 \leftarrow b14 \ldots\ b0 \leftarrow 0$	N,Z,V,C
ASR	Arithmetic Shift Right	$b7 \rightarrow b7 \rightarrow\ \ldots\ b0 \rightarrow C$	N,Z,V,C
ASRA	Arithmetic Shift Right A	$b7 \rightarrow b7 \rightarrow b6 \rightarrow\ \ldots \rightarrow b0 \rightarrow C$	N,Z,V,C
ASRB	Arithmetic Shift Right B	$b7 \rightarrow b7 \rightarrow b6 \rightarrow\ \ldots \rightarrow b0 \rightarrow C$	N,Z,V,C
BCC (rel)	Branch if Carry Clear	Branch if $C = 0$	
BCLR (opr) (msk)	Clear Bit(s)	$M \cap (mm) \rightarrow M$	N,Z,V
BCS (rel)	Branch if Carry Set	Branch if $C = 1$	
BEQ (rel)	Branch if = Zero	Branch if $Z = 1$	
BGE (rel)	Branch if >= Zero	Branch if $N \oplus V = 0$	
BGT (rel)	Branch if > Zero	Branch if $Z \cup (N \oplus V) = 0$	
BHI (rel)	Branch if Higher	Branch if $C \cup Z = 0$	
BHS (rel)	Branch if Higher or Same	Branch if $C = 0$	
BITA (opr)	Bit(s)Test A with Memory	$A \cap M$	N,Z,V
BITB (opr)	Bit(s)Test B with Memory	$B \cap M$	N,Z,V
BLE (rel)	Branch if <= Zero	Branch if $Z \cup (N \oplus V) = 1$	
BLO (rel)	Branch if Lower	Branch if $C = 1$	
BLS (rel)	Branch if Lower or Same	Branch if $C \cup Z = 1$	
BLT (rel)	Branch if < Zero	Branch if $N \oplus V = 1$	
BMI (rel)	Branch if Minus	Branch if $N = 1$	
BNE (rel)	Branch if not = Zero	Branch if $Z = 0$	
BPL (rel)	Branch if Plus	Branch if $N = 0$	
BRA (rel)	Branch Always	Branch if $1 = 1$	
BRCLR(opr) (msk) (rel)	Branch if Bit(s) Clear	Branch if $M \cap mm = 0$	
BRN (rel)	Branch Never	Branch if $1 = 0$	
BRSET(opr) (msk) (rel)	Branch if Bit(s) Set	Branch if $(M)\ \cap mm = 1$	
BSET (opr) (msk)	Set Bit(s)	$M \cup mm \rightarrow M$	N,Z,V

Mnemonic	Operation	Description	Flags
BSR (rel)	Branch to Subroutine		
BVC (rel)	Branch if Overflow Clear	Branch if $V = 0$	
BVS (rel)	Branch if Overflow Set	Branch if $V = 1$	
CBA	Compare A to B	$A - B$	N,Z,V,C
CLC	Clear Carry Bit	$0 \rightarrow C$	C
CLI	Clear Interrupt Mask	$0 \rightarrow I$	I
CLR (opr)	Clear Memory Byte	$0 \rightarrow M$	N,Z,V,C
CLRA	Clear Accumulator A	$0 \rightarrow A$	N,Z,V,C
CLRB	Clear Accumulator B	$0 \rightarrow B$	N,Z,V,C
CLV	Clear Overflow Flag	$0 \rightarrow V$	V
CMPA (opr)	Compare A to Memory	$A - M$	N,Z,V,C
CMPB (opr)	Compare B to Memory	$B - M$	N,Z,V,C
COM (opr)	One's Complement Memory Byte	$\$FF - M \rightarrow M$	N,Z,V,C
COMA	Ones Complement A	$\$FF - A \rightarrow A$	N,Z,V,C
COMB	Ones Complement B	$\$FF - B \rightarrow B$	N,Z,V,C
CPD (opr)	Compare D to Memory 16-Bit	$D - (M : M + 1)$	N,Z,V,C
CPX (opr)	Compare X to Memory 16-Bit	$IX - (M : M + 1)$	N,Z,V,C
CPY (opr)	Compare Y to Memory 16-Bit	$IY - (M : M + 1)$	N,Z,V,C
DAA	Decimal Adjust A	Adjust Sum to BCD	N,Z,V,C
DEC (opr)	Decrement Memory Byte	$M - 1 \rightarrow M$	N,Z,V
DECA	Decrement Accumulator A	$A - 1 \rightarrow A$	N,Z,V
DECB	Decrement Accumulator B	$B - 1 \rightarrow B$	N,Z,V
DES	Decrement Stack Pointer	$SP - 1 \rightarrow SP$	
DEX	Decrement Index Register X	$IX - 1 \rightarrow IX$	Z
DEY	Decrement Index Register Y	$IY - 1 \rightarrow IY$	Z
EORA (opr)	Exclusive OR A with Memory	$A \oplus M \rightarrow A$	N,Z,V
EORB (opr)	Exclusive OR B with Memory	$B \oplus M \rightarrow B$	N,Z,V
FDIV	Fractional Divide 16 by 16	$D / IX \rightarrow IX; r \rightarrow D$	Z,V,C
IDIV	Integer Divide 16 by 16	$D / IX \rightarrow IX; r \rightarrow D$	Z,V,C
INC (opr)	Increment Memory Byte	$M + 1 \rightarrow M$	N,Z,V
INCA	Increment Accumulator A	$A + 1 \rightarrow A$	N,Z,V
INCB	Increment Accumulator B	$B + 1 \rightarrow B$	N,Z,V
INS	Increment Stack Pointer	$SP + 1 \rightarrow SP$	
INX	Increment Index Register X	$IX + 1 \rightarrow X$	Z
INY	Increment Index Register Y	IY+1 → Y	X
JMP(opr)	Unconditional Jump		
JSR(opr)	Jump to subroutine		
LDAA(opr)	Load Accumulator A	$M \rightarrow A$	N,Z,V
LDAB(opr)	Load Accumulator B	$M \rightarrow B$	N,Z,V
LDD(opr)	Load Double Accumulator D	$M \rightarrow A, M + 1 \rightarrow B$	N,Z,V
LDS(opr)	Load Stack Pointer	$(M : M + 1) \rightarrow SP$	N,Z,V
LDX(opr)	Load Index Register X	$(M : M + 1) \rightarrow IX$	N,Z,V
LDY(opr)	Load Index Register Y	$(M : M + 1) \rightarrow IY$	N,Z,V
LSL(opr)	Logical shift left	$C \leftarrow b7 \leftarrow b6 \ldots\ b0 \leftarrow 0$	N,Z,V,C
LSLA	Logical shit left A	$C \leftarrow b7 \leftarrow b6 \ldots\ b0 \leftarrow 0$	N,Z,V,C
LSLB	Logical shift left B	$C \leftarrow b7 \leftarrow b6 \ldots\ b0 \leftarrow 0$	N,Z,V,C
LSLD	Logical shift left Double	$C \leftarrow b15 \leftarrow\ \ldots\ \leftarrow b0 \leftarrow 0$	N,Z,V,C

Mnemonic	Operation	Description	Flags
LSR(opr)	Logical shift right	$0 \rightarrow b7 \rightarrow b6 \ldots b0 \rightarrow C$	N,Z,V,C
LSRA	Logical shift right A	$0 \rightarrow b7 \rightarrow b6 \ldots b0 \rightarrow C$	N,Z,V,C
LSRB	Logical shift right B	$0 \rightarrow b7 \rightarrow b6 \ldots b0 \rightarrow C$	N,Z,V,C
LSRD	Logical shift right Double	$0 \rightarrow b15 \rightarrow \ldots \rightarrow b0 \rightarrow C$	N,Z,V,C
MUL	Multiply A by B	$A \times B \rightarrow D$	C
NEG(opr)	Tow's complement memory byte	$0 - M \rightarrow M$	N,Z,V,C
NEGA	Tow's complement A	$0 - A \rightarrow A$	N,Z,V,C
NEGB	Tow's complement B	$0 - B \rightarrow B$	N,Z,V,C
NOP	No operation	No operation	
ORAA (opr)	OR Accumulator A	$A \cup M \rightarrow A$	N,Z,V
ORAB (opr)	OR Accumulator B	$B \cup M \rightarrow B$	N,Z,V
PSHA	Push A onto Stack	$A \rightarrow$ Stk, $SP = SP - 1$	
PSHB	Push B onto Stack	$B \rightarrow$ Stk, $SP = SP - 1$	
PSHX	Push X onto Stack(Lo First)	$IX \rightarrow$ Stk, $SP = SP - 2$	
PSHY	Push Y onto Stack(Lo First)	$IY \rightarrow$ Stk, $SP = SP - 2$	
PULA	Pull A from Stack	$SP = SP + 1$, $A \leftarrow$ Stk	
PULB	Pull B from Stack	$SP = SP + 1$, $B \leftarrow$ Stk	
PULX	Pull X from Stack (Hi First)	$SP = SP + 2$, $IX \leftarrow$ Stk	
PULY	Pull Y from Stack (Hi First)	$SP = SP + 2$, $IY \leftarrow$ Stk	
ROL (opr)	Rotate Left Memory	$b0 \leftarrow C \leftarrow b7 \leftarrow b6$ $\ldots \leftarrow b0$	N,Z,V,C
ROLA	Rotate Left A	$b0 \leftarrow C \leftarrow b7 \leftarrow b6$ $\ldots \leftarrow b0$	N,Z,V,C
ROLB	Rotate Left B	$b0 \leftarrow C \leftarrow b7 \leftarrow b6$ $\ldots \leftarrow b0$	N,Z,V,C
ROR (opr)	Rotate Right Memory	$b7 \rightarrow b6 \ldots b0 \rightarrow C \rightarrow b7$	N,Z,V,C
RORA	Rotate Right A	$b7 \rightarrow b6 \ldots b0 \rightarrow C \rightarrow b7$	N,Z,V,C
RORB	Rotate Right B	$b7 \rightarrow b6 \ldots b0 \rightarrow C \rightarrow b7$	N,Z,V,C
RTI	Return from Interrupt		N,Z,V,C
RTS	Return from Subroutine		
SBA	Subtract B from A	$A - B \rightarrow A$	N,Z,V,C
SBCA (opr)	Subtract with Carry from A	$A - M - C \rightarrow A$	N,Z,V,C
SBCB (opr)	Subtract with Carry from B	$B - M - C \rightarrow B$	N,Z,V,C
SEC	Set Carry	$1 \rightarrow C$	C
SEI	Set Interrupt Mask	$1 \rightarrow I$	I
SEV	Set Overflow Flag	$1 \rightarrow V$	V
STAA (opr)	Store Accumulator A	$A \rightarrow M$	N,Z,V
STAB (opr)	Store Accumulator B	$B \rightarrow M$	N,Z,V
STD (opr)	Store Accumulator D	$A \rightarrow M$, $B \rightarrow M + 1$	N,Z,V
STOP	Stop Internal Clocks		
STS (opr)	Store Stack Pointer	$SP \rightarrow M : M + 1$	N,Z,V
STX (opr)	Store Index Register X	$IX \rightarrow M : M + 1$	N,Z,V
STY (opr)	Store Index Register Y	$IY \rightarrow M : M + 1$	N,Z,V
SUBA (opr)	Subtract Memory from A	$A - M \rightarrow A$	N,Z,V,C
SUBB (opr)	Subtract Memory from B	$B - M \rightarrow B$	N,Z,V,C
SUBD (opr)	Subtract Memory from D	$D - (M : M + 1) \rightarrow D$	N,Z,V,C
SWI	Software Interrupt		I

Mnemonic	Operation	Description	Flags
TAB	Transfer A to B	A → B	N,Z,V
TAP	Transfer A to CC Register	A → CCR	All
TBA	Transfer B to A	B → A	N,Z,V
TEST	TEST (Only in Test Modes)	Address Bus Counts	
TPA	Transfer CC Register to A	CCR → A	
TST (opr)	Test for Zero or Minus	M – 0	N,Z,V,C
TSTA	Test A for Zero or Minus	A – 0	N,Z,V,C
TSTB	Test B for Zero or Minus	B – 0	N,Z,V,C
TSX	Transfer Stack Pointer to X	SP + 1 → IX	
TSY	Transfer Stack Pointer to Y	SP + 1 → IY	
TXS	Transfer X to Stack Pointer	IX – 1 → SP	
TYS	Transfer Y to Stack Pointer	IY – 1 → SP	
WAI	Wait for Interrupt	Stack Regs & WAIT	
XGDX	Exchange D with X	IX → D, D → IX	
XGDY	Exchange D with Y	IY → D, D → IY	

A.4 An Example of Expanded Structure with HC11

The example presented below is an illustration of a typical structure with 68HC11E9 operating in expanded multiplexed mode (MODA = 1, MODB = 1).

Figure A4.1 shows the microcontroller and the bus demultiplexer, implemented with the latch 74LS573, controlled by the Address Strobe (AS) signal. Note that the unused MCU pins are not represented in this figure.

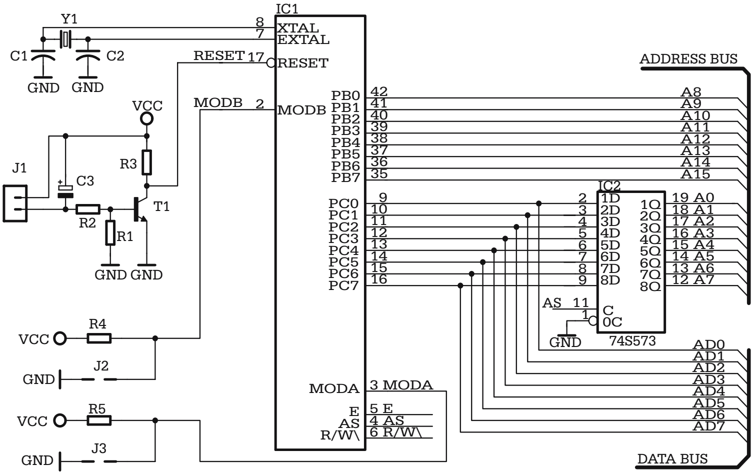


Fig. A4.1. MCU 68HC11E9 and the bus demultiplexer

Figure A4.2 shows the external memory circuits and the address decoder. The external ROM is an 8-kilobyte, 2764 EPROM (ICx), selected with CSE000H. ICy is a 6264 8-kilobyte RAM selected with CS2000H. Besides chip select, the 6264 circuit requires an additional control signal for the OE (Output Enable) input. This is OERAM, obtained by inverting the signal R/W\ with the gate IC7A.

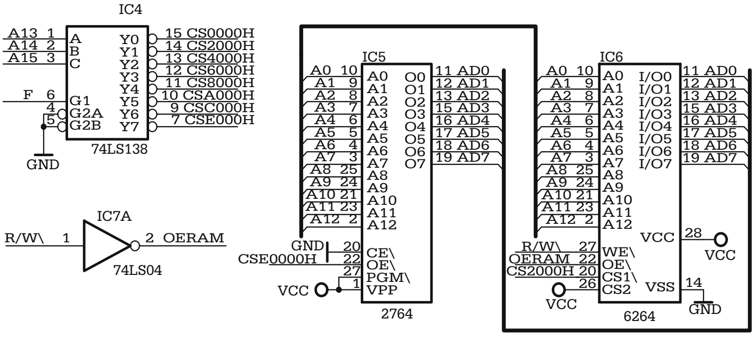

IC4
A13 A14 A15
F
A B C
G1 G2A G2B
Y0 Y1 Y2 Y3 Y4 Y5 Y6 Y7
CS0000H CS2000H CS4000H CS6000H CS8000H CSA000H CSC000H CSE000H
74LS138
GND
IC7A
R/W\
OERAM
74LS04
IC5
A0 A1 A2 A3 A4 A5 A6 A7 A8 A9 A10 A11 A12
O0 O1 O2 O3 O4 O5 O6 O7
AD0 AD1 AD2 AD3 AD4 AD5 AD6 AD7
CE\ OE\ PGM\ VPP
CSE0000H
VCC
2764
IC6
I/O0 I/O1 I/O2 I/O3 I/O4 I/O5 I/O6 I/O7
WE\ OE\ CS1\ CS2
CS2000H
VSS
6264

Fig. A4.2. External memory and address decoder

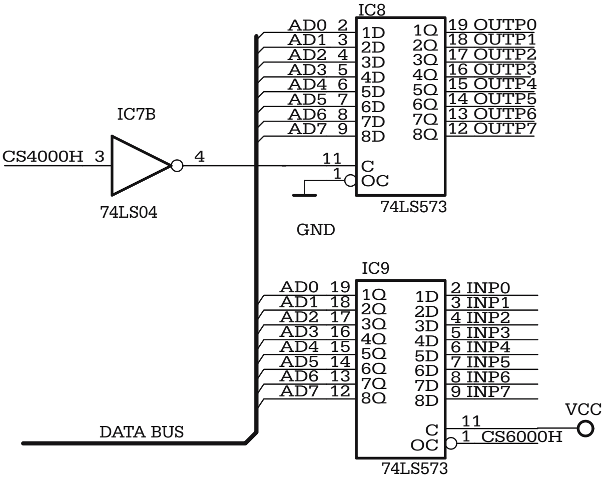


Fig. A4.3. External I/O ports

Two additional selection signals CS4000H and CS6000H are used to control the external input and output ports, shown in Fig. A4.3.

A.5 Using HC11 in Bootstrap Mode

If the input lines MODA and MODB are grounded during RESET, HC11 enters a special operating mode, called *bootstrap*. In bootstrap mode, the microcontroller executes a program, called *bootloader*, located in a small ROM, invisible in the memory map in normal modes.

The bootloader allows user programs to be loaded into the MCU RAM, via the serial communication interface SCI, and, when the transmission completes, the user program is automatically launched.

In principle, the user program loaded this way can use any of the MCU resources, but the most common use of the bootstrap mode is for writing the CONFIG register and EEPROM constants. Some members of the HC11 family have internal EPROM or OTPROM (One Time Programmable ROM), which can also be programmed in bootstrap mode.

The first action of the bootloader is to send a break (a long zero) character on the serial line. If the answer of the host is another break, then the bootloader passes the control to a program located in the EEPROM, by executing an unconditional jump to the first address of the EEPROM. If the host sends a $FF, this is used by the bootloader of 68HC11E9 to determine the baud rate, by selecting one of the two possible baud rates (1200 or 7812 baud). Note that other versions of HC11 allow different baud rates. See the specific data sheets for details.

Once the baud rate is selected, the bootloader starts receiving the binary characters from the serial line, echoes each received character to the host, and stores them in RAM. If the end of RAM is reached, or no character is received for four character times, the bootloader abandons the communication process and executes a jump to the first address of RAM, passing control to the user program.

A complete description of the HC11 bootloader is available in the Motorola application note AN1060, available on the internet.

The development board described in Chap. 9 is provided with two jumpers (JP2, JP3) that control the MODA, MODB inputs and allow the MCU to enter the special bootstrap operating mode. To facilitate testing this operating mode, the accompanying CD contains a small utility program, called BLT11.EXE (Bootloader terminal for HC11).

BLT11 does not require any installation procedure, and may be run from the CD-ROM. It allows the user to select the COM port to use, configures it for 1200 baud, loads a binary at the user's choice and sends its contents to the selected communication port, preceded by $FF.

The binary characters echoed by the MCU are displayed in the terminal window as pairs of two hex digits separated by a space. When the download completes, the user program is automatically launched. This program waits for a character from the SCI and compares it to CR ($0D) and SP (Space $20). If SP is detected, the program reads the current value of the CONFIG register and sends it to the host through the SCI. If CR is detected, the program writes the constant NEWCFG in CONFIG, and sends it to the host to acknowledge the operation.

Below is the full listing of a program that modifies the CONFIG register.

```
     TITLE PROGRAMMING THE CONFIG REGISTER
        INCLUDE  68HC11F1.DEF
NEWCFG  EQU    $0F
*EEPGM  EQU    $01
*EELAT  EQU    $02
*ERASE  EQU    $04
BYTE    EQU    $20
ENDRAM  EQU    $1FF           ;valid for E9 too
CR      EQU    $0D
SP      EQU    $20
        CODE                  ;no ORG!!
START   LDS    #ENDRAM        ;init SP
        LDAA   SCSR           ;clear SCI flags if any
        LDAA   SCDR
        CLR    BPROT
REC05   LDAA   SCSR           ;wait a character from SCI
        ANDA   #$20
        BEQ    REC05
REC10   LDAA   SCDR           ;get character
        CMPA   #CR
        BEQ    E2W            ;if CR write new value
        CMPA   #SP
        BEQ    SENDC          ;if SP send current value
        BRA    REC05          ;endless loop
E2W     LDY    #CONFIG
        BSR    E2BE           ;erase it first
        LDAB   #EELAT
        STAB   PPROG
        LDAA   #NEWCFG
        STAA   CONFIG         ;latch the address
        ORAB   #EEPGM
        STAB   PPROG
        BSR    DLY10          ;wait about 10ms
SENDC   LDAA   SCSR
        ANDA   #$80           ;transmit complete?
        BEQ    SENDC
        LDAA   CONFIG
        STAA   SCDR
        BRA    REC05          ;back to start
     *LOCAL SUBROUTINES
E2BE    LDAB   #$16           ;BYTE=1,ERASE=1,EELAT=1
        STAB   PPROG
        STAB   0,Y            ;latch address
        LDAB   #$17           ;make EEPGM=1
        STAB   PPROG          ;start Vpp (charge pump)
        BSR    DLY10
        CLR    PPROG          ;stop Vpp and return to read
        RTS
DLY10   LDX    #$3000
```

```
DLOOP   DEX
        BNE   DLOOP
        CLR   PPROG
        RTS
        END
```

Remember that the EEPROM cells that form the CONFIG register are doubled by a RAM-type register for read operations, which is loaded only once, at RESET. Any new value written to the CONFIG register becomes visible after the next RESET.

Important note. Do *not* clear the bit NOCOP in the CONFIG register unless you really want to enable the watchdog. Also note that some HC11 versions have a special bit in CONFIG, called NOSEC. Clearing this bit enables a security feature, which prevents the MCU from entering the bootstrap operating mode. You may not be able to erase the CONFIG register again.

A.6 The Registers of AT90S8535

	Bit 7	6	5	4	3	2	1	Bit0
SREG	I	T	H	S	V	N	Z	C
SPH	–	–	–	–	–	SP10	SP9	SP8
SPL	SP7	SP6	SP5	SP4	SP3	SP2	SP1	SP0
GIMSK	INT1	INT0						
GIFR	INTF1	INTF0						
TIMSK	OCIE2	TOIE2	TICIE1	OCIE1A	OCIE1B	TOIE1		TOIEO
TIFR	OCF2	TOV2	ICF1	OCF1A	OCF1B	TOV1		TOV0
MCUCR		SE	SM1	SM0	ISC11	ISC10	ISC01	ISC00
MCUCSR							EXTRF	PORF
TCCR0						CS02	CS01	CS00
TCNT0	Timer/Counter0 (8 Bits)							
TCCR1A	COM1A1	COM1A0	COM1B1	COM1B0			PWM11	PWM10
TCCR1B	ICNC1	ICES1			CTC1	CS12	CS11	CS10
TCNT1H	Timer/Counter1 – Counter Register High Byte							
TCNT1L	Timer/Counter1 – Counter Register Low Byte							
OCR1AH	Timer/Counter1 – Output Compare Register A High Byte							
OCR1AL	Timer/Counter1 – Output Compare Register A Low Byte							
OCR1BH	Timer/Counter1 – Output Compare Register B High Byte							
OCR1BL	Timer/Counter1 – Output Compare Register B Low Byte							
ICR1H	Timer/Counter1 – Input Capture Register High Byte							
ICR1L	Timer/Counter1 – Input Capture Register Low Byte							
TCCR2		PWM2	COM21	COM20	WGM21	CS22	CS21	CS20
TCNT2	Timer/Counter2 (8 Bits)							
OCR2	Timer/Counter2 Output Compare Register							
ASSR	–	–	–	–	AS2	TCN2UB	OCR2UB	TCR2UB
WDTCR	–	–	–	WDTOE	WDE	WDP2	WDP1	WDP0
UCSRC	URSEL	UMSEL	UPM1	UPM0	USBS	UCSZ1	UCSZ0	UCPOL
EEARH	–	–	–	–	–	–	–	EEAR8
EEARL	EEPROM Address Register Low Byte							
EEDR	EEPROM Data Register							
EECR	–	–	–	–	EERIE	EEMWE	EEWE	EERE
PORTA	PORTA7	PORTA6	PORTA5	PORTA4	PORTA3	PORTA2	PORTA1	PORTA0
DDRA	DDA7	DDA6	DDA5	DDA4	DDA3	DDA2	DDA1	DDA0
PINA	PINA7	PINA6	PINA5	PINA4	PINA3	PINA2	PINA1	PINA0
PORTB	PORTB7	PORTB6	PORTB5	PORTB4	PORTB3	PORTB2	PORTB1	PORTB0
DDRB	DDB7	DDB6	DDB5	DDB4	DDB3	DDB2	DDB1	DDB0
PINB	PINB7	PINB6	PINB5	PINB4	PINB3	PINB2	PINB1	PINB0
PORTC	PORTC7	PORTC6	PORTC5	PORTC4	PORTC3	PORTC2	PORTC1	PORTC0
DDRC	DDC7	DDC6	DDC5	DDC4	DDC3	DDC2	DDC1	DDC0
PINC	PINC7	PINC6	PINC5	PINC4	PINC3	PINC2	PINC1	PINC0
PORTD	PORTD7	PORTD6	PORTD5	PORTD4	PORTD3	PORTD2	PORTD1	PORTD0
DDRD	DDD7	DDD6	DDD5	DDD4	DDD3	DDD2	DDD1	DDD0
PIND	PIND7	PIND6	PIND5	PIND4	PIND3	PIND2	PIND1	PIND0
SPDR	SPI Data Register							
SPSR	SPIF	WCOL	–	–	–	–	–	SPI2X
SPCR	SPIE	SPE	DORD	MSTR	CPOL	CPHA	SPR1	SPR0
UDR	USART I/O Data Register							
USR	RXC	TXC	UDRE	FE	OR			
UCR	RXCIE	TXCIE	UDRIE	RXEN	TXEN	CHR9	RXB8	TXB8
UBRR	USART Baud Rate Register Low Byte							
ACSR	ACD		ACO	ACI	ACIE	ACIC	ACIS1	AC IS0
ADMUX						MUX2	MUX1	MUX0
ADCSR	ADEN	ADSC	ADFR	ADIF	ADIE	ADPS2	ADPS1	ADPS0
ADCH	ADC Data Register High Byte							
ADCL	ADC Data Register Low Byte							

A.7 AVR Instruction Set

Arithmetic and Logic Instructions

Mnemonic		Description	Operation	Flags
ADD	Rd, Rr	Add two Registers	Rd ← Rd + Rr	Z,C,N,V,H
ADC	Rd, Rr	Add with Carry two Registers	Rd ←Rd + Rr + C	Z,C,N,V,H
ADIW	Rdl.K	Add Immediate to Word	Rdh:Rdl ← Rdh:Rdl + K	Z,C,N,V,S
SUB	Rd, Rr	Subtract two Registers	Rd ← Rd − Rr	Z,C,N,V,H
SUBI	Rd, K	Subtract Constant from Register	Rd ← Rd − K	Z,C,N,V,H
SBC	Rd, Rr	Subtract with Carry two Registers	Rd ← Rd − Rr − C	Z,C,N,V,H
SBCI	Rd, K	Subtract with Carry Constant from Reg.	Rd ← Rd − K − C	Z,C,N,V,H
SBIW	Rdl.K	Subtract Immediate from Word	Rdh:Rdl←Rdh:Rdl −,K	Z,C,N,V,S
AND	Rd, Rr	Logical AND Registers	Rd ← Rd · Rr	Z,N,V
ANDI	Rd,K	Logical AND Register and Constant	Rd ← Rd · K	Z,N,V
OR	Rd, Rr	Logical OR Registers	Rd ← Rd v Rr	Z,N,V
ORI	Rd, K	Logical OR Register and Constant	Rd ← Rd v K	Z,N,V
EOR	Rd, Rr	Exclusive OR Reqisters	Rd ← Rd 0 Rr	Z,N,V
COM	Rd	One’s Complement	Rd ← 0xFF − Rd	Z,C,N,V
NEG	Rd	Two’s Complement	Rd ← 0x00 − Rd	Z,C,N,V,H
SBR	Rd,K	Set Bit(s) in Register	Rd ← Rd v K	Z,N,V
CBR	Rd,K	Clear Bit(s) in Register	Rd ← Rd (0Xff − K)	Z,N,V
INC	Rd	Increment	Rd ← Rd + 1	Z,N,V
DEC	Rd	Decrement	Rd ← Rd − 1	Z,N,V
TST	Rd	Test for Zero or Minus	Rd ← Rd · Rd	Z,N,V
CLR	Rd	Clear Register	Rd ← Rd 0 Rd	Z,N,V
SER	Rd	Set Register	Rd ← 0xFF	None

Branch Instructions

Mnemonic		Description	Operation	Flags
RJMP	k	Relative Jump	PC ← PC + k + 1	None
IJMP		Indirect Jump to (Z)	PC ← Z	None
RCALL	k	Relative Subroutine Call	PC ← PC + k + 1	None
ICALL		Indirect Call to (Z)	PC ← Z	None
RET		Subroutine Return	PC ← STACK	None
RETI		Interrupt Return	PC ← STACK	I
CPSE	Rd,Rr	Compare, Skip if Equal	If(Rd = r)PC ← PC + 2 or 3	None
CP	Rd,Rr	Compare	Rd − Rr	Z,N,V,C,H
CPC	Rd,Rr	Compare with Carry	Rd − Rr − C	Z,N,V,C,H
CPI	Rd,K	Compare Register with Immediate	Rd − K	Z,N,V,C,H
SBRC	Rr, b	Skip if Bit in Reqister Cleared	If(Rr(b) = 0)PC ← PC + 2 or 3	None
SBRS	Rr, b	Skip if Bit in Registeris Set	if(Rr(b) = 1)PC ← PC + 2 or 3	None
SBIC	P,b	Skip if Bit in I/O Register Cleared	if(P(b) = 0)PC ← PC + 2 or 3	None
SBIS	P,b	Skip if Bit in I/O Register is Set	if (P(b) = 1)PC ← PC + 2 or 3	None
BRBS	s,k	Branch if Status Flag Set	if (SREG(s) = 1) then PC ← PC + k + 1	None
BRBC	s,k	Branch if Status Flag Cleared	if (SREG(s) = 0) then PC ← PC + k + 1	None
BREQ	K	Branch if Equal	if (Z = 1) then PC ← PC + k + 1	None
BRNE	K	Branch if Not Equal	if (Z = 0) then PC ← PC + k + 1	None
BRCS	K	Branch if Carry Set	if (C = 1) then PC ← PC + k + 1	None
BRCC	K	Branch if Carry Cleared	if (C = 0) then PC ← PC + k + 1	None
BRSH	K	Branch if Same or Higher	if (C = 0) then PC ← PC + k + 1	None
BRLO	K	Branch if Lower	if (C = 1) then PC ← PC + k + 1	None
BRMI	K	Branch if Minus	if (N = 1) then PC ← PC + k + 1	None
BRPL	K	Branch if Plus	if (N = 0) then PC ← PC + k + 1	None

Mnemonic		Description	Operation	Flags
BRGE	K	Branch if Greater or Equal, Signed	if (N 0 V = 0) then PC ← PC + k + 1	None
BRLT	K	Branch if Less Than Zero, Signed	if (N 0 V = 1) then PC ← PC + k + 1	None
BRHS	K	Branch if Half Carry Flag Set	if (H = 1) then PC ← PC + k + 1	None
BRHC	K	Branch if Half Carry Flag Cleared	if (H = 0) then PC ← PC + k + 1	None
BRTS	K	Branch if T Flag Set	if (T = 1) then PC ← PC + k + 1	None
BRTC	K	Branch if T Flag Cleared	if (T = 0) then PC ← PC + k + 1	None
BRVS	K	Branch if Overflow Flag is Set	if (V = 1) then PC ← PC + k + 1	None
BRVC	k	Branch if Overflow Flag is Cleared	if (V = 0) then PC ← PC + k + 1	None
BRIE	k	Branch if Interrupt Enabled	if (I = 1) then PC ← PC + k + 1	None
BRID	k	Branch if Interrupt Disabled	if (l = 0) then PC ← PC + k + 1	None

Data Transfer Instructions

Mnemonic		Description	Operation	Flags
MOV	Rd, Rr	Move Between Registers	Rd ← Rr	None
LDI	Rd, K	Load Immediate	Rd ← K	None
LD	Rd, X	Load Indirect	Rd ← (X)	None
LD	Rd, X +	Load Indirect and Post-Inc.	Rd ← (X), X ← X + 1	None
LD	Rd, −X	Load Indirect and Pre-Dec.	X← X − 1, Rd ← (X)	None
LD	Rd, Y	Load Indirect	Rd ← (Y)	None
LD	Rd, Y +	Load Indirect and Post-Inc.	Rd ← (Y), Y← Y + 1	None
LD	Rd, −Y	Load Indirect and Pre-Dec.	Y←Y − 1,Rd ← (Y)	None
LDD	Rd,Y + q	Load Indirect with Displacement	Rd ← (Y + q)	None
LD	Rd, Z	Load Indirect	Rd ← (Z)	None
LD	Rd, Z +	Load Indirect and Post-Inc.	Rd ← (Z),Z ← Z + 1	None
LD	Rd, −Z	Load Indirect and Pre-Dec.	Z ← Z − 1,Rd ← (Z)	None
LDD	Rd, Z + q	Load Indirect with Displacement	Rd ← (Z + q)	None
LDS	Rd, k	Load Direct from SRAM	Rd ← (k)	None
ST	X,Rr	Store indirect	(X) ← Rr	None
ST	X + , Rr	Store indirect and Post-Inc	(X) ← Rr, X ← X + 1	None
ST	−X, Rr	Store indirect and Pre-Dec	X ← X − 1,(X) ← Rr	None
ST	Y, Rr	Store indirect	(Y) ← Rr	None
ST	Y + , Rr	Store indirect and Post-Inc	(Y) ← Rr, Y ← Y + 1	None
ST	−Y, Rr	Store indirect and Pre-Dec	Y ← Y − 1,(Y) ← Rr	None
STD	Y + q,Rr	Store indirect with Displacement	(Y + q) ← Rr	None
ST	Z, Rr	Store indirect	(Z) ← Rr	None
ST	Z + , Rr	Store indirect and Post-Inc	(Z) ← Rr, Z ← Z + 1	None
ST	−Z, Rr	Store indirect and Pre-Dec	Z ← Z − 1,(Z) ← Rr	None
STD	Z + q,Rr	Store indirect with Displacement	(Z + q)← Rr	None
STS	k, Rr	Store Direct to SRAM	(k) ← Rr	None
LPM		Load Program Memory	R0 ← (Z)	None
IN	Rd, P	In Port	RdˆP	None
OUT	P,Rr	Out Port	P ← Rr	None
PUSH	Rr	Push Register on Stack	STACK ← Rr	None
POP	Rd	Pop Register from Stack	Rd ← STACK	None

Bit and Bit-test Instructions

Mnemonic		Description	Operation	Flags
SBI	P,b	Set Bit in I/O Register	I/O(P,b) ← 1	None
CBI	P,b	Clear Bit in I/O Register	I/O(P,b) ← 0	None
LSL	Rd	Logical Shift Left	Rd(n + 1) ← Rd(n),Rd(0) ← 0	Z,C,N,V
LSR	Rd	Logical Shift Right	Rd(n) ← Rd(n + 1), Rd(7) ← 0	Z,C,N,V
ROL	Rd	Rotate Left Through Carry	Rd(0) ← C,Rd(n + 1) ← Rd(n),C ← Rd(7)	Z,C,N,V
ROR	Rd	Rotate Right Through Carry	Rd(7) ← C,Rd(n) ← Rd(n + 1),C ← Rd(0)	Z,C,N,V
ASR	Rd	Arithmetic Shift Right	Rd(n) ← Rd(n + 1),n = 0..6	Z,C,N,V
SWAP	Rd	Swap Nibbles	Rd(3-0) ← Rd(7–4),Rd(7-4) ← Rd(3–0)	None
BSET	s	Flag Set	SREG(s) ← 1	SREG(s)
BCLR	s	Flaq Clear	SREG(s) ← 0	SREG(s)
BST	Rr, b	Bit Store from Register to T	T ← Rr(b)	T
BLD	Rd, b	Bit load from T to Register	Rd(b) ← T	None
SEC		Set Carry	C ← 1	C
CLC		Clear Carry	C ← O	C
SEN		Set Negative Flag	N ← 1	N
CLN		Clear Negative Flag	N ← O	N
SEZ		Set Zero Flag	Z ← 1	Z
CLZ		Clear Zero Flag	Z ← O	Z
SEI		Global Interrupt Enable	l ← 1	I
CLI		Global Interrupt Disable	l ← 0	I
SES		Set Signed Test Flag	S ← 1	S
CLS		Clear Signed Test Flag	S ← O	S
SEV		Set V	V ← 1	V
CLV		Clear V	V ← O	V
SET		Set T in SREG	T ← 1	T
CLT		Clear T in SREG	T ← O	T
SEH		Set Half Carry Flag in SREG	H ← 1	H
CLH		Clear Half Carry Flag in SREG	H ← O	H

Special Instructions

Mnemonic	Description	Operation	Flags
NOP	No Operation		None
SLEEP	Sleep		None
WDR	Watchdog Reset		None

A.8 AT90S8515 Operating with External RAM

Some AVR microcontrollers can allocate some of the I/O lines to extend the bus of the data memory for connecting external memory or other RAM-like external devices. Figure A8.1 presents an example of implementation of the external bus for the microcontroller AT90S8515.

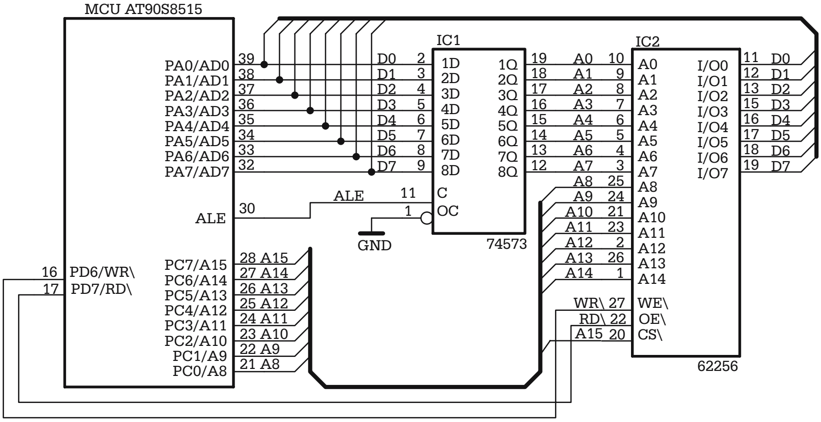


Fig. A8.1. AT90S8515 operating with external bus for data memory

In this example, the external memory is a 32K×8 62256 circuit, selected directly with the address line A15, which makes it visible in the address range $0000–$7FFF in the address space of the data memory.

The control signals for the external bus are generated by the MCU. ALE (Address Latch Enable) is active HIGH, and strobes the lower half of the address into the external address latch IC1 (74LS573), while RD\ (Read) and WR\ (Write) are active LOW and indicate the direction of the data transfer on the external bus. They are directly connected to the control inputs OE\ and WR\ of the external RAM circuit IC2.

A.9 In-system Programming the AVR AT90S8535

The internal flash and EEPROM memory of many of the AVR microcontrollers can be programmed “in system”, without removing the circuit from its socket, through the ISP (In-System Programming) interface. At the hardware level, the ISP interface uses the SPI lines MOSI, MISO and SCK to transfer data between the host and the device to be programmed. The ISP interface takes control of the SPI lines when the RESET input of the MCU is LOW.

Atmel Corporation offers a free software utility, called AVRISP, which allows the use of a personal computer to program the AVR microcontrollers through the ISP. Although the data transfer through the ISP is serial, the ISP interface is connected to the parallel port of the PC by means of a cable adapter. One possible implementation of the ISP cable adapter for the parallel port is presented in Fig. A9.1.

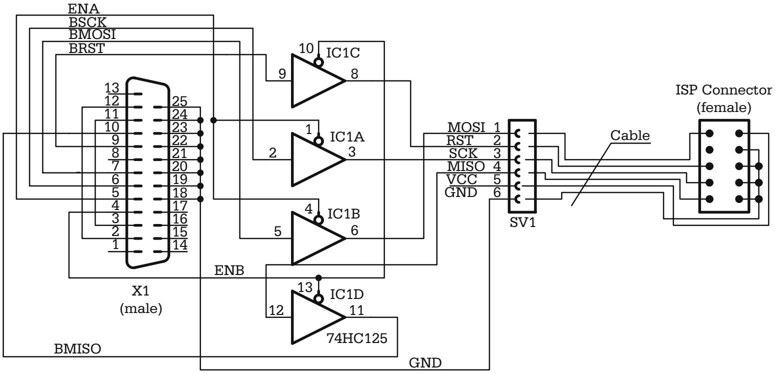


Fig. A9.1. Schematic of the cable adapter between the parallel port of PC and the ISP

Figure A9.2 shows the layout of the PCB for this circuit. Note that the circuit is powered through the ISP cable.

For the ISP to operate, the MCU must be powered, and an external crystal must be present, for the oscillator. The XTAL frequency must be at least twice the frequency of the data transmission clock SCK.

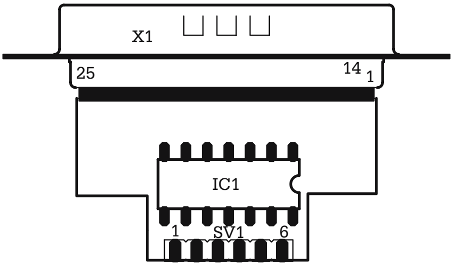


Fig. A9.2. Layout of the PCB for the ISP cable adapter

The ISP protocol requires that 4-byte data packets be transferred through on the interface for each command. For example, the command consisting of the series of bytes [$AC, $53, xx, xx] (xx are "don't care" bytes) instructs the MCU to enter programming mode. During the transmission of the third byte, the MCU echoes the value $53 received as the second byte of the packet to acknowledge the command.

The command to read a byte from the program memory has the following structure:

0010 H000	xxxx aaaa	bbbb bbbb	oooo oooo

H indicates which byte of the location of program memory specified by the bits aaaabbbbbbbb must be read. H = 1 addresses the most significant byte, H = 0 refers to the least significant byte. During the transmission of the fourth byte – oooooooo – the MCU returns the value read from the specified address of the flash memory.

The full list of commands of the ISP protocol is presented in Table A9.3.

To prevent unintentional programming of the flash or EEPROM memory while using the SPI interface, a special protection mechanism, called "lock bits", has been provided. Lock bits are non-volatile bits, accessible only in programming mode, that inhibit further access to the EEPROM or flash memory in programming mode. AT90S8535 has two lock bits, LB1 and LB2, with the functions listed in Table A9.2.

Table A9.1. Functions of the lock bits of AT90S8535

Lock bits		Protection type
LB1	LB2	
1	1	No memory lock features enabled
0	1	Further programming of Flash and EEPROM disabled
0	0	Verify is also disabled

Once programmed with 0, the lock bits can only be brought to the erase value 1 through a chip erase operation.

Fuse bits are non-volatile control bits similar to the lock bits but with different functions. There are two fuse bits named SPIEN and FSTRT.

SPIEN – Serial Programming Enable. Its default value is 0 (enabled). This bit cannot be modified through ISP.

FSTRT – Short Start-up Time determines the duration of the internal RESET pulse, so that the external oscillator can settle. The operation of the watchdog timer is also delayed with a programmable number of cycles, as shown in Table A9.3.

When any of the lock bits is programmed (i.e. have the value 0) the access to the fuse bits is inhibited. Therefore, the fuse bits must be programmed first.

The ISP command *Chip Erase* brings the lock bits to the erased value (1) but does not affect the fuse bits.

Table A9.2. The effect of the fuse bit FSTRT for AT80S8535

FSTRT	Internal RESET duration at $V_{CC} = 5\,V$	Number of WDT cycles
Programmed	1.1 ms	1 K
Unprogrammed	16.0 ms	16 K

The *signature bytes* are three special read-only bytes that indicate the manufacturer, the size of the program memory and the device type. They are used by programmers to identify the devices. See the specific data sheets for the values of the signature bytes. When both the lock bits are programmed (0) the signature bytes are no longer accessible through the ISP interface.

Table A9.3. List of the ISP protocol commands

Command	Command format			
Programming Enable	1010 1100	0101 0011	xxxx xxxx	xxxx xxxx
Chip Erase	1010 1100	100x xxxx	xxxx xxxx	xxxx xxxx
Read Program Memory	0010 H000	xxxx aaaa	bbbb bbbb	oooo oooo
Write Program Memory	0100 H000	xxxx aaaa	bbbb bbbb	dddd dddd
Read EEPROM Memory	1010 0000	xxxx xxxa	bbbb bbbb	oooo oooo
Write EEPROM Memory	1100 0000	xxxx xxxa	bbbb bbbb	dddd dddd
Read Lock and Fuse Bits	0101 1000	xxxx xxxx	xxxx xxxx	12Sx xxxF
Write Lock Bits	1010 1100	1111 1211	xxxx xxxx	xxxx xxxx
Read Signature Byte	0011 0000	xxxx xxxx	xxxx xxbb	oooo oooo
Write FSTRT Fuse	1010 1100	1011 111F	xxxx xxxx	xxxx xxxx

Legend:
a = address high bits; **b** = address low bits; **H** = 0 – low byte, 1 – high byte; **o** = data out; **d** = data in; x = don't care bit; ***1*** = lock bit 1; ***2*** = lock bit 2; **F** = FSTRT fuse; **S** = SPIEN fuse

A.10 The Special Function Registers of 8051

ACC	A.7	A.6	A.5	A.4	A.3	A.2	A.1	A.0
B	B.7	B.6	B.5	B.4	B.3	B.2	B.1	B.0
PSW	CY	AC	F0	RS1	RS0	OV	–	P
SP	Stack Pointer							
DPH	Data pointer – high byte							
DPL	Data pointer – low byte							
P0	P0.7	P0.6	P0.5	P0.4	P0.3	P0.2	P0.1	P0.0
P1	P1.7	P1.6	P1.5	P1.4	P1.3	P1.2	P1.1	P1.0
P3	P3.7	P3.6	P3.5	P3.4	P3.3	P3.2	P3.1	P3.0
IP	–	–	PT2	PS	PT1	PX1	PT0	PX0
IE	EA	–	ET2	ES	ET1	EX1	ET0	EX0
TMOD	GATE	C/T	M1	M0	GATE	C/T	M1	M0
TCON	TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0
TH0	Timer0 data – high byte							
TL0	Timer0 data – low byte							
TH1	Timer1 data – high byte							
TL1	Timer1 data – low byte							
SCON	SM0	SM1	SM2	REN	TB8	RB8	TI	RI
SBUF	Serial data buffer							
PCON	SMOD	–	–	–	GF1	GF0	PD	IDL

A.11 8051 Instruction Set

Arithmetic Instructions

Mnemonic		Operation	Description	Flags
ADD	A,R_n	Add register to Accumulator	$A \leftarrow A + R_n$	C,OV,AC
ADD	A.direct	Add direct byte to Accumulator	$A \leftarrow A +$ direct	C,OV,AC
ADD	A,@R_i	Add indirect RAM to Accumulator	$A \leftarrow A + (R_i)$	C,OV,AC
ADD	A,#data	Add immediate data to Accumulator	$A \leftarrow A +$ data	C,OV,AC
ADDC	A,R_n	Add register to Accumulator with Carry	$A \leftarrow A + R_n + C$	C,OV,AC
ADDC	A.direct	Add direct byte to Accumulator with Carry	$A \leftarrow A +$ direct $+ C$	C,OV,AC
ADDC	A,@R_i	Add indirect RAM to Accumulator with Carry	$A \leftarrow A + (R_i) + C$	C,OV,AC
ADDC	A,#data	Add immediate data to Acc with Carry	2 $A \leftarrow A +$ data $+ C$	C,OV,AC
SUBB	A,R_n	Subtract Register from Acc with borrow	$A \leftarrow A - R_n - C$	C,OV,AC
SUBB	A.direct	Subtract direct byte from Acc with borrow	$A \leftarrow A -$ direct $- C$	C,OV,AC
SUBB	A,@R_i	Subtract indirect RAM from Acc with borrow	$A \leftarrow A - (R_i) - C$	C,OV,AC
SUBB	A,#data	Substract immediate data from Acc with borrow	$A \leftarrow A -$ data $- C$	C,OV,AC
INC	A	Increment Accumulator	$A \leftarrow A + 1$	
INC	R_n	Increment register	$R_n \leftarrow R_n + 1$	
INC	Direct	Increment direct byte	direct $\leftarrow$ direct $+ 1$	
INC	@R_i	Increment direct RAM	$(R_i) \leftarrow (R_i) + 1$	
DEC	A	Decrement Accumulator	$A \leftarrow A - 1$	
DEC	R_n	Decrement Register	$R_n \leftarrow R_n - 1$	
DEC	Direct	Decrement direct byte	direct $\leftarrow$ direct $- 1$	
DEC	@R_i	Decrement indirect RAM	$(R_i) \leftarrow (R_i) - 1$	
INC	DPTR	Increment Data Pointer	DPTR $\leftarrow$ DPTR $+ 1$	
MUL	AB	Multiply A by B	A_{0-7},$B_{15-8} \leftarrow$ AxB	C,OV
DIV	AB	Divide A by B	A_{0-7},$B_{15-8} \leftarrow A / B$	C,OV
DA	A	Decimal Adjust Accumulator	*see datasheet	C

Logic Instructions

Mnemonic		Operation	Description	Flags
ANL	A,R_n	AND Register to Accumulator	$A \leftarrow A \mathit{\Lambda} R_n$	
ANL	A.direct	AND direct byte to Accumulator	$A \leftarrow A \mathit{\Lambda}$ direct	
ANL	A,@R_i	AND indirect RAM to Accumulator	$A \leftarrow A \mathit{\Lambda} (R_i)$	
ANL	A,#data	AND immediate data to Accumulator	$A \leftarrow A \mathit{\Lambda}$ data	
ANL	direct.A	AND Accumulator to direct byte	direct $\leftarrow$ direct $\mathit{\Lambda}$ A	
ANL	direct,#data	AND immediate data to direct byte	direct $\leftarrow$ direct $\mathit{\Lambda}$ data	
ORL	A,R_n	OR register to Accumulator	$A \leftarrow AV R_n$	
ORL	A.direct	OR direct byte to Accumulator	$A \leftarrow A$ V R_n	
ORL	A,@R_i	OR indirect RAM to Accumulator	$A \leftarrow A$ V (R_i)	
ORL	A,#data	OR immediate data to Accumulator	$A \leftarrow A$ V data	
ORL	direct.A	OR Accumulator to direct byte	direct $\leftarrow$ direct V A	
ORL	direct,#data	OR immediate data to direct byte	direct $\leftarrow$ direct V data	
XRL	A,R_n	Exclusive-OR register to Accumulator	$A \leftarrow A \forall R_n$	
XRL	A.direct	Exclusive-OR direct byte to Accumulator	$A \leftarrow A \forall R_n$	
XRL	A,@R_i	Exclusive-OR indirect RAM to Acc	$A \leftarrow A \forall (R_i)$	
XRL	A,#data	Exclusive-OR immediate data to Accr	$A \leftarrow A \forall$ data	
XRL	direct.A	Exclusive-OR Accumulator to direct byte	direct $\leftarrow$ direct $\forall$ A	
XRL	direct,#data	Exclusive-OR immediate data to direct byte	direct $\leftarrow$ direct $\forall$ data	
CLR	A	Clear Accumulator	$A \leftarrow 0$	
CPL	A	Complement Accumulator	$A \leftarrow A \forall$ \$FF	
RL	A	Rotate Accumulator Left	$A_0 \leftarrow A_7 A_{n+1} \leftarrow A_n A_1 \leftarrow A_0$	
RLC	A	Rotate Acc. Left through the Carry	$A_0 \leftarrow C, A_{n+1} \leftarrow A_n, C \leftarrow A_7$	C

Mnemonic		Operation	Description	Flags
RR	A	Rotate Accumulator Right	$A_7 \leftarrow A_0 A_n \leftarrow A_{n+1} A_6 \leftarrow A_7$	
RRC	A	Rotate Acc. Right through the Carry	$A_7 \leftarrow C, A_n \leftarrow A_{n+1}, C \leftarrow A_0$	C
SWAP	A	Swap nibbles within the Accumulator	$A_{3-0} \leftrightarrow A_{7-4}$	

Data Transfer Instructions

Mnemonic		Operation	Description	Flags
MOV	A,R$_n$	Move register to ccumulator	$A \leftarrow R_n$	
MOV	A.direct	Move direct byte to accumulator	A ← direct	
MOV	A,@R$_i$	Move indirect RAM to Accumulator	$A \leftarrow (R_i)$	
MOV	A,#data	Move immediate data to Accumulator	A ← data	
MOV	R$_n$,A	Move Accumulator to register	$R_n \leftarrow A$	
MOV	R$_n$,direct	Move direct byte to register	R$_n$ ← direct	
MOV	R$_n$,#data	Move immediate data to register	R$_n$ ← data	
MOV	direct,A	Move Accumulator to direct byte	direct ← A	
MOV	direct, R$_n$	Move register to direct byte	direct ← R$_n$	
MOV	direct.direct	Move direct byte to direct	direct ← direct	
MOV	direct,@Ri	Move indirect RAM to direct byte	direct ← (R$_i$)	
MOV	direct,#data	Move immediate data to direct byte	direct ← data	
MOV	@R$_i$,A	Move Accumulator to indirect RAM	$(R_i) \leftarrow A$	
MOV	@R$_i$,direct	Move direct byte to indirect RAM	(R$_i$) ← direct	
MOV	@R$_i$,#data	Move immediate data to indirect RAM	(R$_i$) ← data	
MOV	DPTR,#data$_{16}$	Load Data Pointer with a 16-bit constant	DPTR ← data$_{16}$	
MOVC	A,@A + DPTR	Move Code byte relative to DPTR to Acc	A ← (A + DPTR)	
MOVC	A,@A + PC	Move Code byte relative to PC to Acc	A ← (A + PC)	
MOVX	A,@R$_i$	Move External RAM (8-bit addr) to Acc	$A \leftarrow (R_i)$	
MOVX	A,@DPTR	Move Exernal RAM (16-bit addr) to Acc	A ← (DPTR)	
MOVX	@R$_i$,A	Move Acc to External RAM (8-bit addr)	$(R_i) \leftarrow A$	
MOVX	@DPTR,A	Move Acc to External RAM (16-bit addr)	(DPTR) ← A	
PUSH	Direct	Push direct byte onto stack	STACK ← direct	
POP	Direct	Pop direct byte from stack	direct ← STACK	
XCH	A,R$_n$	Exchange register with Accumulator	$A \leftrightarrow R_n$	
XCH	A.direct	Exchange direct byte with Acc	A ↔ direct	
XCH	A,@R,	Exchange indirect RAM with Acc	A↔(Ri)	
XCHD	A,@R,	Exchange low-order Digit indirect RAM with Acc	$A_{3-0} \leftrightarrow (Ri_{3-0})$	

Bit Manipulation Instructions

Mnemonic		Operation	Description	Flags
CLR	C	Clear Carry	C ← 0	C
CLR	Bit	Clear direct bit	bit ← 0	
SETB	C	Set Carry	C ← 1	C
SETB	Bit	Set direct bit	bit ← 1	
CPL	C	Complement Carry	C ← 1 − C	C
CPL	Bit	Complement direct bit	bit ← 1 − bit	
ANL	C,bit	AND direct bit to CARRY	$C \leftarrow C\ \mathit{\Lambda}$ bit	C
ANL	C,/bit	AND complement of direct bit to Carry	$C \leftarrow C\ \mathit{\Lambda}$ /bit	C
ORL	C.bit	OR direct bit to Carry	C ← C V bit	C
ORL	C,/bit	OR complement of direct bit to Carry	C ← C V /bit	C
MOV	C.bit	Move direct bit to Carry	C ← bit	C
MOV	bit.C	Move Carry to direct bit	bit ← C	C

Branch Instructions

Mnemonic		Operation	Description	Flags
JC	rel	Jump if Carry is set	If C = 1 PC = PC + rel	
JNC	rel	Jump if Carry not set	If C = 0 PC = PC + rel	
JB	bit.rel	Jump if direct Bit is set	If bit = 1 PC = PC + rel	
JNB	bit.rel	Jump if direct Bit is Not set	If bit = 0 PC = PC + rel	
JBC	bit.rel	Jump if direct Bit is set and clear bit	If bit = 1 PC = PC + rel,bit ← 0	

ACALL	addr 11	Absolute Subroutine Call	PC = PC + 2
			SP = SP + 1
			(SP) ← PC$_{7-0}$
			SP = SP + 1
			(SP) ← PC1$_{5-8}$
			PC$_{10-0}$ ← ADDR11	

LCALL	addr 16	Long Subroutine Call	PC = PC + 2
			SP = SP + 1
			(SP) ← PC$_{7-0}$
			SP = SP + 1
			(SP) ← PC1$_{5-8}$
			PC$_{15-0}$ ← ADDR16	

RET		Return from Subroutine	PC1$_{5-8}$ ← (SP)
			SP = SP − 1
			PC$_{7-0}$ ← (SP)
			SP = SP + 1	

RETI		Return from interrupt	PC1$_{5-8}$ ← (SP)
			SP = SP − 1
			PC$_{7-0}$ ← (SP)
			SP = SP + 1
			EA ← 1	

AJMP	addr11	Absolute Jump	PC = PC + 2 PC$_{10-0}$ ← addr11	
LJMP	addr16	Long Jump	PC$_{15-0}$ ← addr16	

SJMP	rel	Short Jump (relative addr)	PC = PC + 2
			PC ← PC + rel	

JMP	@A + DPTR	Jump indirect relative to the DPTR	PC ← A + DPTR	

JZ	rel	Jump if Accumulator is Zero	if A = 0
			PC = PC + 2
			PC ← PC + rel	

JNZ	rel	Jump if Accumulator is Not Zero	if A ≠ 0
			PC = PC + 2
			PC ← PC + rel	

CJNE	A,direct,rel	Compare direct byte to Acc and Jump if Not Equal	if A ≠ direct
			PC = PC + 2
			PC ← PC + rel
				C

CJNE	A,#data,rel	Compare immediate to Acc and Jump if Not Equal	if A ≠ data
			PC = PC + 2
			PC ← PC + rel
				C

CJNE	R$_n$,#data,rel	Compare immediate to register and Jump if Not Equal	if R$_n$ ≠ data
			PC = PC + 2
			PC ← PC + rel
				C

CJNE	@R$_i$,#data,rel	Compare immediate to indirect and Jump if Not Equal	if (R$_I$) ≠ data
			PC = PC + 2
			PC ← PC + rel
				C

Mnemonic		Operation	Description	Flags
DJNZ	R_n,rel	Decrement register and Jump if Not Zero	$PC = PC + 2$ $R_n \leftarrow R_n - 1$ if $R_n \neq 0$ $PC = PC + rel$	

DJNZ	direct,rel	Decrement direct byte and Jump if Not Zero	$PC = PC + 2$
			direct ← direct − 1
			if direct ≠ 0
			$PC = PC + rel$	

Index *n* may take values in the range [0,7]. Index *i* may take values in the range [0,1].

A.12 An Example of 8051 Operating with External Bus

Figure A12.1 shows an example of a typical structure of 8051 operating with an external bus.

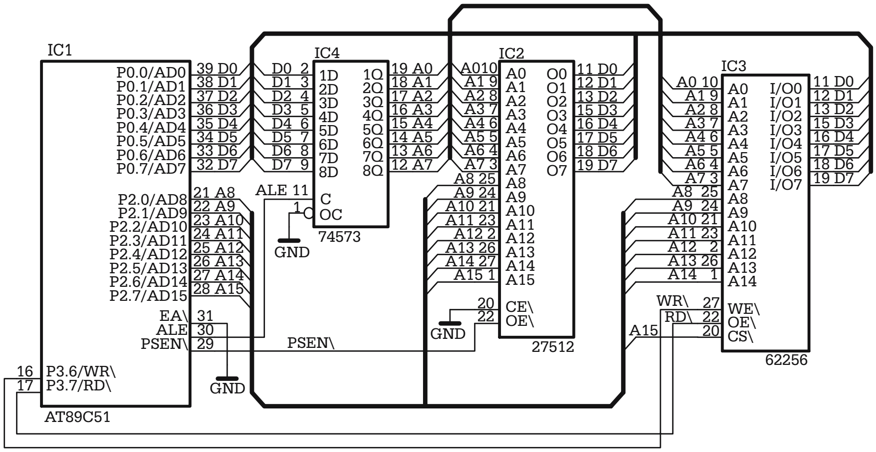


Fig. A12.1. 8051 operating with external bus

The EA\ signal is grounded, which means that the MCU ignores the internal program memory, if any.

A.13 Programming the Internal Memory of 8051

The waveforms of the signals involved in the process of programming the internal memory of 8051 are shown in Fig. A13.1.

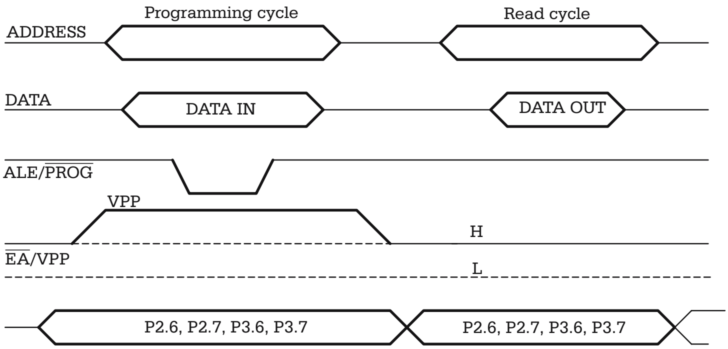


Fig. A13.1. Waveforms for the signals involved in programming 8051 memory

The information in this section concerns those versions of 8051 microcontrollers that have internal EPROM or flash memory. Both EPROM (87C51) and flash (89C51) versions are programmable following the same principles. Only the value of the programming voltage V_{pp} differs. *Note that any attempt to program the internal memory using inappropriate* V_{pp} *may cause permanent damage to the chip. See the specific data sheets for the exact requirements for* V_{pp}.

Table A13.1 presents the status of the control signals for each particular operation. Besides these signals, during any program/verify operations, RST must be HIGH and PSEN\ must be LOW.

Table A13.1. Status of the control signals for program/read memory operations

Mode	MCU Pins					
	ALE/PROG	EA/VPP	P2.6	P2.7	P3.6	P3.7
READ MEMORY	H	H	L	L	H	H
WRITE MEMORY	H-L-H	VPP	L	H	H	H
READ SIGNATURE	H	H	L	L	L	L
ERASE MEMORY*[1]	H-L-H*[2]	VPP	H	L	L	L
WRITE LOCK BIT1	H-L-H	VPP	H	H	H	H
WRITE LOCK BIT2	H-L-H	VPP	H	H	L	L
WRITE LOCK BIT3	H-L-H	VPP	H	L	H	L

Legend:

*[1] – Only applicable for flash versions (89xxx series).

*[2] – Chip erase operations require 10 ms PROG pulse. H-L-H indicates a neagative pulse of 100 s, except the ERASE MEMORY operation, where this pulse must be 10 ms wide.

The signature bytes are two to four data bytes, read from addresses specific for each device. For example, AT89C51 reports three signature bytes, read from the addresses 30h, 31h, 32h. The first signature byte indicates the manufacturer (1Eh for Atmel Corporation), the second byte identifies the device (51h for AT89C51), and the third signature byte indicates the value of the programming voltage V_{pp} (FFh for 12 V, 05h for 5 V).

The lock bits are non-volatile control bits, accessible only in programming mode, similar to those described in Appendix A9 for AVR microcontrollers. The effect of programming the lock bits of AT89C51 is described in Table A13.2.

Table A13.2. The effect of the lock bits for AT89C51

Lock bits status			Protection type
LB1	LB2	LB3	
U	U	U	No program lock features
P	U	U	The EA line is sampled and latched on reset, and further programming of the Flash is disabled. MOVC instructions executed from external program memory are disabled.
P	P	U	Same as above, but verify operations are also disabled
P	P	P	Same as above, but program execution from external memory is also disabled.

Legend
U – Unprogrammed bit (1)
P – Programmed bit (0)

See the accompanying CD for details on how to build a programmer for 8051 microcontrollers.

A.14 SPI Seven-Segment Display Units

This section presents schematics and suggestions on how to use two types of seven-segment display units, both designed to be connected to the SPI interface of a microcontroller.

The first implementation, presented in Fig. A14.1, is multiplexed, i. e. at a certain time moment only one digit is displayed. The software must refresh the data permanently, by switching the active digit fast enough so that the human eye integrates the successive images into a single, complete perception.

The SPI interface of the MCU is brought to the connector SV1. An additional connector SV2 has been provided so that the ISP interface can be used without the need to remove the display unit.

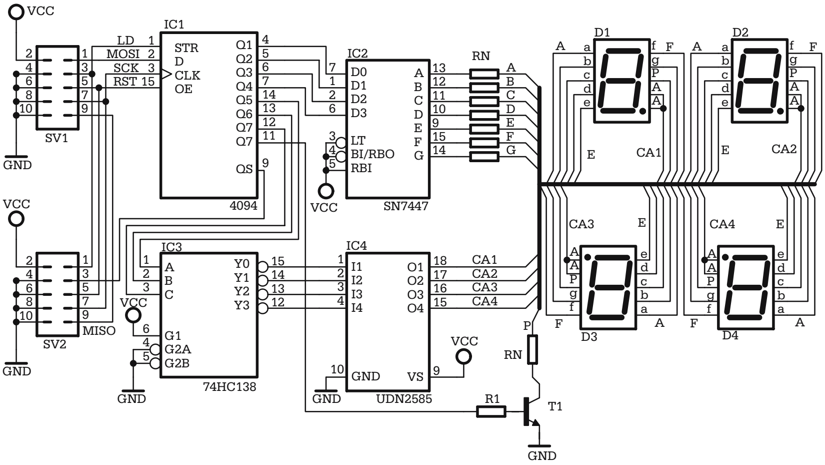


Fig. A14.1. Schematic of a multiplexed seven-segment display

Data from the SPI is received in the shift register 4094 (IC1) and must be prepared by the software as follows:

- The least significant four bits of each byte contain the BCD value of the digit to be displayed.
- The next three bits contain the address of the active digit.
- The most significant bit, if set to 1, activates the decimal point of the corresponding display digit.

4094 receives the serial data sent by the MCU through the MOSI line and shifts it into the internal shift register with the clock SCK. When the serial transfer completes, the software must generate a positive pulse on the LD line so that the data received this way is presented on the output lines Q1–Q8. The SPI must be programmed to send data with the most significant bit first.

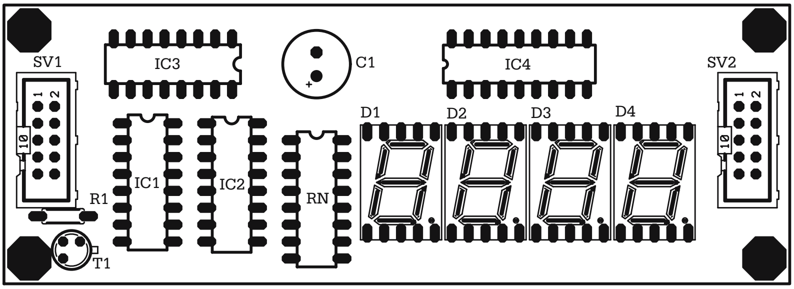


Fig. A14.2. PCB layout for the circuit presented in Fig. 14.1

BCD data is directly applied to the input of the 7447 decoder IC2, while the digit address is decoded by the 74138 circuit IC3 and activates one of the four common anodes of the display, by means of the inverting driver UDN2585 (IC4). The transistor T1 activates the decimal point of the active digit if the most significant bit of the byte in IC1 is set to 1.

Figure A14.2 shows the component layout of the PCB for this display unit.

Another approach for displaying four seven-segment digits with a SPI display is presented in Fig. A14.3.

This time, two data bytes are received in two 4094 registers. The contents of the 4094 registers are interpreted as four BCD digits and applied to 4×7447 circuits, which directly drive the display units.

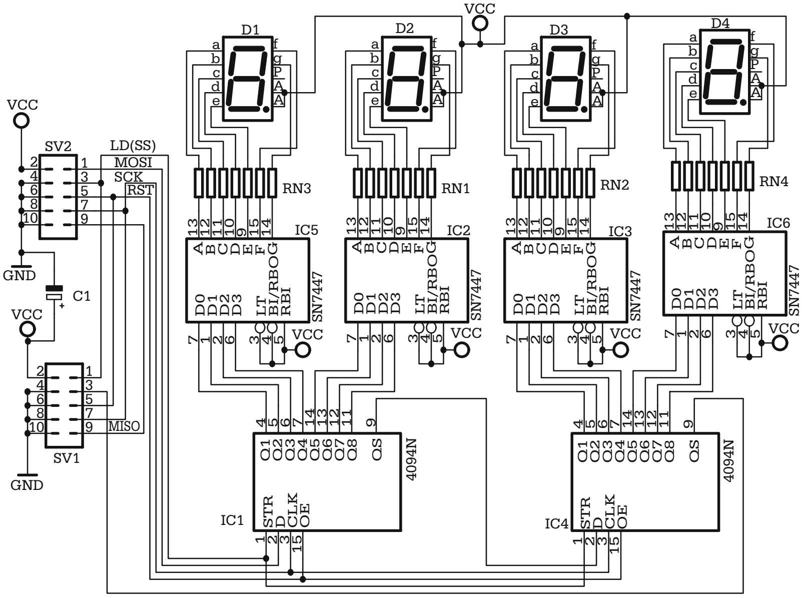


Fig. A14.3. Non-multiplexed seven-segment display unit

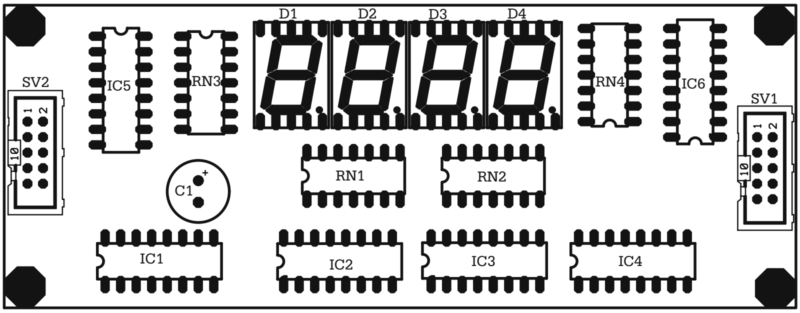


Fig. A14.4. PCB layout for the circuit presented in Fig. A14.3

Note that in this case the two 4094 registers are concatenated, which means that the software must send two bytes of data over the SPI interface. The two bytes must contain four BCD digits (the least significant digit is sent first), followed by a single pulse on LD, which is common for both registers.

This display unit maintains the data indefinitely and there is no need to refresh the information displayed.

The non-multiplexed display requires simpler software, but, for a larger number of digits, the cost of the hardware is higher. Besides that, this solution has the disadvantage that it cannot dynamically change by software the position of the decimal point.

T he PCB layout for this circuit is shown in Fig. A14.4. For all the projects that require a display unit, use the cable described in Fig. A14.5 between the development board and the display, regardless of the type of display (multiplexed or non-multiplexed) used.

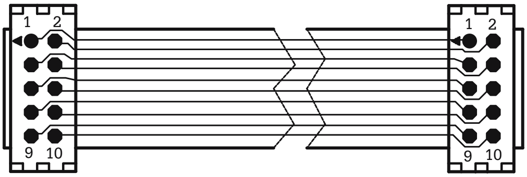


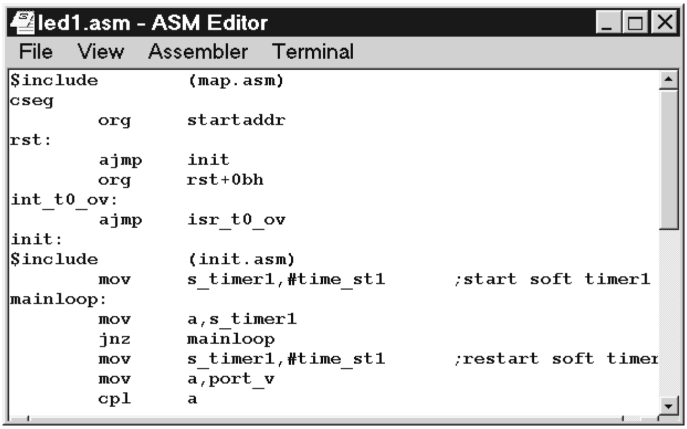
Fig. A14.5. Layout of the cable for connecting the display unit

A.15 Description of the Software Utility ASMEDIT

Some users, familiar with the Windows™ environment, might find it difficult to work with MS-DOS™ type applications, which must be invoked from the command line in a DOS window. Since most freeware assemblers available, including those recommended in this book for HC11 and 8051, are DOS applications, we wrote a small utility program called ASMEDIT.EXE with the following features:

- It contains a simple built-in text editor, which allows the user to create assembler source files, save them to disk, or open existing files for editing.
- The user can choose an assembler, define command line options and switches, if required, then invoke the assembler and run it on the selected source file. When the assembly process completes, ASMEDIT brings the list file generated by the assembler in the main window for viewing.
- ASMEDIT also contains a built-in terminal program capable of using one of the computer's COM ports to communicate with the development boards described in this book through the RS232 interface.

The main window presented by ASMEDIT when launched is shown in Fig. A15.1. The following pull-down menus are available:



```
$include        (map.asm)
cseg
        org     startaddr
rst:
        ajmp    init
        org     rst+0bh
int_t0_ov:
        ajmp    isr_t0_ov
init:
$include        (init.asm)
        mov     s_timer1,#time_st1      ;start soft timer1
mainloop:
        mov     a,s_timer1
        jnz     mainloop
        mov     s_timer1,#time_st1      ;restart soft timer
        mov     a,port_v
        cpl     a
```

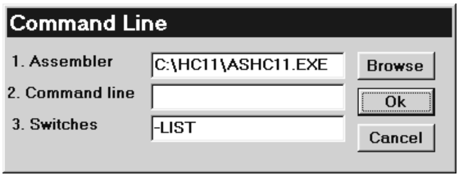
Fig. A15.1. Snapshot of the main window of ASMEDIT

File. This menu presents options for Open, Save, Save as, and Create New source file. If you select the option Open file, ASMEDIT filters only files with the extensions .ASM and .A51.

View. This menu allows the user to select the active window between the source file and the list file. It is also possible to adjust the font size.

Assembler. This pull down menu has only two menu options: Assemble, which runs the assembler for the selected source file, and Select assembler. The dialog window presented in this case is shown in Fig. A15.2. Select the desired executable file by clicking the Browse button, then fill in the command line options, and switches, according to the syntax required by the assembler. Make sure that the assembler is set to generate a list file. For example, the case of the assembler ASHC11 by Peter Gargano, referred in Chap. 9, requires the switch ––LIST to generate the list file.

Terminal. This pull-down menu displays a list of options concerning the built-in terminal of ASMEDIT. The user can select the communication port (default is COM2), set the communication speed and clear the terminal window. In addition to this, the terminal menu contains two options that are useful when using the 8051 development board described in Chap. 11. If the option Send is activated, ASMEDIT looks in the folder where the active source file is located for a file with the same name and the extension .HEX, loads the file and sends it to the development board. The option Send&Go automatically launches a G<aaaa> command to the board, when download completes, where <aaaa> is the starting address of the program, extracted from the HEX file.



1. Assembler C:\HC11\ASHC11.EXE
2. Command line
3. Switches -LIST

Fig. A15.2. Snapshot of the menu option Select Assembler

The output files generated by the assembler are placed in the same folder as the specified source file. If the source file uses the INCLUDE directive, the include files must be present in the same folder, or the full path where they are located must be indicated in the source file, e. g.

```
INCLUDE C:\ASHC11\INCLUDE\68HC11F1.DEF
```

B.1 Contents of the Accompanying CD

The main purpose of the accompanying CD is to allow the reader to save the time needed for typing the many software examples and exercises presented in the book. In addition to this, the fact that a freeware version of the Eagle layout editor is available for download at the CadSoft web site encouraged us to include on the CD the schematics and the PCB design for all the projects presented.

On several occasions, testing the projects in the book required special software utilities, which are also included on the CD.

The CD follows the structure of the book, each chapter in the book has a corresponding folder on the CD, and the subfolders Examples, Exercises, Schematics, and Utils. See Fig. B1.1 for an image of how the CD is organized.

To assemble and test the examples, copy the source files (.ASM) from the CD into the working directory for the selected microcontroller, e. g. C:\ASHC11\WORKor C:\ASM51\WORK, remove the read-only attributes so that they can be edited, then run the appropriate assembler.

If the Eagle layout editor is installed, the schematic files (.SCH) and the PCB design files (.BRD) can be opened simply by clicking on the file name. In case you choose not to install Eagle, the CD also contains a high-resolution bitmap version of the schematics presented in the book.

The software utilities included on the CD are described in the book. They do not require any installation procedure; just copy the files into a folder on your hard

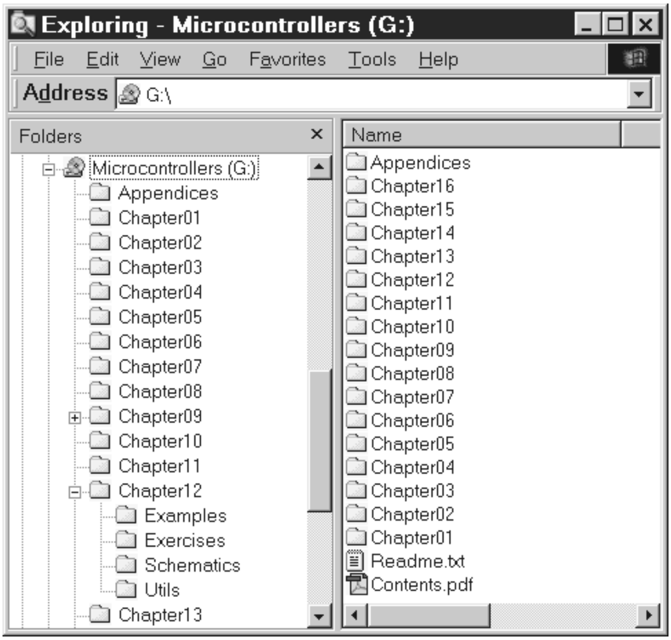


Fig. B1.1. Structure of the accompanying CD

drive and click on the program’s name to run it. All the utilities included need VB40032.DLL. This file is not included. If you don’t have it already, try a search on Google with this keyword to locate and download it, then copy it into your WINDOWS\SYSTEMdirectory.

All the software included or recommended in this book has been tested on several computers under Windows98™ and Windows XP™. We don’t know whether or not it works on other operating systems.

B.2 Recommended Readings and Web References

The information in this book is not intended to replace the data sheets for the microcontrollers presented herein. It is strongly recommended that you consult the data sheets and the application notes referred to in the book, and to download and install the software applications and utilities referred to for each microcontroller discussed, in order to be able to test the software examples presented.

Below is a list of the internet web sites that offer technical documentation and software for the study of the microcontrollers discussed in this book.

The best source for documentation about Motorola 68HC11 microcontrollers is the official web site of the manufacturer at www.freescale.com. For in-depth understanding of the information in this book, download and consult the data sheets for 68HC11E9, 68HC11F1 and the application note AN1060 that contains information about the special bootstrap operating mode.

The freeware assembler recommended for HC11 can be downloaded from http://www.techedge.com.au/utils/ashc11.htm . It is called ASHC11 and was created by Peter Gargano.

For other HC11 related internet resources, visit the web page of Roger Schaefer, at http://www.ezl.com/~rsch/.

Atmel Corporation has an excellent web site at www.atmel.com, offering technical data and software applications to support their products. Download the data sheets for the AVR microcontrollers AT90S8515/8535, and the software applications AVRStudio, and AVR ISP.

Also visit www.avrfreaks.com for further information about AVR microcontrollers.

Good-quality 8051 documentation can be found on the Atmel web site that offers information about their series of microcontrollers with the 8051 architecture.

The freeware 8051 assembler referred to in Chap. 11 can be found at http://plit.de/asem-51/ and was created by W. Heinz.

Technical information about all types of memory devices is available at www.atmel.com. For other peripheral devices such as A/D and D/A converters visit www.maxim-ic.com.

All the schematics and printed circuit boards presented in this book were drawn with the CAD software Eagle™ layout editor, from CadSoft. A freeware version of this software is available for download at www.cadsoft.de. The CD accompanying this book contains the schematics and the PCB design for all the projects described in the book, therefore it is important to download and install the Eagle layout editor in order to be able to edit the existing projects, and use them as starting points for new projects.

Index

Springer Series in

ADVANCED MICROELECTRONICS

1 **Cellular Neural Networks**
Chaos, Complexity
and VLSI Processing
By G. Manganaro, P. Arena,
and L. Fortuna

2 **Technology of Integrated Circuits**
By D. Widmann, H. Mader,
and H. Friedrich

3 **Ferroelectric Memories**
By J.F. Scott

4 **Microwave Resonators and Filters for Wireless Communication**
Theory, Design and Application
By M. Makimoto and S. Yamashita

5 **VLSI Memory Chip Design**
By K. Itoh

6 **Smart Power ICs**
Technologies and Applications
Ed. by B. Murari, R. Bertotti,
and G.A. Vignola

7 **Noise in Semiconductor Devices**
Modeling and Simulation
By F. Bonani and G. Ghione

8 **Logic Synthesis for Asynchronous Controllers and Interfaces**
By J. Cortadella, M. Kishinevsky,
A. Kondratyev, L. Lavagno,
and A. Yakovlev

9 **Low Dielectric Constant Materials for IC Applications**
Editors: P.S. Ho, J. Leu, W.W. Lee

10 **Lock-in Thermography**
Basics and Use
for Functional Diagnostics
of Electronic Components
By O. Breitenstein
and M. Langenkamp

11 **High-Frequency Bipolar Transistors**
Physics, Modelling, Applications
By M. Reisch

12 **Current Sense Amplifiers**
for Embedded SRAM
in High-Performance
System-on-a-Chip Designs
By B. Wicht

13 **Silicon Optoelectronic Integrated Circuits**
By H. Zimmermann

14 **Integrated CMOS Circuits for Optical Communications**
By M. Ingels and M. Steyaert

15 **Gettering Defects in Semiconductors**
By V.A. Perevostchikov
and V.D. Skoupov

16 **High Dielectric Constant Materials**
VLSI MOSFET Applications
Editors: H.R. Huff and D.C. Gilmer

17 **System-level Test and Validation of Hardware/Software Systems**
By M. Sonza Reorda, Z. Peng,
and M. Violante