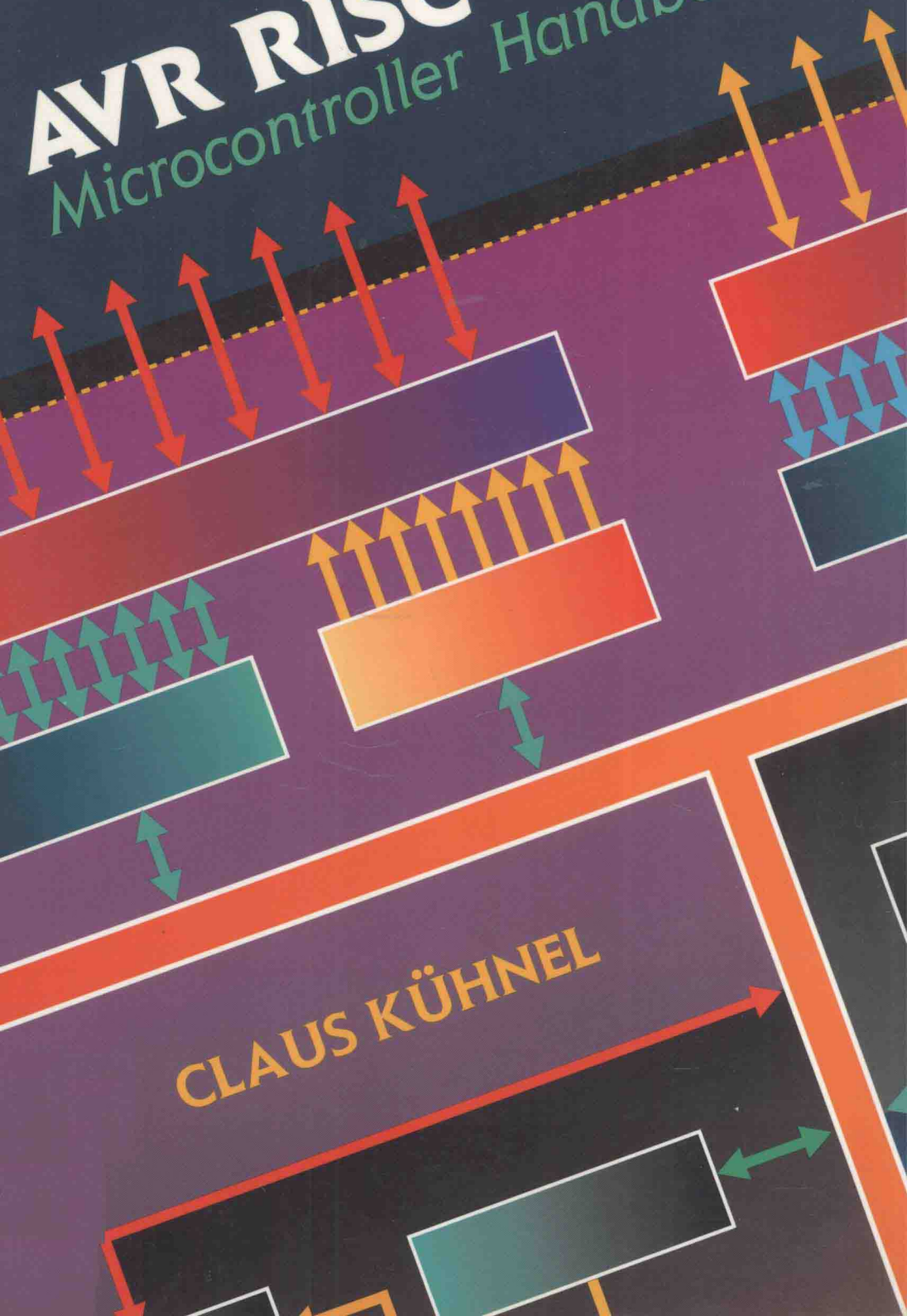


# AVR RISC

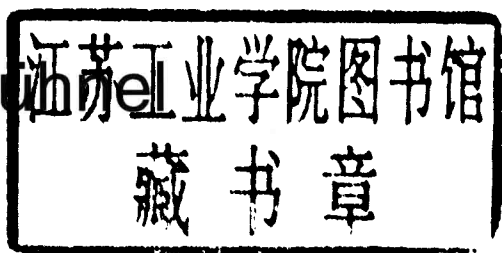
Microcontroller Handbook



CLAUS KÜHNEL

# AVR RISC Microcontroller Handbook

by Claus Koenig




Newnes

Boston Oxford Johannesburg Melbourne New Delhi Singapore

Newnes is an imprint of Butterworth-Heinemann.

Copyright © 1998 by Butterworth-Heinemann

 A member of the Reed Elsevier group

All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the publisher.

AVR is a registered trademark of Atmel Corporation.



Recognizing the importance of preserving what has been written, Butterworth-Heinemann prints its books on acid-free paper whenever possible.



Butterworth-Heinemann supports the efforts of American Forests and the Global ReLeaf program in its campaign for the betterment of trees, forests, and our environment.

#### **Library of Congress Cataloging-in-Publication Data**

Kühnel, Claus, 1951-

AVR RISC microcontroller handbook / by Claus Kühnel.

p. cm.

Includes index.

ISBN 0-7506-9963-9 (alk. paper)

1. Programmable controllers. 2. RISC microprocessors. I. Title.

TJ223.P76K83 1998

629.8'9—dc21

98-14859

CIP

#### **British Library Cataloguing-in-Publication Data**

A catalogue record for this book is available from the British Library.

The publisher offers special discounts on bulk orders of this book.

For information, please contact:

Manager of Special Sales

Butterworth-Heinemann

225 Wildwood Avenue

Woburn, MA 01801-2041

Tel: 781-904-2500

Fax: 781-904-2620

For information on all Butterworth-Heinemann publications available, contact our World Wide Web home page at: <http://www.bh.com>

10 9 8 7 6 5 4 3 2 1

Printed in the United States of America

# **AVR RISC Microcontroller Handbook**

---

# ***Preface***

3

The AVR enhanced microcontrollers are based on a new RISC architecture that has been developed to take advantage of semiconductor integration and software capabilities in the 1990s. The resulting microcontrollers offer the highest MIPS/mW capability available in the 8-bit microcontroller market.

High-level languages are rapidly becoming the standard programming methodology for embedded microcontrollers because of improved time-to-market and simplified maintenance support. The AVR architecture was developed together with C language experts to ensure that the hardware and software work hand-in-hand to develop highly efficient, high-performance code.

To optimize code size, performance, and power consumption, the AVR architecture has incorporated a large fast-access register file and fast single-cycle instructions.

The AVR architecture supports a complete spectrum of price performance from simple small-pin-count controllers to high-range devices with large on-chip memories. The Harvard-style architecture directly addresses up to 8 Mbytes of data memory. The register file is dual mapped and can be addressed as a part of the on-chip SRAM memory to enable fast context switching.

The AVR enhanced RISC microcontroller family is manufactured with ATMEL's low-power nonvolatile CMOS technology. The on-chip in-system-programmable (ISP) downloadable flash memory allows the program memory to be reprogrammed in-system through an SPI serial port or conventional memory programmer. By combining an enhanced RISC architecture with downloadable flash memory on the same chip, the AVR microcontroller family offers a powerful solution to embedded control application.

This book describes the new AVR architecture and the program development for those microcontrollers of the family available in early 1997. Some tools from ATMEL and third-party companies help to give a first impression

of the AVR microcontrollers. Thus, the evaluation of hardware and programming in Assembler and C of that type of microcontroller is supported very well. A simulator makes program verification possible without any hardware.

The development of the AVR microcontroller family by ATMEL shows clearly that remarkable results are not limited to high-end microcontrollers that are often the focus of consideration.

I thank ATMEL for the development of this interesting microcontroller family, because studying these new devices and their development environment was very interesting and made writing this book enjoyable.

Finally, I wish to thank ATMEL Norway and IAR Sweden for their support of this project and my wife, Jutta, for her continued understanding during the preparation of this book.

---

# Contents

**Preface**

**ix**

---

<b>1</b>	<b>Some Basics</b>	<b>1</b>
1.1	Architecture	1
1.2	Important Terms	5
1.3	Numbers	7

---

<b>2</b>	<b>Hardware Resources of AVR Microcontrollers</b>	<b>9</b>
2.1	Architectural Overview	9
2.2	The Arithmetic Logic Unit	13
2.3	Program and Data Memories	14
2.3.1	Downloadable Flash Program Memory	14
2.3.2	SRAM Data Memory	14
2.3.3	General-Purpose Register File	16
2.3.4	I/O Register	16
2.3.5	EEPROM Data Memory	17
2.4	Peripherals	21
2.4.1	Timer/Counter	21
2.4.2	Watchdog Timer	31
2.4.3	Serial Peripheral Interface SPI	32
2.4.4	Universal Asynchronous Receiver and Transmitter	37
2.4.5	Analog Comparator	43
2.4.6	I/O Ports	48
2.5	Reset and Interrupt System	57
2.5.1	Interrupt Vector Table	57
2.5.2	Reset Sources	58
2.6	Clock	60

---

<b>3</b>	<b>Handling the Hardware Resources</b>	<b>63</b>
3.1	Memory Addressing Modes	63
3.1.1	Register Direct Addressing	63
3.1.2	I/O Direct Addressing	63
3.1.3	SRAM Direct Addressing	65
3.1.4	SRAM Indirect Addressing	65
3.1.5	Constant Addressing Using the LPM Instruction	69
3.1.6	Jumps and Calls	69
3.2	Instruction Set	71
3.3	Reset and Interrupt Handling	112
3.4	Watchdog Handling	115
3.5	Stack	116
3.6	Program Constructs	120
3.6.1	Conditional Branches	120
3.6.2	Program Loops	123
3.7	Defensive Programming	127
3.7.1	Refreshing Port Pins and Important Registers	127
3.7.2	Polling Inputs	128

---

<b>4</b>	<b>Development Tools</b>	<b>131</b>
4.1	ATMEL AVR Assembler and Simulator	131
4.1.1	ATMEL AVR Assembler	133
4.1.2	ATMEL AVR Simulator	139
4.2	ATMEL AVR Studio	144
4.3	IAR Embedded Workbench EWA90	146
4.3.1	Summary of Available AVR Tools	148
4.3.2	IAR C Compiler	148
4.3.3	Macro-Assembler for Time-Critical Routines	150
4.3.4	Linker	151
4.3.5	ANSI C Libraries	151
4.3.6	IAR CWA90 Debugger/Simulator	151
4.3.7	EWA90 Demo of AVR Embedded Workbench	153
4.4	AVR Pascal from E-LAB Computers	155
4.5	AVR BASIC from Silicon Studio	166
4.6	Programmer and Evaluation Boards	168
4.6.1	AVR Development Board from Atmel	169
4.6.2	ISP Starter Kit from Equinox	171
4.6.3	SIMMSTICK from Silicon Studio	173
4.6.4	Parallel Port Programmer BA1FB	175
4.6.5	Serial Port Programmer PonyProg	176



---

<b>5</b>	<b>Example Programs</b>	<b>179</b>
5.1	Example Programs in AVR Assembler	179
5.1.1	Assembler Programs for the AT90S1200	180
5.1.2	Assembler Programs for the AT90S8515	228
5.2	Example Program in C	261
5.3	Example Programs in AVR BASIC	265
5.3.1	Microcontroller Test	265
5.3.2	Pulse-Width Modulation and Serial Communication for the AT90S8515	266
<b>Appendix A</b>	<b>Part Numbering System</b>	<b>273</b>
<b>Appendix B</b>	<b>Pin Configurations</b>	<b>275</b>
<b>Appendix C</b>	<b>Schematics of SIMMSTICK Modules</b>	<b>277</b>
<b>Appendix D</b>	<b>Register and Bit Definitions</b>	<b>281</b>
<b>Appendix E</b>	<b>Some Fundamentals to RS-232</b>	<b>287</b>
<b>Appendix F</b>	<b>Some Fundamentals to RS-422 and RS-485</b>	<b>293</b>
<b>Appendix G</b>	<b>8-Bit Intel Hex File Format</b>	<b>297</b>
<b>Appendix H</b>	<b>Decimal-to-Hex-to-ASCII Converter</b>	<b>299</b>
<b>Appendix I</b>	<b>Overview of Atmel's Application Notes and Software</b>	<b>301</b>
<b>Appendix J</b>	<b>Literature</b>	<b>305</b>
<b>Appendix K</b>	<b>Contacts</b>	<b>307</b>
	<b>Index</b>	<b>309</b>

In the following chapters we will use some special terms that are perhaps not so familiar to a beginner. Some explanations should make the world of microcontroller terms and functionality more transparent.

---

## 1.1 Architecture

All microcontrollers have more or less the same function groups. Internally we find memory for instructions and data and a central processing unit (CPU) for handling the program flow, manipulating the data, and controlling the peripherals.

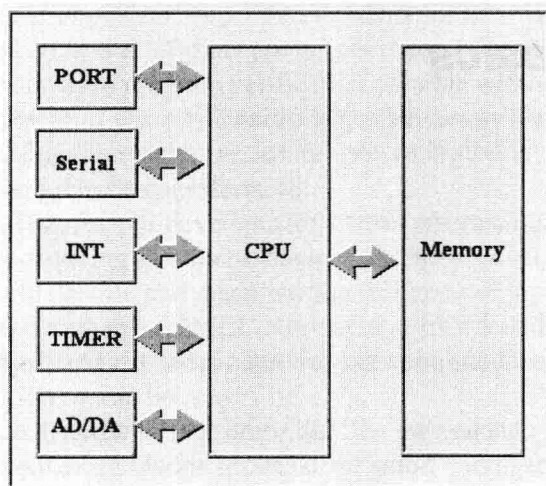
Figure 1-1 shows the basic function blocks in a microcontroller. The CPU communicates with each of these function blocks (memory and peripherals).

To build a powerful microcontroller, it is important to reduce the tasks carried out from the CPU itself and to optimize the handling of the remaining tasks.

On the left side of Figure 1-1, peripherals are arranged. These peripherals react with the world outside, or in more technical terms, with the process. In modern microcontrollers, the peripherals relieve the CPU by handling the external events separately.

In an asynchronous serial communication, for example, the CPU transmits the character to be sent only to the serial port. The required serialization and synchronization are performed by the serial port itself. On the other side, receiving a character is important for the CPU only when all bits are stored in a buffer and are ready for an access of the CPU.

A port is built by a certain number of connections between the microcontroller and the process, often a factor of 8. It usually supports a bitwise digital input and/or output (I/O).



**Figure 1-1**  
Microcontroller function blocks.

Serial ports communicate with other external components by means of serial communication protocols. Asynchronous and synchronous serial communications must be differentiated. Both kinds of serial communication have their own building blocks—Universal Asynchronous Receiver and Transmitter (UART) for asynchronous communication, and Serial Peripheral Interface (SPI) for synchronous communication. In Figure 1-1 this differentiation is not shown.

Because the microcontroller is designed for process-related applications with real-time character, some other function groups are implemented in a microcontroller.

Modern microcontrollers have a fairly comfortable interrupt system. An interrupt breaks the running program to process a special routine, called the interrupt service routine. Some external events require an immediate reaction. These events generate an interrupt; the whole system will be frozen, and in an interrupt service routine the concerning event is handled before the program goes on. If the microcontroller has to process many interrupts, an interrupt controller can be helpful.

To fulfill timing conditions, microcontrollers have one or more timers implemented. The function blocks usually work as timer and/or counter subsystems. In the simplest case, we have an 8-bit timer register. Its contents will be incremented (or decremented) with each clock cycle (CLK). Any time the value 255 (or 0) is reached, the register will overflow (or underflow) with the

next clock. This overflow (or underflow) is signaled to the CPU. The actual delay depends on the preload value.

If the preload value equals zero, then the overflow will occur after 256 clock periods. If the preload value equals 250, in an up-counter the overflow will occur after six clock cycles. A block diagram of a simple timer is shown in Figure 1-2.

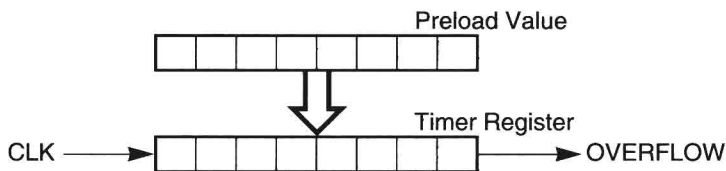
Last but not least, we had in Figure 1-1 an AD/DA subsystem. For adaptations to a real-world process, analog-to-digital and/or digital-to-analog converters are often required. This is not the place to discuss all features of these more or less complex subsystems. Some microcontrollers include such an AD/DA subsystem or can control an external one. In other cases, only analog comparators are integrated.

Thus, each peripheral has its own intelligence for handling external events and to realize preprocessing.

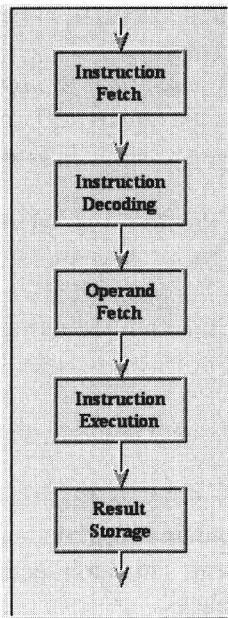
On the right side of Figure 1-1, we find the microprocessor part (CPU and memory). The memory contains program and data. CPU and memory are connected through a bus system. This architecture—called “von Neumann” architecture—has some drawbacks.

In Figure 1-3 instruction handling is explained. Before an instruction can be operated, it must first be fetched from memory. Next, the program counter must be incremented. After this incrementation, the program counter points to the next instruction in memory. Before execution of the fetched instruction, it has to be decoded. As a result of this decoding, further memory accesses for operands or addresses are possible. The instruction execution includes arithmetic or logical operations, followed by storage of the result back to the memory.

It seems not so difficult to understand that the handling of one instruction requires more than one memory access. Usually, one instruction manipulates one data byte. Therefore, several memory accesses are inconsistent with the manipulation of one data byte by one instruction.



**Figure 1-2**  
Timer.



**Figure 1-3**  
Instruction handling.

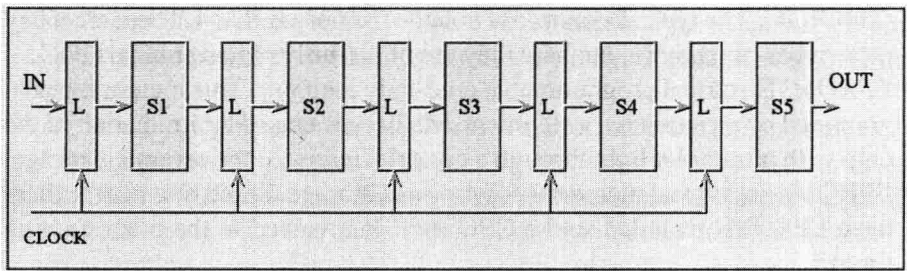
Very large scale integration (VLSI) technology has made it possible to build very fast CPUs. However, slow memory access times inhibits this evolution. This fundamental problem of the von Neumann architecture is called the “von Neumann bottleneck.”

To avoid these limitations, system designers implement some sort of cache memory—a fast memory buffer between the main memory and the CPU. Another, and most recent approach, is to separate the paths to the memory system for instructions and data—the “Harvard” architecture.

The RISC design philosophy also tries to eliminate the “von Neumann bottleneck” by strict limitation of memory operations by means of many internal registers.

In order to increase performance, pipelining is widely used in modern microprocessor architectures. A basic linear pipeline consists of a cascade of processing stages.

The instruction unit in a microprocessor architecture can consist of pipeline stages for instruction fetch, instruction decode, operand fetch, execute instruction, and store results. Using such a technique for preparing an execute operation allows a new instruction to be executed every clock cycle. In this



**Figure 1-4**  
Instruction pipeline.

design, five different instructions are flowing simultaneously in the pipeline. Figure 1-4 shows an instruction pipeline for all steps of the instruction handling according to Figure 1-3.

All blocks in the pipeline are pure combinational circuits performing arithmetic or logic operations over the data stream flowing through the pipeline. The stages are separated by interface latches.

If a conditional program flow instruction changes the sequence in which the program is executed, the prefetched instructions, addresses, and operands in the instruction pipe are discarded. Different procedures are used to fill these empty slots with valid information. Since the branch-type instructions have damaging effects on the pipeline architecture performance, this is one of the most complex design stages in modern computer architecture.

---

## 1.2 Important Terms

The following important terms will help in understanding subsequent chapters.

- *Internal architecture*: The concept of building the internal electronics of a microcontroller.
- *Memory*: A function block for program and data storage. Here it is important to distinguish between nonvolatile and volatile memories. Nonvolatile memories are required for storage of programs so the system does not have to be reprogrammed after power has been off. Working variables and intermediate results need to be stored in a memory that can be written (or programmed) quickly and easily during system operation. It is not important to know these data after power off. Examples of nonvolatile memories are EPROM and

OTP-ROM. The typical example of volatile memory is RAM. There are many more types of memory, but here they are of no further interest in this book.

- *EPROM*: Electrical programmable read-only memory. This memory is programmed by programmer equipment. Memory is erased by irradiation of the chip with ultraviolet light through a crystal window in the ceramic package. EPROMs are typical memories for program storage. Some microcontrollers have EPROMs included so the CPU itself is involved in the programming process.
- *OTP-ROM*: One-time programmable EPROM. This type of EPROM is one-time programmable because the package is without a crystal window and therefore not UV erasable. Using cheap plastic packages without a window instead of windowed ceramic packages decreases the cost significantly. Therefore, the PROM included in a microcontroller is often an OTP-ROM.
- *EEPROM*: Electrical erasable and programmable read-only memory. For reading, it is no different from a normal EPROM. Writing or, better, programming an EEPROM differs from that for a normal EPROM completely. The program cycle is about 10 ms for a byte or a block of bytes. Because of the program time, the EEPROM is suitable for storage of seldom-changing data, such as initialization or configuration data. For modern EEPROMs, 10 million program cycles are possible.
- *Flash memory*: Nonvolatile read–write memory for program and data storage. Flash memories combine EPROM programming with *EEPROM*-like in-system electrical erasure. In contrast to EEPROM, a bitwise erasure is impossible.
- *RAM*: Random access memory, which can be programmed and read at any time. RAMs are typical memories for data storage and are volatile.
- *Oscillator*: A circuit that produces a constant-frequency square wave used by the computer as a timing or sequencing reference. A microcontroller typically includes all elements of this circuit except the frequency-determining component(s) (crystals, ceramic resonators, or RC components). In some cases all frequency-determining components are also on-chip.
- *Reset circuit*: Generates a reset impulse to reset the computer, in some cases. The most important reset is the power-on reset. Switching power-on starts the program of the microcontroller.
- *I/O ports*: The connections to the process. Such ports are mainly bit-programmable in both directions.
- *Watchdog*: A counter circuit that must be reset by the running program. If the program hangs, no watchdog reset can occur, and the watchdog counter overflows. As a result of this overflow, the watchdog initiates a reset and avoids wild running of the microcontroller

- *Real-time clock/counter*: A further counter circuit able to count real-time pulses from the process side or controlled by a clock generated by the internal clock.
- *Terminal*: Equipment for serial I/O to the microcontroller. In most cases a PC running a terminal program is used.
- *Program installation*: Installation of an user program on the hard disk of a personal computer. The installation process includes copying the file(s), unencrypting these when needed, generating of a program group in a Windows environment, and so forth.
- *Program initialization*: To provide installed software with the required constants and/or parameters. For example, initialization would provide baud rate and handshake parameters for a serial communication.
- *Instruction set*: The whole list of instructions that will be understood by the microcontroller.
- *MSB*: Most significant bit. In the 8-bit data word  $D7:D0 = 10101010$ , the MSB is  $D7 = 1$ .
- *LSB*: Least significant bit. In the 8-bit data word  $D7:D0 = 10101010$ , the LSB is  $D0 = 0$ .
- *Pull-up resistor*: A resistor that gives a Hi signal in a high-impedance circuit.
- *Pull-down resistor*: A resistor that gives a Lo signal in a high-impedance circuit.
- *Kbyte, Mbyte*: Units of bits and bytes. “K” here does not mean a value of 1000, and “M” does not mean 1,000,000. In the binary system used in information technology, “K” stands for  $2^{10} = 1024$ , and “M” for  $2^{10} * 2^{10} = 1,048,576$ .
- *PDIP*: Plastic dual inline package for integrated circuits.
- *SOIC*: Small-outline integrated circuit.
- *PLCC*: Plastic J-leaded chip carrier.

---

### 1.3 Numbers

Numbers can be displayed in various formats. Usually we think of decimal numbers. In digital systems, and also in the microcontroller world, we have to think binary because only the two states (Lo and Hi) are allowed.

For a byte-wide number (8 bits) we get the relations among binary, decimal, and hexadecimal number notation shown in Table 1-1.

To indicate the number system used, an index is usually used in text notation. The hexadecimal number  $11_{\text{H}}$  is thus equivalent to the decimal  $17_{\text{D}}$ . Normally, the index for decimal numbers is not shown.



**Table 1-1**

Notation of numbers.

<i>Binary Number</i>	<i>Decimal Equivalent</i>	<i>Hexadecimal Equivalent</i>	<i>Notation in Text</i>	<i>Notation in Assembler or BASIC</i>	<i>Notation in C</i>
0000 0000	0	0	0 <sub>H</sub>	\$0000	0x0000
0000 0001	1	1	1 <sub>H</sub>	\$0001	0x0001
0000 0010	2	2	2 <sub>H</sub>	\$0002	0x0002
0000 0011	3	3	3 <sub>H</sub>	\$0003	0x0003
...	...	...	...	...	...
0000 1110	14	E	E <sub>H</sub>	\$000E	0x000E
0000 1111	15	F	F <sub>H</sub>	\$000F	0x000F
0001 0000	16	10	10 <sub>H</sub>	\$0010	0x0010
0001 0001	17	11	11 <sub>H</sub>	\$0011	0x0011
.....	...	...	...	...	...
1111 1110	126	FE	FE <sub>H</sub>	\$00FE	0x00FE
1111 1111	127	FF	FF <sub>H</sub>	\$00FF	0x00FF

In Assembler and BASIC, hexadecimal numbers are normally presented in the \$-format. In C, the notation of a hexadecimal number is given in a special format. The rightmost column in Table 1-1 shows this format.