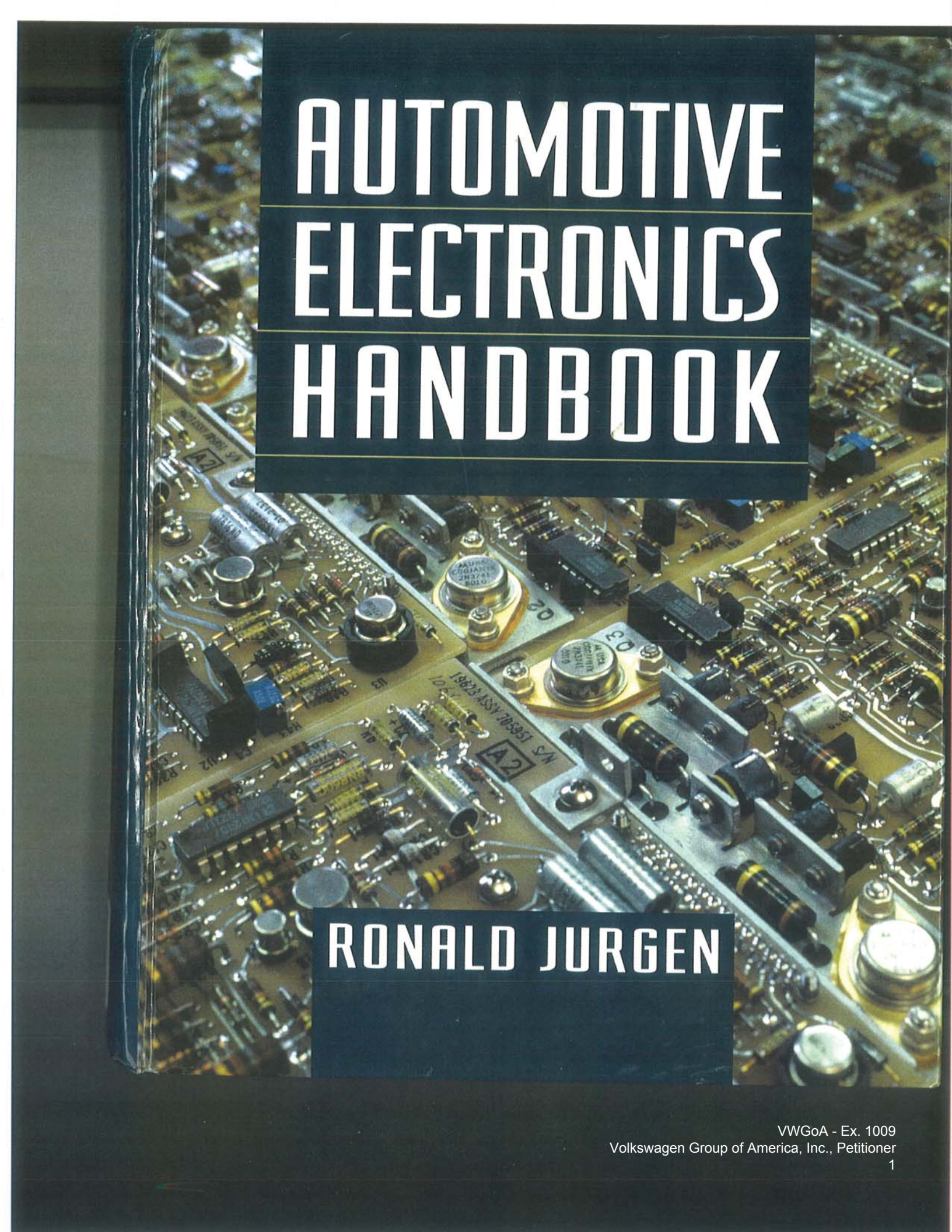


AUTOMOTIVE ELECTRONICS HANDBOOK

Ronald K. Jurgen Editor in Chief

McGraw-Hill, Inc.
New York San Francisco Washington, D.C. Auckland Bogotá
Caracas Lisbon London Madrid Mexico City Milan
Montreal New Delhi San Juan Singapore
Sydney Tokyo Toronto

Related McGraw-Hill Books of Interest

Handbooks

Avallone and Baumeister • MARK'S STANDARD HANDBOOK FOR MECHANICAL ENGINEERS
Benson • AUDIO ENGINEERING HANDBOOK
Brady • MATERIALS HANDBOOK
Chen • COMPUTER ENGINEERING HANDBOOK
Considine • PROCESS/INDUSTRIAL INSTRUMENTS AND CONTROL HANDBOOK
Coombs • PRINTED CIRCUITS HANDBOOK
Coombs • ELECTRONIC INSTRUMENT HANDBOOK
Di Giacomo • DIGITAL BUS HANDBOOK
Fink and Beaty • STANDARD HANDBOOK FOR ELECTRICAL ENGINEERS
Fink and Christiansen • ELECTRONICS ENGINEERS' HANDBOOK
Ganic • MCGRAW-HILL HANDBOOK OF ESSENTIAL ENGINEERING INFORMATION
Harper • ELECTRONIC PACKAGING AND INTERCONNECTION HANDBOOK
Harper and Sampson • ELECTRONIC MATERIALS AND PROCESSES HANDBOOK
Hicks • STANDARD HANDBOOK OF ENGINEERING CALCULATIONS
Hodson • MAYNARD'S INDUSTRIAL ENGINEERING HANDBOOK
Johnson • ANTENNA ENGINEERING HANDBOOK
Juran and Gryna • JURAN'S QUALITY CONTROL HANDBOOK
Kaufman and Seidman • HANDBOOK OF ELECTRONICS CALCULATIONS
Lenk • MCGRAW-HILL ELECTRONIC TESTING HANDBOOK
Lenk • LENK'S DIGITAL HANDBOOK
Mason • SWITCH ENGINEERING HANDBOOK
Schwartz • COMPOSITE MATERIALS HANDBOOK
Townsend • DUDLEY'S GEAR HANDBOOK
Tuma • ENGINEERING MATHEMATICS HANDBOOK
Waynant • ELECTRO-OPTICS HANDBOOK
Woodson • HUMAN FACTORS DESIGN HANDBOOK

Other

Boswell • SUBCONTRACTING ELECTRONICS
Gieck • ENGINEERING FORMULAS
Ginsberg • PRINTED CIRCUIT BOARD DESIGN
Johnson • ISO 9000
Lenk • MCGRAW-HILL CIRCUIT ENCYCLOPEDIA AND TROUBLESHOOTING GUIDE, VOLS. 1 AND 2
Lubben • JUST-IN-TIME MANUFACTURING
Markus and Sclater • MCGRAW-HILL ELECTRONICS DICTIONARY
Saylor • TQM FIELD MANUAL
Soin • TOTAL QUALITY CONTROL ESSENTIALS
Whitaker • ELECTRONIC DISPLAYS
Young • ROARK'S FORMULAS FOR STRESS AND STRAIN

To order or to receive additional information on these or any other McGraw-Hill titles, please call 1-800-822-8158 in the United States. In other countries, please contact your local McGraw-Hill office.

BC14BCZ

Library of Congress Cataloging-in-Publication Data

Automotive electronics handbook / Ronald Jurgen, editor in chief.
p. cm.
Includes index.
ISBN 0-07-033189-8
1. Automobiles—Electronic equipment. I. Jurgen, Ronald K.
TL272.5.A982 1994
629.25'49—dc 94-39724
CIP

2 3 4 5 6 7 8 9 0 AGM/AGM 9 0 9 8 7 6 5

ISBN 0-07-033189-8

The sponsoring editor for this book was Stephen S. Chapman, the editing supervisor was Virginia Carroll, and the production supervisor was Suzanne W. B. Rapcavage. It was set in Times Roman by North Market Street Graphics.

Printed and bound by Arcata Graphics/Martinsburg.

This book is printed on acid-free paper.

This book is dedicated to Robert H. Lewis and to the memories of Douglas R. Jurgen and Marion Schappel.

CONTENTS

Part 4 Displays and Information Systems

Part 5 Safety, Convenience, Entertainment, and Other Systems

CONTRIBUTORS

Robert E. Bicking *Honeywell, Micro Switch Division* (CHAP. 4)
Tracy Blake *Arizona State University* (CHAP. 30)
David S. Boehmer *Intel Corporation* (CHAP. 11)
Werner Brehm *Robert Bosch GmbH* (CHAP. 13)
Wolfgang Bremer *Robert Bosch GmbH* (CHAP. 22)
Wolfgang Bullmer *Robert Bosch GmbH* (CHAP. 13)
Jerry L. Cage *Allied Signal, Inc.* (CHAP. 15)
Tom Chrapkiewicz *Philips Semiconductor* (CHAP. 25)
Armin Czinczel *Robert Bosch GmbH* (CHAP. 16)
Jeffrey N. Denenberg *Noise Cancellation Technologies, Inc.* (CHAP. 31)
William C. Dunn *Motorola Semiconductor Products* (CHAP. 7)
Randy Frank *Motorola Semiconductor Products* (CHAPS. 2, 5, 32)
Robert L. French *R. L. French & Associates* (CHAP. 29)
Frieder Heintz *Robert Bosch GmbH* (CHAP. 22)
Gary C. Hirschlieb *Robert Bosch GmbH* (CHAP. 12)
Raymond S. Hobbs *Arizona Public Service Company* (CHAP. 30)
Gerhard Hötzel *Robert Bosch GmbH* (CHAP. 6)
Robert Hugel *Robert Bosch GmbH* (CHAP. 22)
Ronald K. Jurgen *Editor* (CHAPS. 1, 20, 21)
George G. Karady *Arizona State Univeristy* (CHAP. 30)
Donald B. Karner *Electric Transportation Application* (CHAP. 30)
Shinichi Kato *Nissan Motor Co., Ltd.* (CHAP. 24)
Bernhard K. Mattes *Robert Bosch GmbH* (CHAP. 23)
Fred Miesterfeld *Chrysler Corporation* (CHAP. 26)
Salim Momin *Motorola Semiconductor Products* (CHAP. 32)
James P. Muccioli *JASTECH* (CHAPS. 27, 28)
Klaus Müller *Robert Bosch GmbH* (CHAP. 10)
Kurt Neuffer *Robert Bosch GmbH* (CHAP. 13)
Harald Neumann *Robert Bosch GmbH* (CHAP. 6)
Paul Nickson *Analog Devices, Inc.* (CHAP. 3)
Johann Riegel *Robert Bosch GmbH* (CHAP. 6)

Makoto Sato *Honda R&D Co., Ltd.* (CHAP. 18)
Gottfried Schiller *Robert Bosch GmbH* (CHAP. 12)
Shari Stottler *Robert Bosch GmbH* (CHAP. 12)
Richard Valentine *Motorola Inc.* (CHAPS. 14, 19)
Helmut Weyl *Robert Bosch GmbH* (CHAP. 6)
Hans-Martin Wiedenmann *Robert Bosch GmbH* (CHAP. 6)
William G. Wolber *Cummins Electronics Co., Inc.* (CHAPS. 8, 9)
Akatsu Yohsuke *Nissan Motor Co., Ltd.* (CHAP. 17)

PREFACE

Automotive electronics as we know it today encompasses a wide variety of devices and systems. Key to them all, and those yet to come, is the ability to sense and measure accurately automotive parameters. Equally important at the output is the ability to initiate control actions accurately in response to commands. In other words, sensors and actuators are the heart of any automotive electronics application. That is why they have been placed first in this handbook where they are described in technical depth. In other chapters, application-specific discussions of sensors and actuators can be found.

The importance of sensors and actuators cannot be overemphasized. The future growth of automotive electronics is arguably more dependent on sufficiently accurate and low-cost sensors and actuators than on computers, controls, displays, and other technologies. Yet it is those nonsensor, nonactuator technologies that are to many engineers the more "glamorous" and exciting areas of automotive electronics.

In the section on control systems, a key in-depth chapter deals with automotive microcontrollers. Without them, all of the controls described in the chapters that follow in that section—engine, transmission, cruise, braking, traction, suspension, steering, lighting, windshield wipers, air conditioner/heater—would not be possible. Those controls, of course, are key to car operation and they have made cars over the years more drivable, safe, and reliable.

Displays, trip computers, and on- and off-board diagnostics are described in another section, as are systems for passenger safety and convenience, antitheft, entertainment, and multiplex wiring. Displays and trip computers enable the driver to readily obtain valuable information about the car's operation and anticipated trip time. On- and off-board diagnostics have of necessity become highly sophisticated to keep up with highly sophisticated electronic controls. Passenger safety and convenience items and antitheft devices add much to the feeling of security and pleasure in owning an automobile. Entertainment products are what got automotive electronics started and they continue to be in high demand by car buyers. And multiplex wiring, off to a modest start in production cars, holds great promise for the future in reducing the cumbersome wiring harnesses presently used.

The section on electromagnetic interference and compatibility emphasizes that interference from a variety of sources, if not carefully taken into account early on, can raise havoc with what otherwise would be elegant automotive electronic designs. And automotive systems themselves, if not properly designed, can cause interference both inside and outside the automobile.

In the final section on emerging technologies, some key newer areas are presented:

- Navigation aids and intelligent vehicle-highway systems are of high interest worldwide since they hold promise to alleviate many of vehicle-caused problems and frustrations in our society.
- While it may be argued that electric vehicles are not an emerging technology, since they have been around for many years, it certainly is true that they have yet to come into their own in any really meaningful way.
- Electronic noise cancellation is getting increasing attention from automobile designers seeking an edge over their competitors.

The final chapter on future vehicle electronics is an umbrella discussion that runs the gamut of trends in future automotive electronics hardware and software. It identifies potential technology developments and trends for future systems.

Nearly every chapter contains its own glossary of terms. This approach, rather than one overall unified glossary, has the advantage of allowing terms to be defined in a more application-specific manner—in the context of the subject of each chapter. It should also be noted that there has been no attempt in this handbook to cover, except peripherally, purely mechanical and electrical devices and systems. To do so would have restricted the number of pages available for automotive electronics discussions.

Finally, the editor would like to thank all contributors to the handbook and particularly two individuals: Otto Holzinger of Robert Bosch GmbH in Stuttgart, Germany and Randy Frank of Motorola Semiconductor Products in Phoenix, Arizona. Holzinger organized the many contributions to this handbook from his company. Frank, in addition to contributing two chapters himself and cocontributing a third, organized the other contributions from Motorola. Without their help, this handbook would not have been possible.

Ronald K. Jurgen

CHAPTER 11
AUTOMOTIVE MICROCONTROLLERS

David S. Boehmer
Senior Applications Engineer
Intel Corporation

A microcontroller can be found at the heart of almost any automotive electronic control module or ECU in production today. Automotive systems such as antilock braking control (ABS), engine control, navigation, and vehicle dynamics all incorporate at least one microcontroller within their ECU to perform necessary control functions. Understanding the various features and offerings of microcontrollers that are available on the market today is important when making a selection for an application. This chapter is intended to provide a look at various microcontroller features and provide some insight into their characteristics from an automotive application point of view.

11.1 MICROCONTROLLER ARCHITECTURE AND PERFORMANCE CHARACTERISTICS

A microcontroller can essentially be thought of as a single-chip computer system and is often referred to as a single-chip microcomputer. It detects and processes input signals, and responds by asserting output signals to the rest of the ECU. Fabricated upon this highly integrated, single piece of silicon are all of the features necessary to perform embedded control functions. Microcontrollers are fabricated by many manufacturers and are offered in just about any imaginable mix of memory, I/O, and peripheral sets. The user customizes the operation of the microcontroller by programming it with his or her own unique program. The program configures the microcontroller to detect external events, manipulate the collected data, and respond with appropriate output. The user's program is commonly referred to as code and typically resides on-chip in either ROM or EPROM. In some cases where an excessive amount of code space is required, memory may exist off-chip on a separate piece of silicon. After power-up, a microcontroller executes the user's code and performs the desired embedded control function.

Microcontrollers differ from microprocessors in several ways. Microcontrollers can be thought of as a complete microcomputer on a chip that integrates a CPU with memory and various peripherals such as analog-to-digital converters (A/D), serial communication units (SIO, SSIO), high-speed input and output units (HSIO, EPA, PWM), timer/counter units, and

standard low-speed input/output ports (LSIO). Microcontrollers are designed to be embedded within event-driven control applications and generally have all necessary peripherals integrated onto the same piece of silicon. Microcontrollers are utilized in applications ranging from automotive ABS to household appliances in which the microcontroller's function is predefined and limited user interface is required.

Microprocessors, on the other hand, typically require external peripheral devices to perform their intended function and are not suited to be utilized in single-chip designs. Microprocessors basically consist of a CPU with register arrays and interrupt handlers. Peripherals such as A/D and HSIO are rarely integrated onto microprocessor silicon. Microprocessors are designed to process large quantities of data and have the capability to handle large amounts of external memory. Although microprocessors are typically utilized in applications which are much more human-interface and I/O intensive such as personal computers and office workstations, they are beginning to find their way into embedded applications.

Choosing a microcontroller for an application is a process that takes careful investigation and thought. Items such as memory size, frequency, bus size, I/O requirements, and temperature range are all basic requirements that must be considered when choosing a microcontroller. The microcontroller family must possess the performance capability necessary to successfully accomplish the intended task. The family should also provide a memory, I/O, and frequency growth path that allows easy upgradability to meet market demands. Additionally, the microcontroller must meet the application's thermal requirements in order to guarantee functionality over the intended operating temperature range. Items such as these must all be considered when choosing a microcontroller for an automotive application.

11.1.1 Block Diagram

Usually the first item a designer will see when opening a microcontroller data book or data sheet is a block diagram. A block diagram provides a high-level pictorial representation of a microcontroller and depicts the various peripherals, I/O, and memory functions the microcontroller has to offer. The block diagram gives the designer a quick indication if the particular microcontroller will meet the basic memory, I/O, and peripheral needs of their application. Figure 11.1 shows a block diagram for a state-of-the-art microcontroller. It depicts 32 Kbytes

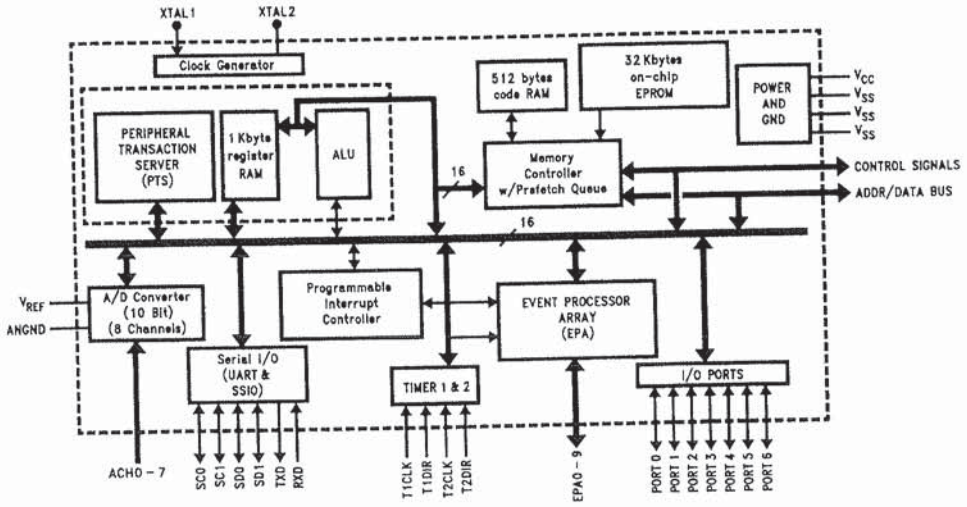


FIGURE 11.1 Microcontroller block diagram.

of EPROM, 1 Kbyte of register RAM, 6 I/O ports, an A-to-D converter, 2 timers, high-speed input/output (I/O) channels, as well as many other peripherals. These features may be "excessive" to a designer looking for a microcontroller to implement in an automotive trip-computer application but would be excellently suited for automotive ABS/traction control or engine control.

11.1.2 Pin-Out Diagram

A microcontroller's pin-out diagram is used to specify the functions assigned to pins relative to their position on a given package. An example pin-out diagram is shown in Fig. 11.2. Note that most pins have multiple functions assigned to them. Pins that can support more than one function are referred to as multifunction pins. The default function for multifunction pins is normally that of low-speed input and output (discussed later in this chapter). If the user should wish to select the secondary or special function associated with the pin, he or she can do so by writing to the appropriate special function register. There are some exceptions. A good example is pins used for interfacing to external memory. If the device is instructed to power-up executing from external memory as opposed to on-chip memory, the address data bus and associated control pins will revert to their special function as opposed to low-speed I/O.

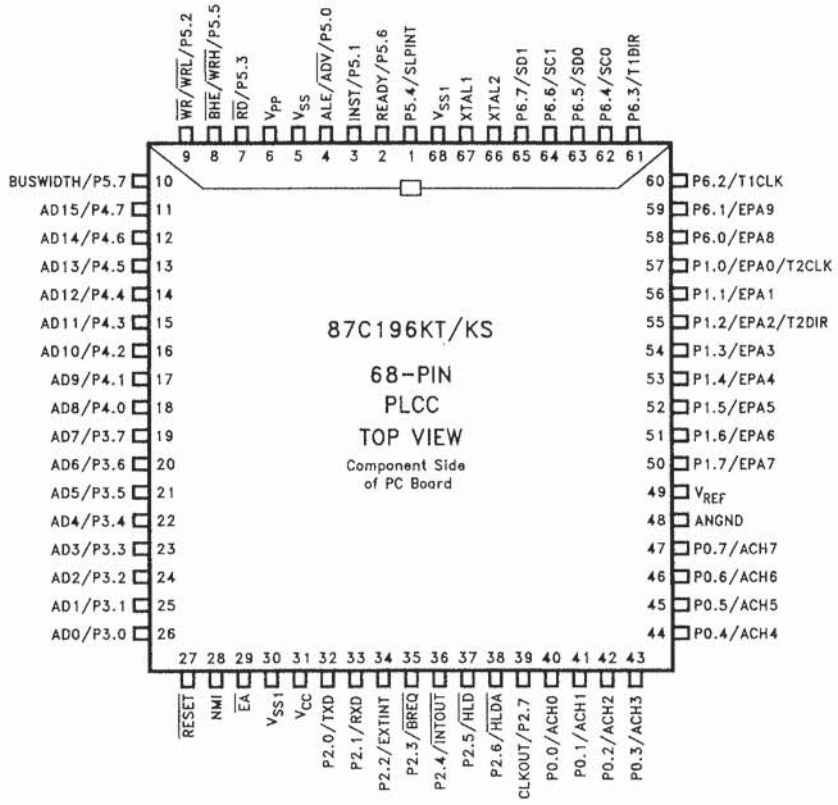


FIGURE 11.2 Microcontroller pin-out diagram.

11.1.3 Central Processing Unit

The central processing unit or CPU can be thought of as the brain of a microcontroller. The CPU is the circuitry within a microcontroller where instructions are executed and decisions are made. Mathematical calculations, data processing, and control signal generation all take place within the CPU. Major components of the CPU include the arithmetic logic unit (ALU), register file, instruction register, and a microcode engine. The CPU is connected to the bus controller and other peripherals via a bidirectional data bus.

Microcontrollers are, for the most part, digital devices. As digital devices, microcontrollers utilize a binary numbering system with a base of 2. Binary data digits or *bits* are expressed as either a logic "1" (boolean value of true) or a logic "0" (boolean value of false). In a 5-V system, a logic "1" may be simply defined as a +5-V state and a logic "0" may be defined as a 0-V state. A bit is a single memory or register location that can contain either a logic "1" or a logic "0" state. Bits of data can be arranged as a *nibble* (4 bits of data), a *byte* (8 bits of data), or as a *word* (16 bits of data). It should also be noted that, in some instances, a word may be defined as the data width that a given microcontroller can recognize at a time, be it 8 bits or 16 bits. For purposes of this chapter, we will refer to a word as being 16 bits. Data can also be expressed as a double word which is an unsigned 32-bit variable with a value between 0 and 4,294,967,295. Most architectures support this data only for shifts, dividends of a 32-by-16 divide, or for the product of a 16-by-16 multiply.

The most common way of referring to a microcontroller is by the width of its CPU. This indicates the width of data that the CPU can process at a time. A microcontroller with a CPU that can process 8 bits of data at a time is referred to as an 8-bit microcontroller. A microcontroller with a CPU that can process 16 bits of data at a time is referred to as a 16-bit microcontroller. With this in mind, it is easy to see why 16-bit microcontrollers offer higher performance than their 8-bit counterparts. Figure 11.3 illustrates a typical 16-bit CPU dia-

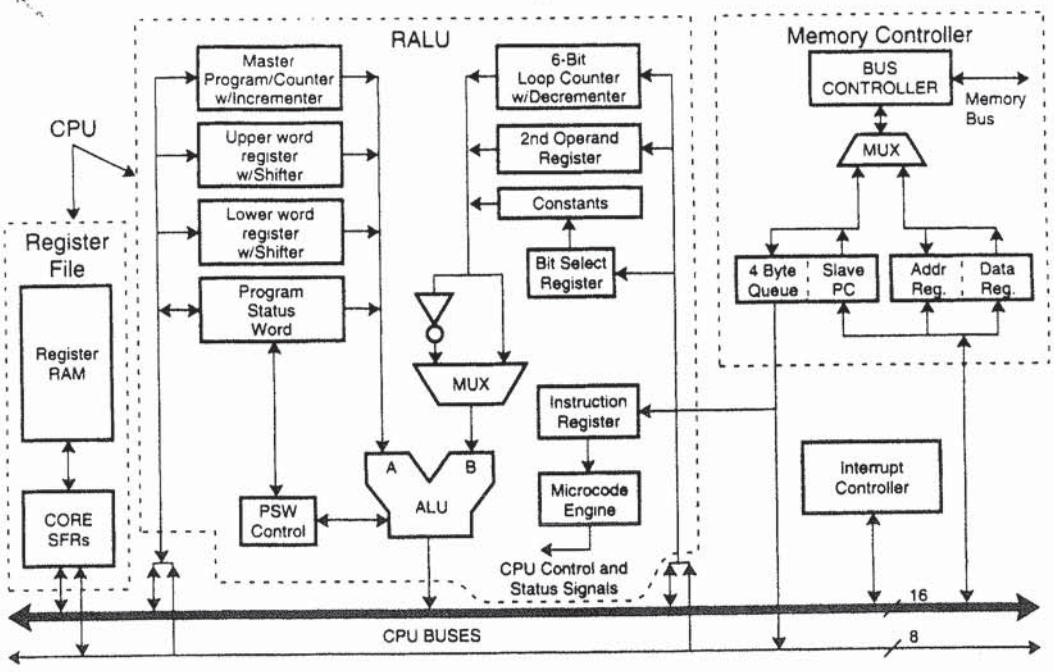


FIGURE 11.3 16-bit CPU.

gram. The microcode engine controls the CPU. Instructions to the CPU are taken from the instruction queue and temporarily stored in the instruction register. This queue is often referred to as a *prefetch queue* and it decreases execution time by staging instructions to be executed. The microcode engine then decodes the instructions and generates the correct sequence of events to have the ALU perform the desired function(s).

Arithmetic Logic Unit. The ALU is the portion of the CPU that performs most mathematical and logic operations. After an instruction is decoded by the microcode engine, the data specified by the instruction is loaded into the ALU for processing. The ALU then processes the data as specified by the instruction.

Register File. The register file consists of memory locations that are used as temporary storage locations while the user's code is executing. The register file is implemented as RAM and consists of both RAM memory locations and special function registers (SFRs). RAM memory locations are used as temporary data storage during execution of the user's code. After power-up, RAM memory locations default to a logic "0" and data in SFR locations contain default values as specified by the microcontroller manufacturer.

Special Function Registers. SFRs allow the user to configure and monitor various peripherals and functions of the microcontroller. By writing specific data to an SFR, the users can configure the microcontroller to meet the exact needs of their application. Figure 11.4 shows an example of a serial port SFR used for configuration. Note that each bit location within the SFR determines a specific function and can be programmed to either a logic "1" or "0". If more than two configuration choices are possible, two or more bits will be combined to produce the multiple choices. An example of this would be the mode bits (M1 and M2) in the example SFR (Fig. 11.4). Bit locations marked "RSV" are reserved and should be written to with a value as indicated by the manufacturer.

SP_CON (1FBBH)

7	6	5	4	3	2	1	0
0	0	PAR	TB8	REN	PEN	M2	M1

M2, M1	Mode	Function
	00	Mode 0: Synchronous
	01	Mode 1: Standard asynchronous
	10	Mode 2: Asynchronous (receiver interrupt on 9th bit = 1)*
	11	Mode 3: Asynchronous (9th bit = parity or data)**
PEN	Parity Enable. Enables the Parity function for Mode 1 or Mode 3; cannot be enabled for Mode 2.	
REN	Receiver Enable. Enables the receiver to write to SBUF_RX.	
TB8	Transmission Bit 8. Set the ninth data bit for transmission (Modes 2 and 3). Cleared after each transmission; not valid if parity is enabled.	
PAR***	0 = even parity 1 = odd parity	
Bits 6, 7	Reserved; write as zeros for future product compatibility.	

* Mode 2: Asynchronous (receiver: interrupt on 9th bit = 1; transmitter: 9th bit = TB8)
** Mode 3: Asynchronous (receiver: always interrupt on 9th bit; transmitter: 9th bit = parity for PEN = 1
*** Par bit only available on 8XC196KT and KS devices. 9th bit = TB8 for PEN = 0
For 8XC196KR, JR, KQ, JQ devices, this bit should be written as a zero to maintain compatibility with future devices.

FIGURE 11.4 Special function control register example.

SP_STAT (1FB9H)

7	6	5	4	3	2	1	0
RB8/RPE	RI	TI	FE	TXE	OE	X	X

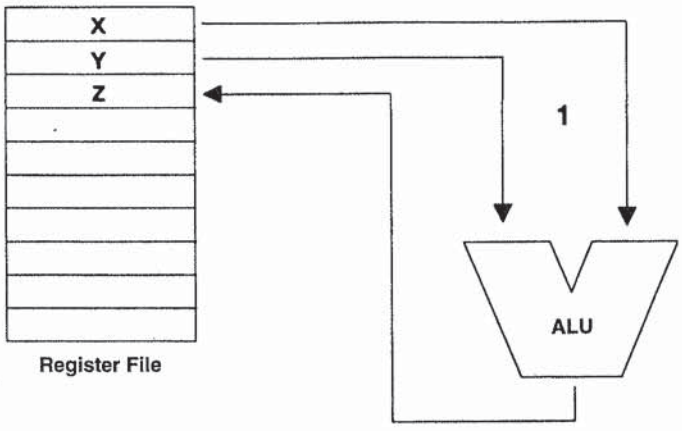
Bits 0, 1	Reserved; ignore data.
OE	Set on buffer overrun error.
TXE	Set on transmitter empty. When set may write 2 bytes to transmit buffer.
FE	Framing error; set it no STOP bit is found at the end of a reception. When set may write 1 byte to transmit buffer.
TI	Transmit interrupt; set at the beginning of the STOP bit transmission.
RI	Receive interrupt; set after the last data bit is received.
RPE	(Parity enabled) Receive parity error (Modes 1 and 3 only); set if parity is enabled and a parity error occurred.
RB8	(Parity disabled) Received Bit 8 (Modes 2 and 3 only); set if the 9th bit is high on reception.

FIGURE 11.5 Special function status register example.

Some SFRs can be read by the user to determine the current status of a given peripheral. Figure 11.5 shows an example of a serial port status register that, when read, indicates the current status of the microcontroller's serial port. Note that each bit location corresponds to a particular state of the serial port. Bit locations marked "RSV" are reserved and should be ignored when read.

Register Direct vs. Accumulator-Based Architectures. Microcontroller architectures can be classified as either the register-direct or accumulator-based type. These terms refer to the means by which the CPU must handle data when performing mathematical, logical, or storage operations.

Register-direct architectures allow the programmer to essentially use most, if not all, of the microcontroller's entire RAM array as individual accumulators. That is, the programmer can perform mathematical or storage operations directly upon any of the RAM locations. This simplifies task switching because program variables may be left in their assigned registers while servicing interrupts. Figure 11.6 illustrates a register-to-register type architecture (such



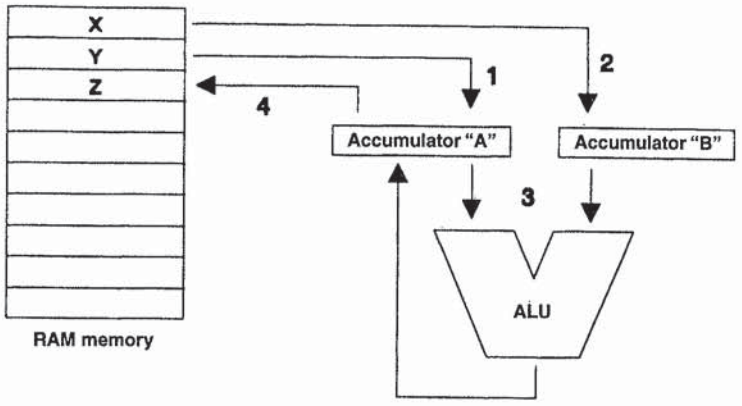
Z = X + Y

```
1. MUL Z,X,Y
```

FIGURE 11.6 Register-to-register architecture example.

as Intel's MCS®-96). This architecture essentially has 232 "accumulators" (more are available through a windowing mechanism) of which any can be operated on directly by the RALU. The true advantage of this type of architecture is that it reduces accumulator bottleneck and speeds throughput during program execution.

Accumulator-based architectures require the user to first store the data to be manipulated into a temporary storage location, referred to as an "accumulator," prior to performing any type of data operation. After the operation is completed, the user program must then store the result to the desired destination location. Figure 11.7 depicts an example of an accumulator-based architecture.




```
Z = X + Y
1. LD A,Y
2. LD B,X
3. MUL A,B
4. ST A,Z
```

FIGURE 11.7 Accumulator-based architecture.

Program Counter. The Program Counter (PC) controls the sequencing of instructions to be executed. The PC is a 16-bit register located within the CPU which holds the address of the next instruction to be executed. After an instruction is fetched, the PC is automatically incremented to point to the next instruction.

SP starting address (SP+12):	
(SP+10):	80FFh
(SP+8):	A5A5h
(SP+6):	6E20h
(SP+4):	5555h
(SP+2):	0000h
SP ending address (SP):	8000h

FIGURE 11.8 Stack pointer example

Stack and Stack Pointer. The stack is an area of memory (typically user-assigned) that is used to store data temporarily in a FILO (first-in, last-out) fashion. The stack is primarily used for storing program information (such as the program counter or interrupt mask registers) when an interrupt service routine is invoked. It is also sometimes used to pass variables between subroutines. The stack is typically accessed through PUSH and POP instructions. Execution of the PUSH instruction "pushes" the contents of the spec-

POP instruction "pops" the contents of the specified operand off of the stack. The stack pointer (SP) is a register which points to the next available word location on the stack. Consider the example shown in Fig. 11.8 which shows the contents of the stack after the following code sequence is executed:

PUSH #80FFh	pushes immediate data 80FFh onto stack
PUSH #0A5A5h	pushes immediate data A5A5h onto stack
PUSH 82h	pushes data @ 82h (assume it's 6E20h) onto stack
PUSH #5555h	pushes immediate data 5555h onto stack
PUSH 4Eh	pushes data @ 4Eh (assume it's 0000h) onto stack
PUSH #8000h	pushes immediate data 8000h onto stack

Continuing with the preceding example, if a POP instruction were executed, the data at the current SP address (SP) would be "popped" off the stack and stored to the address specified by the instruction's operand. Executing the POP instruction results in the SP being incremented by 2.

Program Status Word and Flags. The program status word (PSW) is a collection of boolean flags which retain information concerning the state of the user's program. These flags are set or cleared depending upon the result obtained after executing certain instructions as specified by the microcontroller manufacturer. PSW flags are not directly accessible by the user's program; access is typically through instructions which test one or more of the flags to determine proper program flow. Following is a summary of common PSW flags as supported by Intel's MCS-96® architecture:

Z: The *Zero* flag is set when an operation generates a result equal to zero. The Z flag is never set by the add-with-carry (ADDC/ADDCB) or subtract-with-carry (SUBC/SUBCB) instructions, but is cleared if the result is nonzero. These two instructions are normally used in conjunction with ADD/ADDB and SUB/SUBB instructions to perform multiple-precision arithmetic. The operation of the Z flag for these instructions leaves it indicating the proper result for the entire multiple-precision calculation.

N: The *Negative* flag is set when an operation generates a negative result. Note that the N flag will be in the algebraically correct state even if overflow occurs. For shift operations, the N flag is set to the same value as the most significant bit of the result.

V: The *oVerflow* flag is set when an operation generates a result that is outside the range for the destination data type. For shift-left instructions, the V flag is set if the most significant bit of the operand changes at any time during the shift. For an unsigned word divide, the V flag is set if the quotient is greater than 65,535. For a signed word divide, the V flag is set if the quotient is less than –32,768 or greater than 32,767.

VT: The *oVerflow Trap* flag is set when the V flag is set, but it is only cleared by instructions which are specially designated to clear the VT flag (such as CLRVT, JVT, and JNVT). The VT flag allows for testing possible overflow conditions at the end of a sequence of related arithmetic operations. This is normally more efficient than testing the V flag after each instruction.

C: The *Carry* flag is set to indicate either (1) the state of the arithmetic carry from the most significant bit of the ALU for an arithmetic operation or (2) the state of the last bit shifted out of an operand for a shift. Arithmetic borrow after a subtract operation is the complement of the C flag (i.e., if the operation generated a borrow, then C = 0).

ST: The *STicky* bit flag is set to indicate that, during a right shift, a 1 has been shifted first into the C flag and then shifted out. The ST flag can be used along with the C flag to control rounding after a right shift. Imprecise rounding can be a major source of error in a numerical calculation; use of both the C and ST flags can increase accuracy as described in the following paragraphs.

Consider multiplying two 8-bit quantities and then scaling the result down to 12 bits:

```
MULUB   AX, CL, DL      (CL * DL = AX)
SHR     AX, #4          (AX is shifted right by 4 bits)
```

If the C flag is set after the shift, it indicates that the bits shifted off the end of the operand were greater than or equal to one-half the least significant bit of the 12-bit result. If the C flag is cleared after the shift, it indicates that the bits shifted off the end of the operand were less than half the LSB of the 12-bit result. Without the ST flag, the rounding decision must be made on the basis of the C flag alone. (Normally the result would be rounded up if the C flag is set.) The ST flag allows a finer resolution in the rounding decision as shown here:

C	ST	Bits shifted off
0	0	Value = 0
0	1	0 < Value < ½ LSB
1	0	Value = ½ LSB
1	1	Value > ½ LSB

Jump instructions are the most common instructions to utilize PSW flags for determining the operation to perform. Instructions that test PSW flags are very useful when program flow needs to be altered dependent upon the outcome of an arithmetic operation. The most common example of this would be for program loops that are to be executed a certain number of times. Following are examples of several MCS-96 instructions whose operation is dependent upon the state of one or more program status word flags:

JC (Jump if C flag is set.) If the C (carry) bit is set, the program will jump to the address location specified by the operand. If the C flag is cleared, control will pass to the next sequential instruction.

JGT (Jump if signed greater than.) If both the N (negative) and the Z (zero) flags are clear, the program will jump to the address location specified by the operand. If either of the flags is set, control will pass to the next sequential instruction.

JLE (Jump if signed less than or equal.) If either the N or Z flags is set, the program will jump to the address location specified by the operand. If both the N and Z flags are cleared, control will pass to the next sequential instruction.

11.1.4 Bus Controller

The bus controller serves as the interface between the CPU and the internal program memory and the external memory spaces. The bus controller maintains a queue (commonly called the prefetch queue) of prefetched instruction bytes and responds to CPU requests for data memory references. The prefetch queue decreases execution time by staging instructions to be executed. The capacities of prefetch queues vary but for the MCS-96 architecture, it is 4 bytes deep.

When using a logic analyzer to debug code it is important to consider the effects of the prefetch queue. It is not possible to accurately determine when an instruction will execute by simply watching when it is fetched from external memory. This is because the prefetch queue is filled in advance of instruction execution. It is also important to consider the effects when a jump or branch occurs during program execution. When the program sequence changes because of a jump, interrupt, call, or return, the PC is loaded with the new address, the queue is flushed and processing continues. Consider the situation in which the external address/data bus is being monitored when a program branch occurs. Because of the prefetch queue, it will appear as if instructions past the branch point were executed, when in fact they were only loaded into the prefetch queue.

11.1.5 Frequency of Operation

Microcontrollers are being offered in an ever-increasing range of operating frequencies. Most high-end automotive applications currently use microcontrollers operating in the 12- to 20-MHz range, with 24 MHz becoming not so uncommon. Microcontrollers with frequencies as high as 30 and 32 MHz are available as prototypes and will soon be available for production. Operating frequency becomes especially important when a microcontroller must perform high-speed event control such as required in ABS braking and engine control. Applications such as these typically have to detect, calculate, and respond to external events within a given amount of time. In ABS applications, this time is commonly referred to as loop time and defines the amount of time that the microcontroller has to execute the main loop of the software algorithm to achieve optimal performance.

Operating frequency can be directly related to the speed at which a microcontroller will execute the user's code. For instance, let's look at how long it takes for a particular microcontroller to execute the following generic subroutine. For this example, consider the execution times rather than the operations each instruction is performing.

```
{6}  PUSHF
{3}  NOTB   PTS_COUNT_EPA1
{5}  ADDB   NUM_OF_PULSES_1, PTS_COUNT_EPA1, #00h
{5}  SUB    INV_SPEED_1, FTIME_1, ITIME_1
{27} DIV    INV_SPEED_1, NUM_OF_PULSES_1
{5}  LD     Temp1+2, #Speed_high_constant
{5}  LD     Temp1, #Speed_low_constant
{27} DIV    Temp1, INV_SPEED_1
{4}  ST     Temp1, EPA1_FREQ
{11} RET
```

In this example, the numbers in brackets {} denote how many state times it will take the microcontroller to execute the given line of code. A state time is the basic time measurement for all microcontroller operations. For this MCS-96 family microcontroller, a state time is based on the crystal frequency divided by two. A state time for other microcontrollers may be based upon the crystal frequency divided by three. For this particular microcontroller, a state time can be calculated by the following formula (other microcontroller families use similar formulas):

$$1 \text{ state time} = 1[(\text{frequency of operation})/2]$$

Applying this formula, 1 state time = 125 ns when operating at 16 MHz, and 167 ns when operating at 12 MHz. The example code sequence takes the microcontroller 98 state times to execute. This equates to 16.37 μs to execute at an operating frequency of 12 MHz. At 16 MHz, it takes only 12.25 μs for the microcontroller to execute the subroutine. An operating frequency of 16 MHz results in the microcontroller executing approximately 34 percent more instructions in a given time than at a frequency of 12 MHz.

Another consideration when choosing an operating frequency is the clocking resolution of on-chip timer/counters. The maximum clocking rate of on-chip timer/counters is limited by the frequency the microcontroller is being clocked at. As an example, if an on-chip timer/counter is set up to increment/decrement at a rate determined by CLOCK/4, this would result in 333 ns resolution at 12 MHz. However, if the clock speed were increased to 16 MHz, a higher and more desirable resolution of 250 ns is achieved.

11.1.6 Instruction Set

An often overlooked feature that gives a microcontroller the capability to perform desired operations and manipulate data is its instruction set. A microcontroller's instruction set con-

sists of a set of unique commands which the programmer uses to instruct the microcontroller on what operation to perform.

An *instruction* is a binary command which is recognized by the CPU as a request to perform a certain operation. Examples of typically supported operations are loads, moves, and stores which transfer data from one memory location to another. There are also jumps and branches which are used to alter program flow. Arithmetic instructions include various multiples, divides, subtracts, additions, increments, and decrements. Instructions such as ANDs, ORs, XORs, shifts, and so forth, allow the user to perform logical operations upon data. In addition to these basic instructions, microcontrollers often support specialized instructions unique to their architecture or intended application.

Instructions can be divided into two parts, the *opcode* and *operand.* The opcode (sometimes referred to as the machine instruction) specifies the operation to take place and the operand specifies the data to be operated upon. Instructions typically consist of either 0, 1, 2, or 3 operands to support various operations. As an example, consider the following MCS-96 architecture instructions:

PUSHF (0 operands) is an instruction that pushes the program status word (PSW) onto the stack. Since this instruction operates on a predefined location, no operand is necessary.

Format: PUSHF

PUSH (1 operand) is an instruction that pushes the specified word operand onto the stack.

Format: PUSH (SRC)

ADD (2 operands) adds two words together and places the result in the destination (leftmost) operand location.

Format: ADD (DST),(SRC)

ADD (3 operands) adds two words together as the 2-operand ADD instruction, but in this case, a third operand is specified as the destination.

Format: ADD (DEST),(SRC1),(SRC2)

Instructions support one or more of six basic addressing types to access operands within the address space of the microcontroller. If programmers wish to take full advantage of a microcontroller architecture, it is important that they fully understand the details of the supported addressing types. The six basic types of addressing modes are termed register-direct, indirect, indirect with autoincrement, immediate, short-indexed, and long-indexed. The following descriptions describe these modes as they are handled by hardware in register-to-register architectures.

The *register-direct* addressing mode is used to directly access registers within the lower 256 bytes of the on-chip register file. The register is selected by an 8-bit field within the instruction and the register address must conform to the operand type's alignment rules. Depending upon the instruction, typically up to three registers can take part in the calculation.

Examples:

ADD AX,BX,CX	AX = BX + CX
MUL AX,BX	AX = AX*BX
INCB CL	CL = CL + 1

The *indirect addressing* mode accesses a word in the lower register file containing the 16-bit operand address. The indirect address can refer to an operand anywhere within the address space of the microcontroller. The register containing the indirect address is selected by an 8-bit field within the instruction. An instruction may contain only one indirect reference; the remaining operands (if any) must be register-direct references.

Examples:

LD BX,[AX] BX = mem_word(AX)

In this example, assume that before execution:

contents of AX = 2FC2h

contents of 2FC2h = 3F26h

Then after execution,

contents of BX = 3F26h

ADDB AL,BL,[CX] AL = BL + mem_byte(CX)

The *indirect with autoincrement* addressing mode is the same as the indirect mode except that the variable that contains the indirect address is autoincremented after it is used to address the operand. If the instruction operates on bytes or short integers, the indirect address variable is incremented by one; if it operates on words or integers, the indirect address will be incremented by two.

Examples:

LD BX,[AX]+	BX = mem_word(AX)
	AX = AX + 2
ADDB AL,BL,[CX]+	AL = BL + mem_byte(CX)
	CX = CX + 1

For the *immediate addressing* mode, an operand itself is in a field in the instruction. An instruction may contain only one immediate reference; the remaining operand(s) must be register-direct references.

Example:

ADD AX,#340 AX = AX + 340 (decimal)

For the *short-indexed addressing* mode, an 8-bit field in the instruction selects a word variable (which is contained in square brackets) in the lower register file that contains an address. A second 8-bit field in the instruction stream is sign-extended and summed with the word variable to form an operand address.

Since the 8-bit field is sign-extended, the effective address can be up to 128 bytes before the address in the word variable and up to 127 bytes after it. An instruction may contain only one short-indexed reference; the remaining operand(s) must be register-direct references.

Example:

LD AX,4[BX] AX = mem_word(BX + 4)

In this example, assume that before execution:

contents of BX = A152h

The operand address is then A152h + 04h = A156h

The *long-indexed addressing* mode is like the short-indexed mode except that a 16-bit field is taken from the instruction and added to the word variable to form the operand. No sign extension is necessary. An instruction may contain only one long-indexed reference and the remaining operand(s) must be register-direct references.

Examples:

ST AX,TABLE[BX]	mem_word(TABLE + BX) = AX
AND AX,BX,TABLE[CX]	AX = BX and mem_word(TABLE + CX)

11.1.7 Programming Languages

The two most common types of programming languages in use today for automotive microcontrollers are *assembly languages* and *high-level languages* (HLLs). Program development begins with the user writing code in either an assembly language or an HLL. This code is written as a text file and is referred to as a source file. The source file is then assembled or compiled using the appropriate assembler/compiler program. The assembler translates the source code into object code and creates what is referred to as an object file. The object file contains machine language instructions and data that can be loaded into an evaluation tool for debugging and validation. The object can also be converted into a hex file for EPROM programming or ROM mask generation as discussed later in this chapter. The program development process is illustrated in Fig. 11.9.

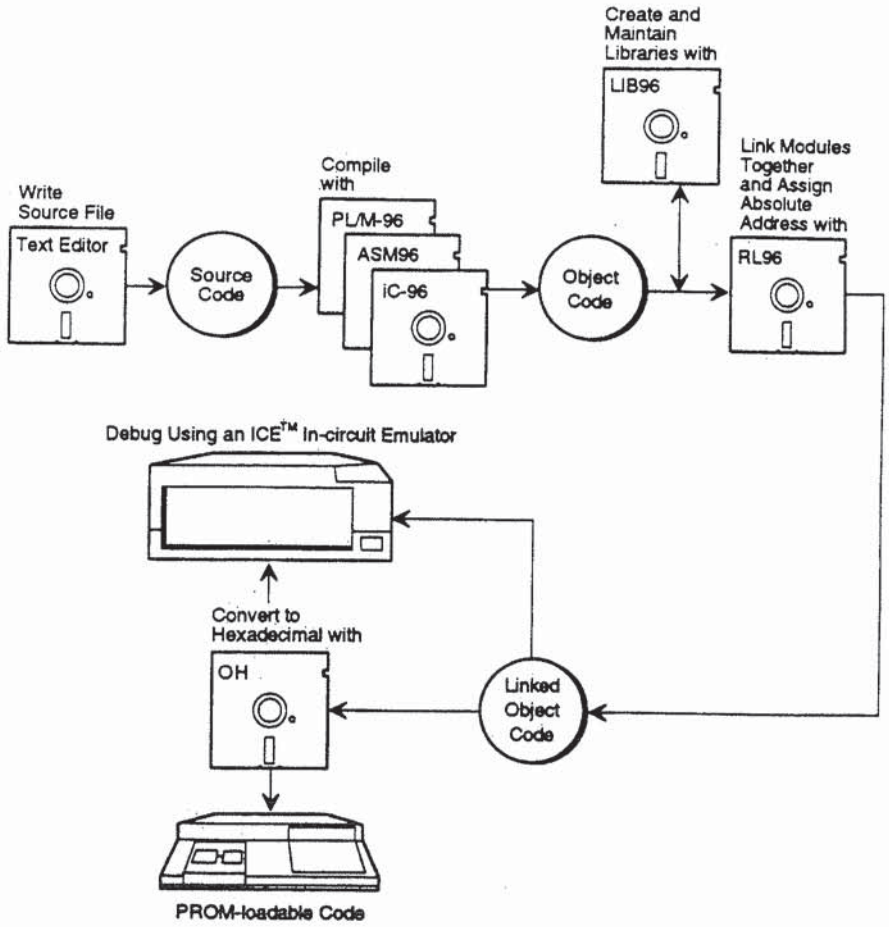


FIGURE 11.9 The program development process.

Assembly Language Programming. An assembly language is a low-level programming language that is specific to a given microcontroller family. Assemblers translate language operation codes (mnemonics) directly into machine instructions that instruct the microcontroller

on what operation to perform. Because the programmer is essentially using the microcontroller's machine code to write assembly language programs, more precise control of the device can be achieved through the direct manipulation of individual bits within registers. Because of their efficiency, assembly language programs require less code space than high-level languages. Assembly language programs consist of three parts: machine instructions, assembler directives, and assembler controls.

A *machine instruction* is a machine code that can be executed by the microcontroller's CPU. The collection of machine instructions that a particular microcontroller can execute is referred to as its instruction set. An example of a machine instruction is the opcode for the MULB instruction (Fig. 11.10) from Intel's MCS-96 assembly language. MULB is the mnemonic that represents the machine instruction which performs the specified multiplication operation. When executed by the microcontroller, the MULB opcode results in the multiplication of the two byte operands with the result being placed in a word destination location.

MULB (Three Operands)

Format

```
MULB   wreg,breg,baop
```

Operation

The second and third byte operands are multiplied using signed arithmetic and the 16-bit result is stored into the destination (leftmost) operand. The sticky bit flag is undefined after the instruction is executed.

```
(DEST) ← (SRC1) * (SRC2)
```

Opcode Pattern

11111110	01011aa	baop	breg	wreg

Flags Affected

ST

Examples

```
MULB    DELTA, TIMER1, #2
MULB    ALPHA, BETA, GAMMA
MULB    ALPHA, DELTA, 10[GAMMA]
```

FIGURE 11.10 Machine instruction example: MULB.

Assembler directives allow the user to specify auxiliary information (such as storage reservation, location counter control, definition of nonexecutable code, object code relocation, and flow of assembler processing) that determines the manner in which the assembler generates object code from the user's source file input.

Assembler controls set the mode of operation for the assembler and direct the flow of the assembly process. Assembler controls can be classified into primary controls and general controls. Primary controls are set at the beginning of the assembly process and cannot be changed during the assembly. Primary controls allow the user to specify items such as print options, page lengths and widths, error messages, and cross-referencing. General controls can be specified in the invocation line or on control lines anywhere in the source file and can appear any number of times in the program. General controls either cause an immediate action or an immediate change of conditions in which the condition specified remains in effect until another general control causes it to change.

High-level Language Programming. Unlike low-level languages (such as assembly languages), a high-level language is a general purpose language that can support numerous microcontroller architectures. The most common high-level language used for automotive

applications is C. C programs are written with statements rather than specific instructions from a microcontroller's instruction set. High-level languages utilize a software program known as a *compiler* to translate the user's source code into the specific microcontroller's machine language. Each microcontroller family has its own unique compiler to support selected high-level languages. Although high-level languages tend to be less efficient than assembly languages, their advantage lies in ease of writing code and better debugging capability. The use of statements as opposed to specific instructions better suits high-level languages toward control of procedures (to implement complex software algorithms) as opposed to the microcontroller itself.

11.1.8 Interrupt Structure

The interrupt structure is one of the more important features of an automotive microcontroller. Applications such as automotive ABS and engine control can be referred to as event-driven control systems. Event-driven control systems require that normal code execution be halted to allow a higher-priority task or event to take place. These higher-priority tasks are known as interrupts and can initiate a change in the program flow to execute a specialized routine. When an interrupt occurs, instead of executing the next instruction, the CPU branches to an interrupt service routine (ISR). The branch can occur in response to a request from an on-chip peripheral, an external signal, or an instruction. In the simplest case, the microcontroller receives the request, performs the desired operation and returns to the task that was interrupted.

ISRs are typically serviced via software but it is becoming common for microcontroller manufacturers to implement special on-chip hardware ISR functions for commonly performed operations. These ISRs are typically microcoded or *hardwired* into the microcontroller as described later in this section.

Software, or Normal, Servicing of Interrupts. The software servicing of interrupts is fairly straightforward as shown in Fig. 11.11. When an interrupt source is enabled by the user and a

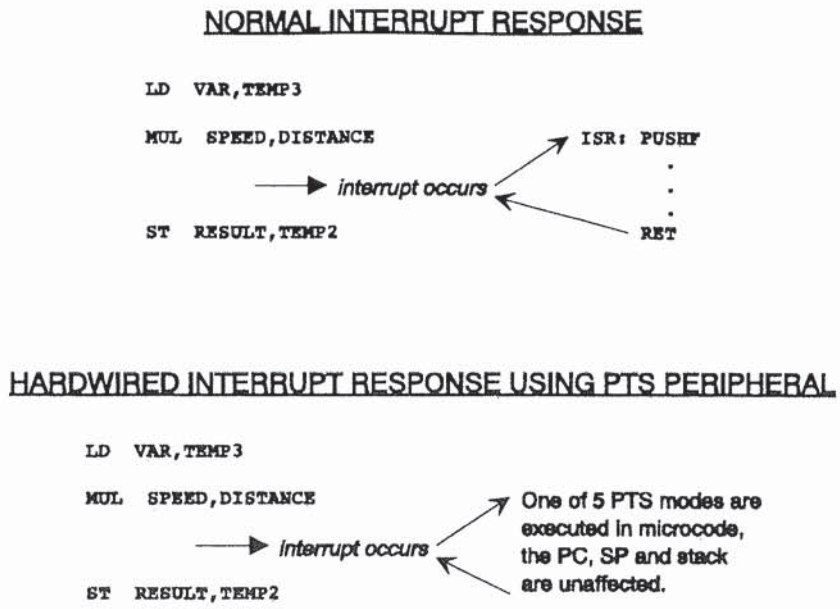


FIGURE 11.11 Comparison of normal interrupts and hardwired interrupts.

valid interrupt event occurs, the CPU will fetch the starting address of the ISR from the interrupt vector table. The interrupt vector table is a dedicated section of memory that contains the user-programmed start address of the various ISRs. After fetching the ISR address, the CPU automatically pushes the current program counter (PC) onto the stack and loads the PC with the ISR beginning address. This results in the program flow vectoring to the ISR address. The user-programmed ISR is then executed. The last instruction within the ISR is a return instruction that pops the old PC off the stack. This results in program flow continuing from where it was interrupted.

Interrupt mask registers allow the user to prevent or *mask* undesirable interrupts from occurring during various sections of the program. This is a very desirable feature and allows for custom tailoring of the interrupt structure to meet the needs of a particular application. Enabling or disabling of all interrupts (known as globally enabling/disabling) is typically supported with a software instruction such as DI (globally disable all interrupts) or EI (globally enable all interrupts).

Hardware, or Microcoded, Interrupt Structures. Hardware interrupt structures differ from software interrupts in that the user doesn't have to provide the ISR to be executed when the interrupt occurs. With a hardware interrupt structure, the ISR is predefined by being hardwired or *microcoded* into the microcontroller. This is advantageous because it requires less code space and requires less CPU overhead. Stack operations are not necessary since interrupt vectors do not have to be fetched. Most microcontroller manufacturers have their own proprietary solution for hardware ISR's, which are all somewhat similar to one another. For purposes of this section, we will briefly describe the peripheral transaction server as implemented on members of Intel's MCS-96 family of microcontrollers.

The PTS provides a microcoded hardware interrupt handler which can be used in place of a normal ISR. The PTS requires much less overhead than a normal ISR since it operates without modification of the stack. Any interrupt source can be selected by the user to trigger a PTS interrupt in place of a normal ISR. The PTS is similar to a direct memory access (DMA) controller in that when a PTS interrupt, or *cycle,* occurs, data is automatically moved from one location of memory to another as specified by the user. Figure 11.11 compares a regular ISR to a PTS interrupt cycle.

The PTS allows for five modes of operation; single-byte transfer, multiple-byte transfer, PWM, PWM toggle, and A/D scan mode. Each mode is configurable through an 8-byte, user-defined PTS control block (PTSCB) located in RAM. The user may enable virtually any normal interrupt source to be serviced by a PTS interrupt by simply writing to the appropriate bit in an SFR known as the PTS_SELECT register. When a PTS interrupt is enabled and the event occurs, a microcoded interrupt service routine executes in which the contents of the PTSCB are read to determine the specific operation to be performed. More details on the PTSCB can be found in the application example found in this section.

The major advantage of the PTS for automotive applications is its fast response time. The PTS is ideally suited for transferring single or multiple bytes/words of data in response to an interrupt. An example of this is the serial port example which will be described shortly. Another example of the usefulness of the PTS (using A/D scan mode) would be if the user wanted to automatically store A/D conversion results every time a conversion completed within a user-defined scan of A/D channels. The PTS could also be configured to automatically transfer a block of data between memory locations every time an interrupt occurs.

Application Example of PTS Single-Byte Transfer Mode. This example shows how the PTS can be used to automatically transmit and receive 8-byte messages over the serial port. Data to be transmitted and received data are stored in separate tables. The use of the PTS for this purpose greatly reduces CPU overhead and code-space requirements. The layout of the user-defined PTSCB for single-byte transfer mode is shown in Fig. 11.12. PTS_DEST within the PTSCB contains the destination address for the data transfer and PTS_SOURCE contains the source address for the transfer.

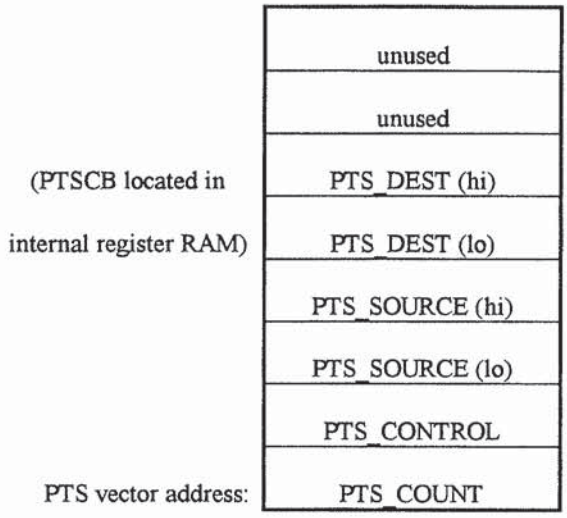


FIGURE 11.12 PTS control block for single-byte transfer mode.

Two PTSCBs are set up for this example, one in response to receive (RX) interrupts and one in response to transmit (TX) interrupts. The RX PTSCB's PTS_DEST is initialized with the start address of the receive data table and the TX PTSCB's PTS_DEST is initialized with the address of the serial port's transmit buffer.

PTS_CONTROL is a byte that specifies the PTS operation to be performed. Its layout is shown in Fig. 11.13.

PTS_COUNT is a down counter that is used to keep track of how many PTS interrupts or cycles have occurred since the last initialization. PTS_COUNT is initialized by the user to any value below 256 and is decremented everytime the corresponding PTS cycle occurs. It is often used to keep track of how many pieces of data have been transferred. In this example, PTS_COUNT is used to determine when a complete 8-byte message has been transmitted or received. After PTS_COUNT expires, an "end-of-PTS" or "normal" ISR occurs, in which the user utilizes the data as required by the application. When an interrupt source is enabled by the user to be a PTS interrupt, the following sequence of events occurs every time the corresponding interrupt occurs:

1. Instead of a normal interrupt, the user has selected it to do a PTS cycle.
2. The microcoded PTS routine fetches the PTS_CONTROL byte from the PTSCB whose start address is specified by the user in the PTS interrupt vector table. The microcoded PTS routine then:

   reads data to be transferred from address specified by PTS_SOURCE

   writes the data to address specified by PTS_DEST

   optionally increments/updates PTS_SOURCE and PTS_DEST addresses

   decrements PTS_COUNT
3. When PTS_COUNT reaches "0", an end of PTS interrupt occurs and the normal ISR is executed in which the user utilizes the received data as necessary (for RX interrupts) or reloads the transmit table with new data (for TX interrupts).

Interrupt Latency. Interrupt latency is defined as the time from when the interrupt event occurs (not when it is acknowledged) to when the microcontroller begins executing the first

PTS_CONTROL BYTE (for single and multiple-byte transfers)

7	6	5	4	3	2	1	0
M2	M1	M0	B/W	SU	DU	SI	DI

M2, M1, M0:	Mode	Function
	000	PTS Block Transfer
	100	PTS Single Transfer

B/W:	Byte/Word:	"0" = Word; "1" = Byte
SU:	Source Update:	"1" = update source address
DU:	Destination Update:	"1" = update destination address
SI:	Increment Source:	"1" = Increment Source address
DI:	Increment Destination:	"1" = Increment Destination address

FIGURE 11.13 PTS control byte for single- and multiple-byte transfer modes.

instruction of the interrupt service routine. Interrupt latency must be carefully considered in timing-critical code as is found in many automotive applications.

There is a delay between an interrupt's triggering and its acknowledgment. An interrupt is not acknowledged until the currently executing instruction is finished. Further, if the interrupt signal does not occur at least some specified (assume four for this discussion) state times before the end of the current instruction, the interrupt may not be acknowledged until after the next instruction has been executed. This is because an instruction is fetched and prepared for execution a few state times before it is actually executed. Thus, the maximum delay between interrupt generation and its acknowledgment is approximately four state times plus the execution time of the next instruction.

It should also be noted that most microcontrollers have protected instructions (such as RETURN, PUSH, POP) which inhibit interrupt acknowledgment until after the following instruction is executed. These instructions can increase interrupt-to-acknowledgment delay.

When an interrupt is acknowledged, the interrupt pending bit is cleared and a call is forced to the location indicated by the corresponding interrupt vector. This call occurs after the completion of the current instruction, except as noted previously. For the MCS-96 architecture, the procedure of fetching the interrupt vector and forcing the call requires 16 state times. The stack being located in external memory will add an additional two state times to this number.

Latency is the time from when an interrupt is generated (not acknowledged) until the microcontroller begins executing interrupt code. The maximum latency occurs when an inter-

rupt occurs too late for acknowledgment following the current instruction. The worst case is calculated assuming that the current instruction is not a protected one. The worst-case latency is the sum of three terms:

1. The time for the current instruction to finish (assume four state times).
2. The state times required for the next instruction. This time is basically the time it takes to execute the longest instruction used in the user's code (assume it's a 16-state DIV instruction).
3. The response time (assume 16 states, 18 for an externally located stack).

Thus, for this scenario, the maximum delay would be 4 + 16 + 16 = 36 state times. This equates to approximately 4.5 μs for a MCS-96 microcontroller operating at 16 MHz. This latency can increase or decrease depending upon the longest execution-time instruction used. Figure 11.14 illustrates an example of this worst-case scenario.

Interrupt latency can be reduced by carefully selecting instructions in areas of code where interrupts are expected. Using a protected instruction followed immediately by a long instruction increases the maximum latency because an interrupt cannot occur after the protected instruction.

11.1.9 Fabrication Processes

The basic fabrication processes that are widely used for automotive microcontrollers today are NMOS (N-channel metal-oxide semiconductor) and CMOS (complementary MOS). The scope of this chapter does not allow for an in-depth discussion of these processes, although a brief description of the structures used to build on-chip circuitry will be discussed. These terms refer to the components used in the construction of MOSFET (MOS field effect transistor) inverters which are the basis of logic on digital devices. NMOS inverters are constructed of N-channel transistors only, whereas CMOS inverters are constructed of both N-channel and P-channel transistors. This section will describe the basic operation of each inverter along with its pros and cons.

Simply stated, a P-channel transistor conducts when a logic "0" is applied to its gate. Conversely, N-channel transistors conduct when a logic "1" is applied to their gate. Figures 11.15 and 11.16 show a simplified cross-sectional view and the electrical symbol for N- and P-channel devices, respectively.

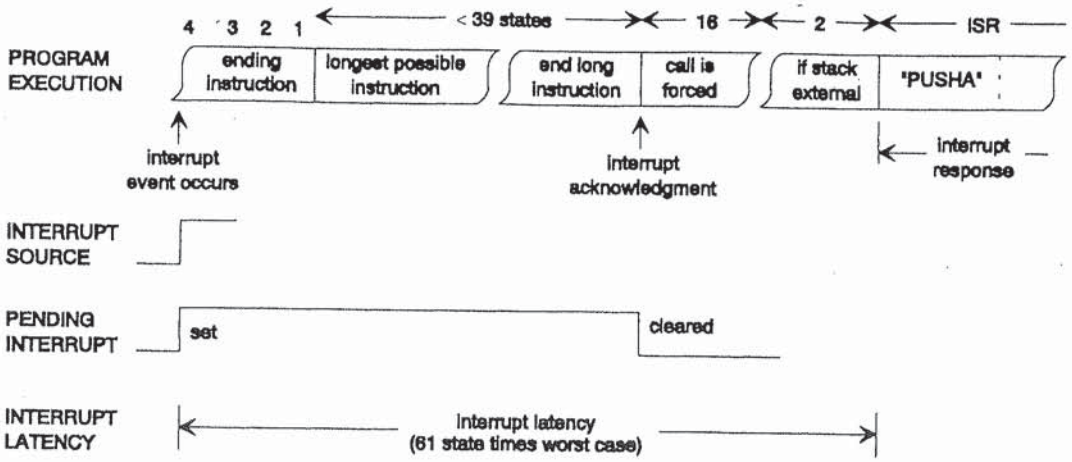


FIGURE 11.14 Worst case interrupt latency example.

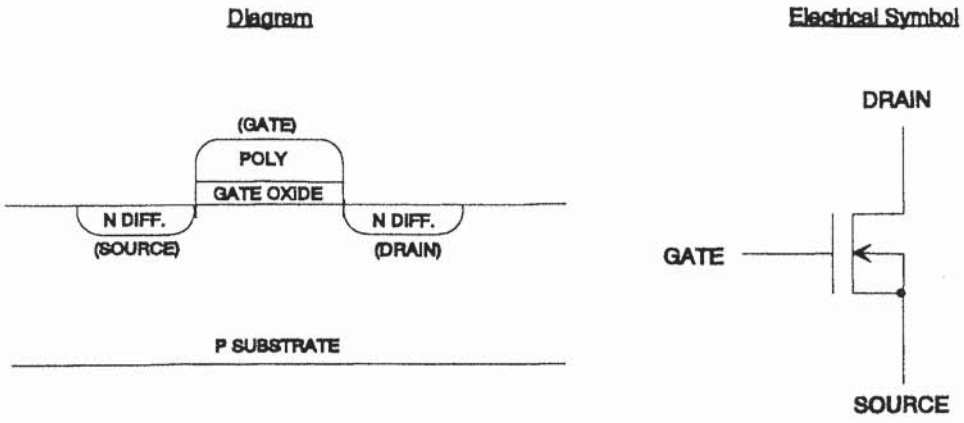


FIGURE 11.15 N-channel transistor.

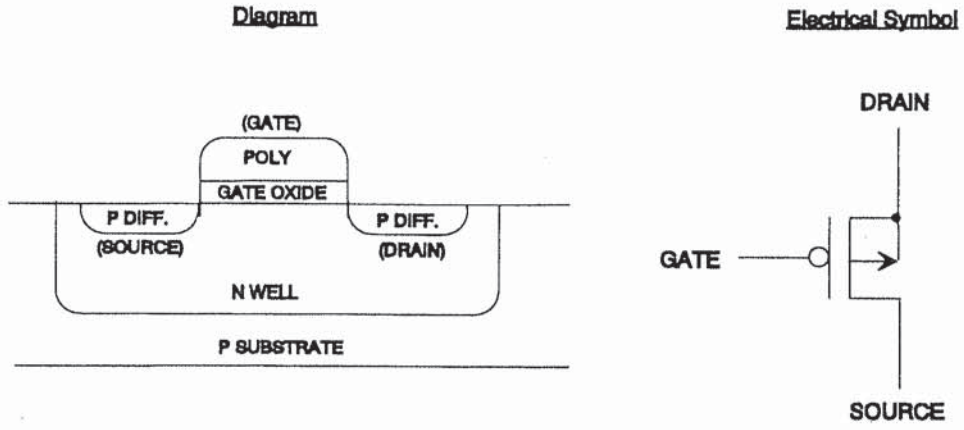


FIGURE 11.16 P-channel transistor.

NMOS Inverters. NMOS inverters are constructed of two NMOS transistors in which one is utilized as a resistance (Q2) and the other is utilized as a switch (Q1). A depletion-mode NMOS transistor is commonly utilized for the resistance device. A basic NMOS inverter is shown in Fig. 11.17. Note that Q2 is always on and acts as a resistor.

When a logic "0" is applied to the inverter's input, Q1 is turned off, which results in Q2 driving a logic 1 at the output. When a logic "1" is applied to the inverter's input, Q1 is turned on and overcomes Q2. This results in a logic "0" at the output.

NMOS microcontrollers are still produced in large quantities today. An advantage of NMOS processes is the simplistic circuit configuration which results in higher chip densities. NMOS devices are also less sensitive to electrostatic discharge (ESD) than CMOS devices. An inherent disadvantage of NMOS design is the slower switching speeds and higher power dissipation due to the dc current path from power to ground through Q1 and Q2 when the inverter is driving a logic "0".

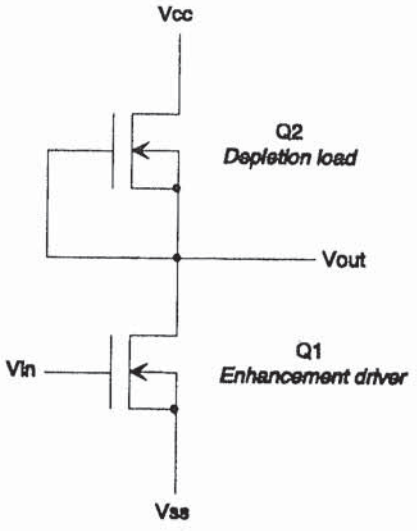


FIGURE 11.17 NMOS inverter.

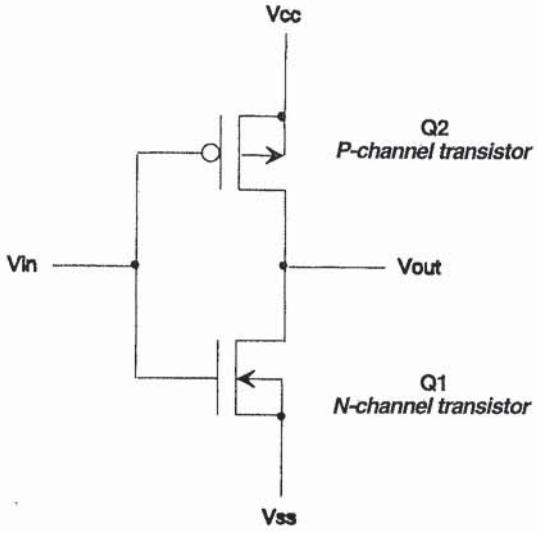


FIGURE 11.18 CMOS inverter.

CMOS Inverters. The CMOS is the most widely used process for automotive microcontrollers today. CMOS inverters are constructed of both P-channel and N-channel transistors that have their inputs tied together as shown in Fig. 11.18. When a logic "0" is applied to the inverter's input, Q1 is turned off and Q2 is turned on, which results in Q2 driving a logic "1" at the output. When a logic "1" is applied to the inverter's input, Q2 is turned off and Q1 is turned on, which results in Q1 driving a logic "0" at the output. Note that only one of these two devices will conduct at a time when the input is "1" or "0". While the input switches, both Q1 and Q2 may conduct for a short time resulting in a small amount of power dissipation.

The main advantages of CMOS logic are greatly improved switching times and lower power consumption, which is due to the complementary design of the inverter. A disadvantage of CMOS logic is that it is more expensive due to its increased complexity and more demanding fabrication process. CMOS logic is more susceptible to ESD damage, although microcontroller manufacturers have countered this by incorporating very effective ESD protection devices onto the silicon.

11.1.10 Temperature Range

Another important factor that must be considered when choosing a microcontroller is the temperature range in which it will be required to operate. The two most common temperature specifications specified by microcontroller manufacturers are *ambient temperature under bias* (TA) and *storage temperature*. These specifications are based upon package thermal characteristics as determined through device and package testing. Storage temperature refers to the temperature range that a microcontroller can be subjected to during periods of nonoperation. Storage temperature specifications are more extreme than ambient temperature under bias temperatures and are usually all the same regardless of the specified ambient temperature range. The common storage temperature range in industry is –60 to +150 °C. While powered-down, a given microcontroller must not be subjected to temperatures that exceed its specified storage temperature range.

Ambient temperature under bias (TA) refers to the temperature range that the microcontroller is guaranteed to operate at within a given application. While powered-up or operating, a microcontroller must not be subjected to temperatures that exceed its specified ambient temperature range. The most common ambient temperature ranges in industry are:

Commercial	0 to +70 °C
Extended	–40 to +85 °C
Automotive	–40 to +125 °C

11.2 MEMORY

Microcontrollers execute customized programs that are written by the user. These programs are stored in either on-chip or off-chip memory and are often referred to as the *user's code.* On-chip memory is actually integrated onto the same piece of silicon as the microcontroller and is accessed over the internal data bus. Off-chip memory exists on a separately packaged piece of silicon and is typically accessed by the microcontroller over an external address/data bus.

A memory map shows how memory addresses are arranged in a particular microcontroller. Figure 11.19 shows a typical microcontroller memory map.

<table>
<tr><th>Address</th><th colspan="3">Memory Function</th></tr>
<tr><td>0FFFFh
0A000h</td><td colspan="3">External Memory</td></tr>
<tr><td>9FFFh
2080h</td><td colspan="3">Internal ROM/EPROM or External Memory</td></tr>
<tr><td>207Fh
2000h</td><td colspan="3">Internal ROM/EPROM or External Memory
(Interrupt vectors, CCB's, Security Key, Reserved locations, etc.)</td></tr>
<tr><td>1FFFh
1F00h</td><td colspan="3">Internal Special Function Registers (SFR's)</td></tr>
<tr><td>1EFFh
0600h</td><td colspan="3">External Memory</td></tr>
<tr><td>05FFh
0400h</td><td colspan="3">INTERNAL RAM (Address with indirect or indexed modes.)
(Also know as Code RAM)</td></tr>
<tr><td>03FFh
0100h</td><td>Register RAM</td><td>Upper Register File (Address with indirect or indexed modes or through windows.)</td><td rowspan="3">Register File</td></tr>
<tr><td>00FFh
0018h</td><td>Register RAM</td><td rowspan="2">Lower Register File (Address with direct, indirect or indexed modes.)</td></tr>
<tr><td>0017h
0000h</td><td>CPU SFRs</td></tr>
</table>

FIGURE 11.19 Microcontroller memory map.

Memory is commonly referred to in terms of Kbytes of memory. One Kbyte is defined as 1024 bytes of data. Memory is most commonly arranged in bytes which consist of 8 bits of data. For instance, a common automotive EPROM is referred to as a "256k × 8 EPROM". This EPROM contains 256-Kbytes 8-bit memory locations or 2,097,152 bits of information.

11.2.1 On-Chip Memory

On-chip microcontroller memory consists of some mix of five basic types: random access memory (RAM), read-only memory (ROM), erasable ROM (EPROM), electrically erasable ROM (EPROM), and flash memory. RAM is typically utilized for run-time variable storage and SFRs. The various types of ROM are generally used for code storage and fixed data tables.

The advantages of on-chip memory are numerous, especially for automotive applications, which are very size and cost conscious. Utilizing on-chip memory eliminates the need for external memory and the "glue" logic necessary to implement an address/data bus system. External memory systems are also notorious generators of switching noise and RFI due to their high clock rates and fast switching times. Providing sufficient on-chip memory helps to greatly reduce these concerns.

RAM. RAM may be defined as memory that has both read and write capabilities so that the stored information can be retrieved (read) and changed by applying new information to the cell (write). RAM found on microcontrollers is that of the static type that uses transistor cells connected as flip-flops. A typical six-transistor CMOS RAM cell is shown in Fig. 11.20. It consists of two cross-coupled CMOS inverters to store the data and two transmission gates, which provide the data path into or out of the cell. The most significant characteristic of static memory is that it loses its memory contents once power is removed. After power is removed, and once it is reapplied, static microcontroller RAM locations will revert to their default state of a logic "0". Because of the number of transistors used to construct a single cell, RAM memory is typically larger per bit than EPROM or ROM memory.

Although code typically cannot be executed from register RAM, a special type of RAM often referred to as *code RAM* is useful for downloading small segments of executable code. The difference between code and register RAM is that code RAM can be accessed via the

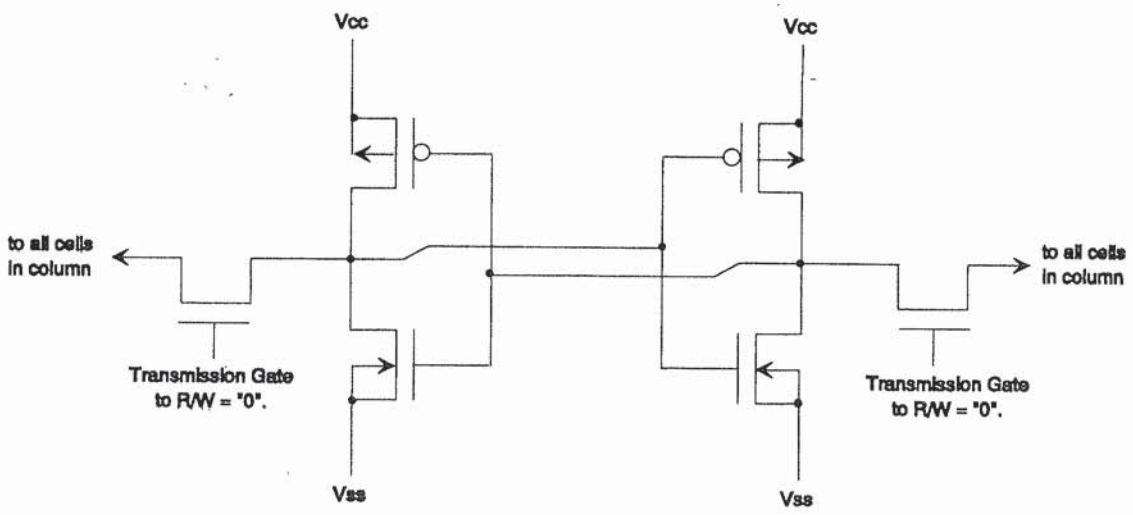


FIGURE 11.20 CMOS RAM memory cell.

memory controller, thus allowing code to be executed from it. Code RAM is especially useful for end-of-line testing during ECU manufacturing by allowing test code to be downloaded via the serial port peripheral.

ROM. Read-only memory (ROM), as the name implies, is memory that can be read but not written to. ROM is used for storage of user code or data that does not change since it is a nonvolatile memory that retains its contents after power is removed. Code or data is either entered during the manufacturing process (masked ROM, or MROM) or by later programming (programmable ROM, or PROM); either way, once entered it is unalterable.

A ROM cell by itself (Fig. 11.21) is nothing more than a transistor. ROM cells must be used in a matrix of word and bit lines (as shown in Fig. 11.22) in order to store information. The word lines are connected to the address decoder and the bit lines are connected to output buffers. The user's code is permanently stored by including or omitting individual cells at word and bit line junctions within the ROM array. For MROMs, this is done during wafer fabrication. For PROMs, this is done by blowing a fuse in the source/drain connection of each cell. To read an address within the array, the address decoder applies the address to the memory matrix. For any given intersection of a word and bit line, the absence of a cell transistor allows no current to flow and causes the transistor to be off. This indicates an unprogrammed ROM cell. The presence of a complete cell conducts and is sensed as a logical "0", indicating a programmed cell. The stored data on the bit lines is then driven to the output buffers.

MROMs are typically used for applications whose code is stable and in volume production. After the development process is complete and the user's program has been verified, the user submits the ROM code to the microcontroller manufacturer. The microcontroller manufacturer then produces a mask that is used during manufacturing to permanently embed the program within the microcontroller. This mask layer either enables or disables individual ROM cells at the junctions of the word and bit lines. An advantage of MROM microcontrollers is that they come with user code embedded, which saves time and money since postproduction programming is not necessary. A disadvantage of MROM devices is that, since the mask with the user code has to be supplied early in the manufacturing process, throughput time (TPT) is longer.

Some versions of ROM (such as Intel's Quick-ROM) are actually not ROMs, but rather EPROMs, which are programmed at the factory. These devices are packaged in plastic devices, which prevents them from being erased since ultraviolet light cannot be applied to the actual EPROM array. Throughput time for QROMs is faster since the user code isn't required until after the actual manufacturing of the microcontroller is complete. As with

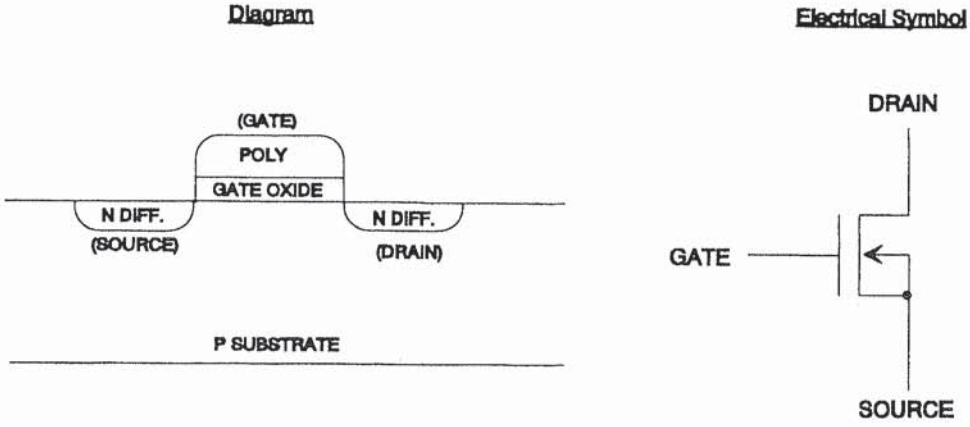


FIGURE 11.21 ROM memory cell.

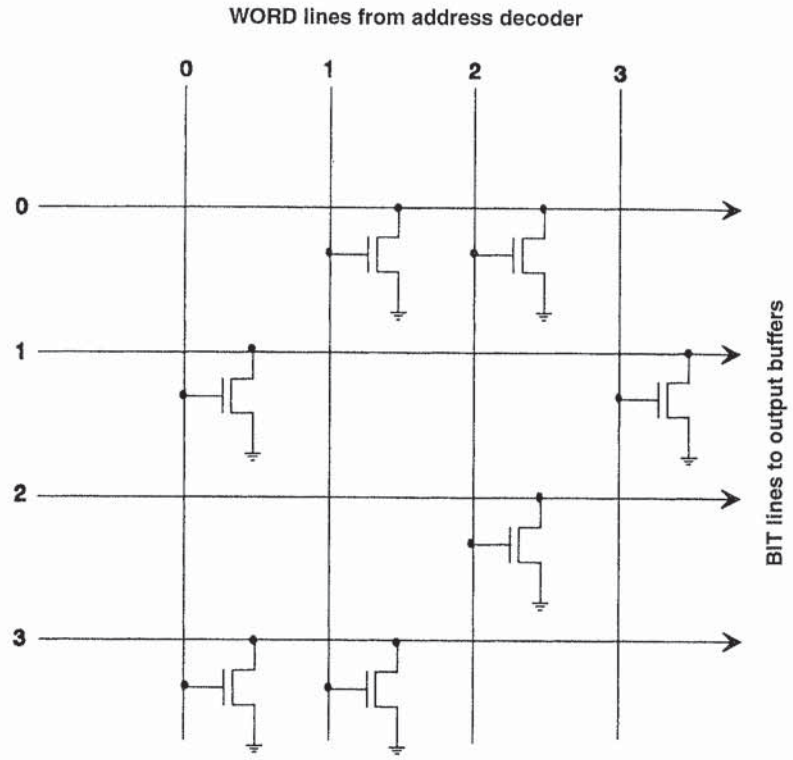


FIGURE 11.22 Simplified ROM memory matrix.

MROMs, the user supplies the ROM code to the microcontroller manufacturer. Instead of creating a mask with the ROM code, the manufacturer programs it into the device just prior to final test.

EPROM. EPROM devices are typically used during application development since this is when user code is changed often. EPROMs are delivered to the user unprogrammed. This allows the user to program the code into memory just prior to installation into an ECU module. Many EPROM microcontrollers actually provide a mechanism for in-module programming. This feature allows the user to program the device via the serial port while it is installed in the module. EPROM devices come assembled in packages either with or without a transparent window. Windowed devices are true EPROM devices that allow the user to erase the memory contents by exposing the EPROM array to ultraviolet light. These devices may be reprogrammed over and over again and thus are ideally suited for system development and debug during which code is changed often. EPROM devices assembled in a package without a window are commonly referred to as *one-time programmable devices* or OTPs. OTPs may only be programmed once, since the absence of a transparent window prevents UV erasure. OTPs are suited for limited production validation (PV) builds in which the code will not be erased.

A typical EPROM cell is shown in Fig. 11.23. It is basically an N-channel transistor that has an added poly1 floating gate to store charge. This floating gate is not connected and is surrounded by insulating oxide that prevents electron flow. The mechanism used to program an EPROM cell is known as *hot electron injection.* Hot electron injection occurs when very high drain (9-V) and select gate (12-V) voltages are applied. This gives the negatively charged electrons enough energy to surmount the oxide barrier and allows them to be stored on the gate.

This has the same effect as a negative applied gate voltage and turns the transistor off. When the cell is unprogrammed, it can be turned on like a normal transistor by applying 5 V to the poly2 select gate. When it is programmed, the 5 V will not turn on the cell. The state of the cell is determined by attempting to turn on the cell and detecting if it turns on. Erasure is performed through the application of ultraviolet (UV) light, which gives just the right amount of energy necessary for negatively charged electrons to surmount the oxide barrier and leave the floating gate.

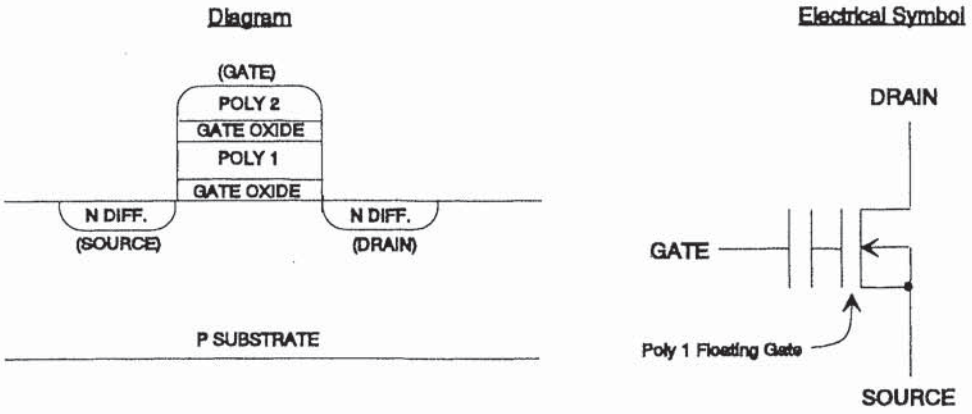


FIGURE 11.23 EPROM memory cell.

Flash. Flash memory is the newest nonvolatile memory technology and is very similar to EPROM. The key difference is that flash memory can be electrically erased. Once programmed, flash memory contents remain intact until an erase cycle is initiated via software. Like EEPROM, flash memory requires a programming and erase voltage of approximately 12.0 V. Since a clean, regulated 12-V reference is not readily available in automotive environments, this need is often provided for through the incorporation of an on-chip charge pump. The charge pump produces the voltage and current necessary for programming and erasure from the standard 5-V supply voltage. The advantage of flash is in its capability to be programmed *and* erased in-module without having to be removed. In-module reprogrammability is desirable since in-vehicle validation testing doesn't always allow for easy access to the microcontroller. Flash also allows for last-minute code changes, data table upgrades, and general code customization during ECU assembly. Since a flash cell is nearly identical in size to that of an EPROM cell, the high reliability and high device density capable with EPROM is retained. The main disadvantage of flash is the need for an on-chip charge pump and special program and erase circuitry, which adds cost.

A flash memory cell is essentially the same as an EPROM cell, with the exception of the floating gate. The difference is a thin oxide layer which allows the cell to be electrically erased. The mechanism used to erase data is known as *Fowler-Nordheim tunneling,* which allows the charge to be transferred from the floating gate when a large enough field is created. Hot electron injection is the mechanism used to program a cell, exactly as is done with EPROM cells. When the floating gate is positively charged, the cell will read a "1", when negatively charged, the cell will read a "0".

EEPROM. EEPROM (electrically erasable and programmable ROM, commonly referred to as E^2ROM) is a ROM that can be electrically erased and programmed. Once programmed, EEPROM contents remain intact until an erase cycle is initiated via software. Like flash, programming and erase voltages of approximately 12 V are required. Since a clean, regulated 12-V reference is not readily available in automotive environments, this requirement is satisfied using an on-chip charge pump as is done for flash memory arrays. Like flash, the advantage of EEPROM is its

capability to be programmed and erased in-module. This allows the user to erase and program the device in the module without having to remove it. EEPROM's most significant disadvantage is the need for an on-chip charge pump. Special program and erase circuitry also adds cost.

An EEPROM cell is essentially the same as an EPROM cell with the exception of the floating gate being isolated by a thin oxide layer. The main difference from flash is that Fowler-Nordheim electron tunneling is used for *both* programming and erasure. This mechanism allows charge to be transferred to or from the floating gate (depending upon the polarity of the field) when a large enough field is created. When the floating gate is positively charged, the cell will read a "1"; when negatively charged, the cell will read a "0".

11.2.2 Off-Chip Memory

Off-chip memory offers the most flexibility to the system designer, but at a price; it takes up additional PCB real estate as well as additional I/O pins. In cost- and size-conscience applications, such as automotive ABS, system designers almost exclusively use on-chip memory. However, when memory requirements grow to sizes in excess of what is offered on-chip (such as is common in electronic engine control), the system designer must implement an off-chip memory system. Off-chip memory is flexible because the user can implement various memory devices in the configuration of his choice. Most microcontrollers on the market today offer a wide variety of control pins and timing modes to allow the system designer flexibility when interfacing to a wide range of external memory systems.

Accessing External Memory. If circuit designers must use external memory in their applications, the type of external address/data bus incorporated onto the microcontroller should be considered. If external memory is not used, this will have, if any, impact upon the application. There are two basic types of interfaces used in external memory systems. Both of these are parallel interfaces in which bits of data are moved in a parallel fashion and are referred to as *multiplexed* and *demultiplexed* address/data buses.

Multiplexed Address/Data Buses. As the name implies, multiplexed address/data buses allow the address as well as the data to be passed over the same microcontroller pins by multiplexing the two in time. Figure 11.24 illustrates a typical multiplexed 16-bit address/data bus system as is implemented with Intel's 8XC196Kx family of microcontrollers.

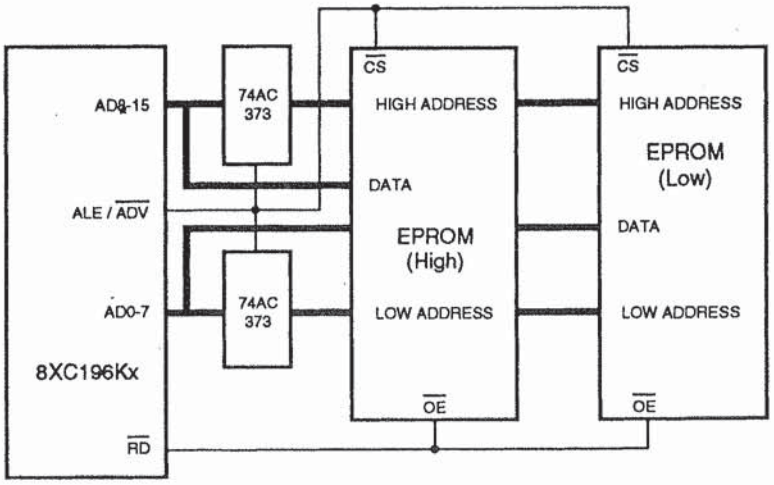


FIGURE 11.24 Multiplexed address/data bus system.

During a multiplexed bus cycle (refer to Fig. 11.25), the address is placed on the bus during the first half of the bus cycle and then latched by an external address data latch. The signal to latch the address comes from a signal generated by the microcontroller, called address latch enable (ALE). The address must be present on the bus for a specified amount of time prior to ALE being asserted. After the address is latched, the microcontroller asserts either a read (RD#) or a write (WR#) signal to the external memory device.

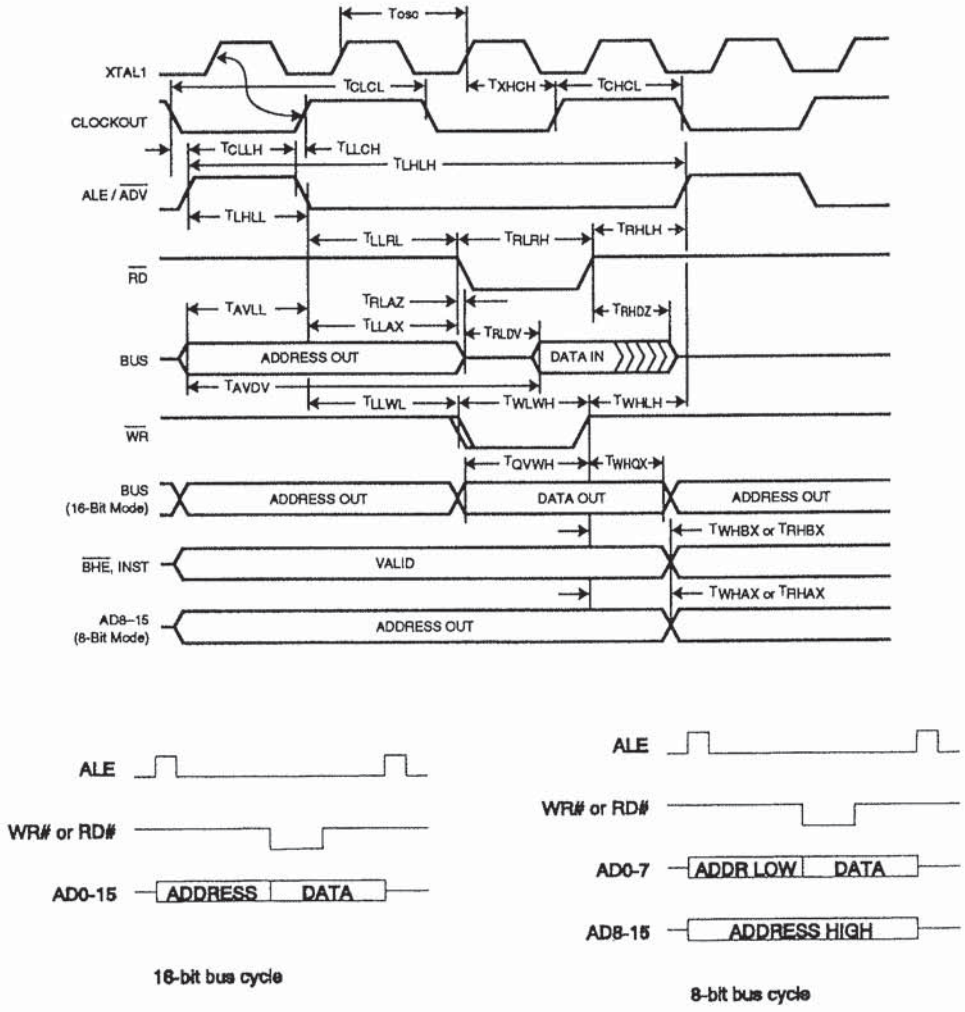


FIGURE 11.25 Multiplexed bus cycle and timing diagram.

For a read cycle, the microcontroller will pull its RD# output pin low and float the bus to allow the memory device to output the data located at the address latched on its address pins. The data returned from external memory must be on the bus and stable for a specified setup time before the rising edge or RD#, which is when the microcontroller latches the data.

For a write cycle, the microcontroller will pull its WR# pin low and then output data on the bus to be written to the external memory. After a specified setup time, the microcontroller will

release its WR# signal, which signals to the memory device to latch the data on the bus into the address location present on its address pins.

Advantages of multiplexed address/data bus systems are that fewer microcontroller pins are required since address and data share the same pins. For a true 16-bit system, this translates into a multiplexed system requiring 16 fewer pins (for address and data) than would be required by a demultiplexed system. A disadvantage is that an external latch is required to hold the address during the second half of the bus cycle; this adds to the component count.

Demultiplexed Address/Data Buses. Microcontrollers with demultiplexed address/data buses implement separate, dedicated address and data buses as shown in Fig. 11.26.

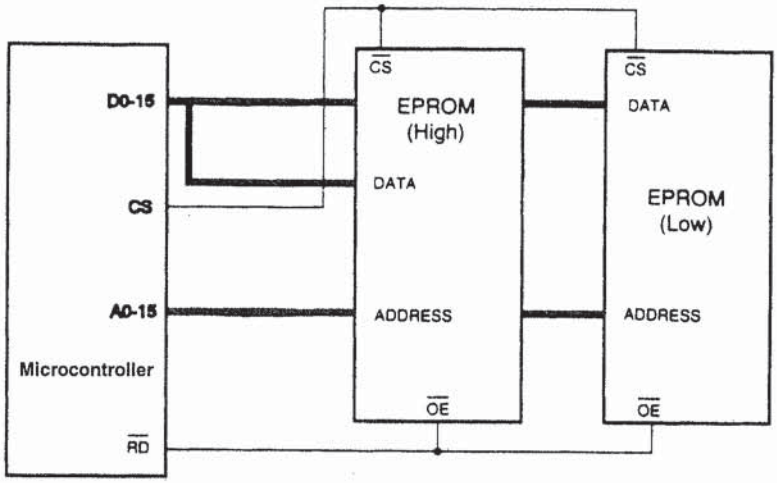


FIGURE 11.26 Typical demultiplexed address/data bus system.

The operation of a demultiplexed address/data bus is basically the same as the multiplexed type with the exception of not having an ALE signal to latch the address for the second half of the bus cycle. The operation of the RD#,WR#, address, and data lines is essentially the same as for that of a multiplexed system.

During a demultiplexed bus cycle, the microcontroller places the address on the address bus and holds it there for the entire bus cycle. For a read of external memory, the microcontroller asserts the RD# signal (or WR# for a write signal) just as would be done for a multiplexed bus cycle. The memory device will respond accordingly by either placing the data to be read on the data bus or by latching the data to be written off of the data bus. Figure 11.27 illustrates a simplified demultiplexed bus cycle.

An advantage of multiplexed address/data bus systems is that external data latches are not necessary, which saves on system component count. A disadvantage, as mentioned earlier, is that more microcontroller pins must be allocated for the interface, which leaves fewer pins for other I/O purposes.

11.3 LOW-SPEED INPUT/OUTPUT PORTS

Low-speed input/output (LSIO) ports allow the microcontroller to read input signals as well as provide output signals to and from other electronic components such as sensors, power drivers,

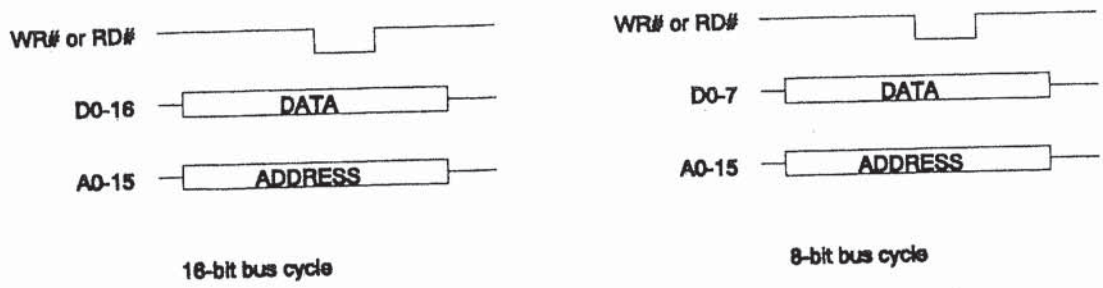


FIGURE 11.27 Demultiplexed bus cycle.

digital devices, actuators, and other microcontrollers. The term "low-speed" is used to describe these ports because unlike high-speed I/O (HSIO) ports which are interrupt driven, LSIO port data must be manually read and written by the user program. Interrupt-driven I/O is typically not possible on port pins configured for LSIO operation. It is common for modern high-performance microcontrollers to utilize multifunctional port pins which can be configured for a special function as well as LSIO. LSIO ports most commonly consist of eight port pins in parallel, which are supported by byte registers. For example, by writing to a single-byte special function register, an entire port can be configured, read, or written. Manipulating individual bits in the port register allows the user flexibility in accessing either single or multiple port pins.

11.3.1 Push-Pull Port Pin Configuration

The term *push-pull,* or *complementary,* output is commonly used to define a port pin that has the capability to output either a logic "1" or "0". Figure 11.28 shows a basic push-pull port pin configuration. Referring to Fig. 11.28, writing a "1" to the data output register enables the P-channel MOSFET and pulls the pin to +5 V, thus driving a logic "1" at the port pin. When a "0" is written

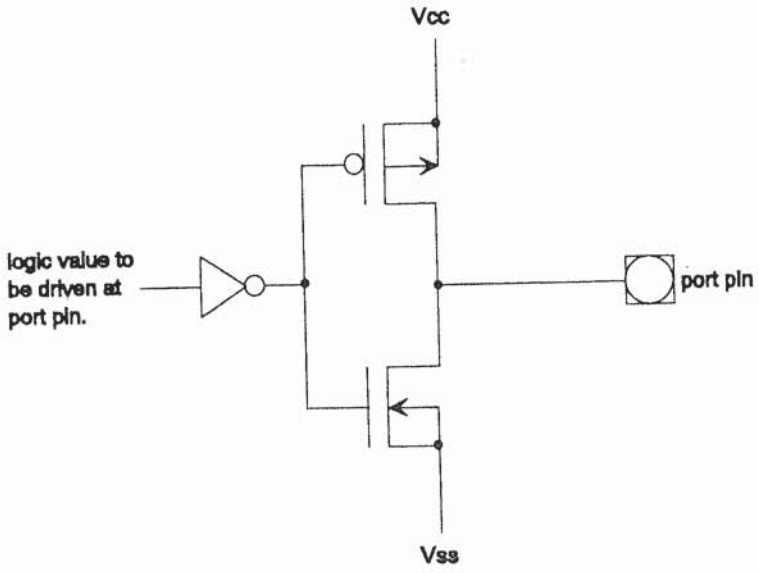


FIGURE 11.28 Push-pull port pin.

to the data register, the N-channel MOSFET is enabled and thus provides a current path to ground which results in a logic 0 at the port pin. Note that during this time the P-channel pull-up MOSFET is disabled to prevent contention at the port pin. Also note that the port logic design does not allow both the P-channel and the N-channel devices to be driving at the same time.

11.3.2 Open-Drain Port Pin Configuration

Open-drain port pins (Fig. 11.29) are useful for handshaking signals over which multiple devices will have control. The fact that the P-channel transistor is either omitted or disabled dictates the need for an external pull-up resistor. An example of an application for open-drain port pins would be for a bus contention line between two microcontrollers communicating on a common bus. During normal operation, the line is pulled high by the external pull-up resistor to signal to either microcontroller that no contention exists. If one of the microcontrollers should detect contention on the bus, it simply outputs a logic "0", which signals the contention to the other processor. To output the "0", the port only has to overcome the external pull-up which the user should appropriately size to match the port drive specifications.

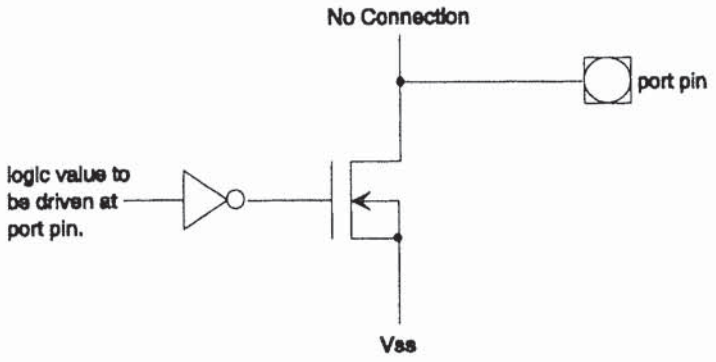


FIGURE 11.29 Open-drain port pin.

11.3.3 High-Impedance Input Port Pin Configuration

High impedance, or "Hi-z," port pins (Fig. 11.30) are used strictly as inputs since no drivers exist on these types of pins. Hi-z refers to the relatively high input impedance of the port pin. This high input impedance prevents the port pin circuitry from actively loading the input signal. Note that the pin is connected to the gates of a CMOS inverter, which drives internal circuitry. Usually a certain amount of hysteresis is built into these pins and is specified in the data sheet.

11.3.4 Quasi Bidirectional Port Pin Configuration

Quasi bidirectional (QBD) port pins are those that can be used as either input or output without the need for direction control logic. QBD port pins can output a strong low value or a weak high value. The weak high value can be externally overridden, providing an input function. Figure 11.31 shows a QBD port pin diagram and its transfer characteristic.

Writing a "1" to the port pin disables the strong low driver (Q2) and enables a very weak high driver (Q3). To get the pin to transition high quickly, a strong high driver (Q1) is enabled for one state time and then disabled (leaving only Q3 active).

It is important to keep in mind that since the port pin can be externally overridden with a logic "0", reading the port pin could falsely indicate that it was written as a logic "0".

The ability to overdrive the weak output driver is what gives the quasi bidirectional port pin its input capability. To reduce the amount of current that flows when the pin is externally pulled low, the weak output driver (Q4) is turned off when a valid logic "0" is detected. The input transfer characteristic of a quasi bidirectional port pin is shown in Fig. 11.31.

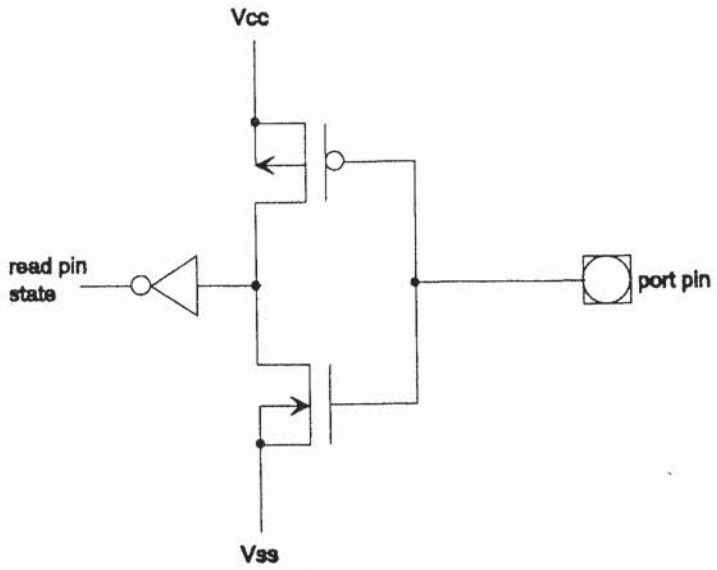


FIGURE 11.30 High-impedance input port pin.

11.3.5 Bidirectional Port Example

The following example describes the operation of a state-of-the-art bidirectional port structure. This particular structure is used upon newer members of Intel's MCS-96 automotive microcontroller family. A single port consists of eight multifunction, parallel port pins (see Fig. 11.32), which are controlled (on a by-pin basis) with four special function registers referred to as Px_PIN, Px_REG, Px_MODE, and Px_DIR. As is common with other high-performance microcontrollers, the pins of this port are shared with alternate special functions controlled by other on-chip peripherals. The Px_MODE register allows the programmer to choose either LSIO or the associated special function for any given port pin. Writing a "1" to the appropriate bit selects the corresponding pin as special function whereas a "0" selects LSIO. The function of the Px_PIN and Px_REG registers is fairly straightforward. In order to read the value on the pin, the user simply reads the Px_PIN register. To write a value to the Px_REG register, the user simply writes the desired output value to the Px_REG register. The Px_DIR register allows the user to configure the port pin as either input or output.

In order to prevent an undefined pin state during reset, port pins revert to a default state during reset. For the Intel Kx bidirectional port structure, this state is defined as a weak logic "1". The transistor that drives this state is labeled as WKPU in Fig. 11.32 and is asserted in reset until the user writes to the Px_MODE register to configure the port pin.

Ports such as this offer the user much flexibility in assigning their function within an application. Following are three examples that depict how these ports may be configured by the user by writing values to the appropriate bit within the port SFR. Also note that the eight pins of a port may be configured individually on a pin-by-pin basis.

To configure a given port pin as a high-impedance input pin, the user must write the fol-

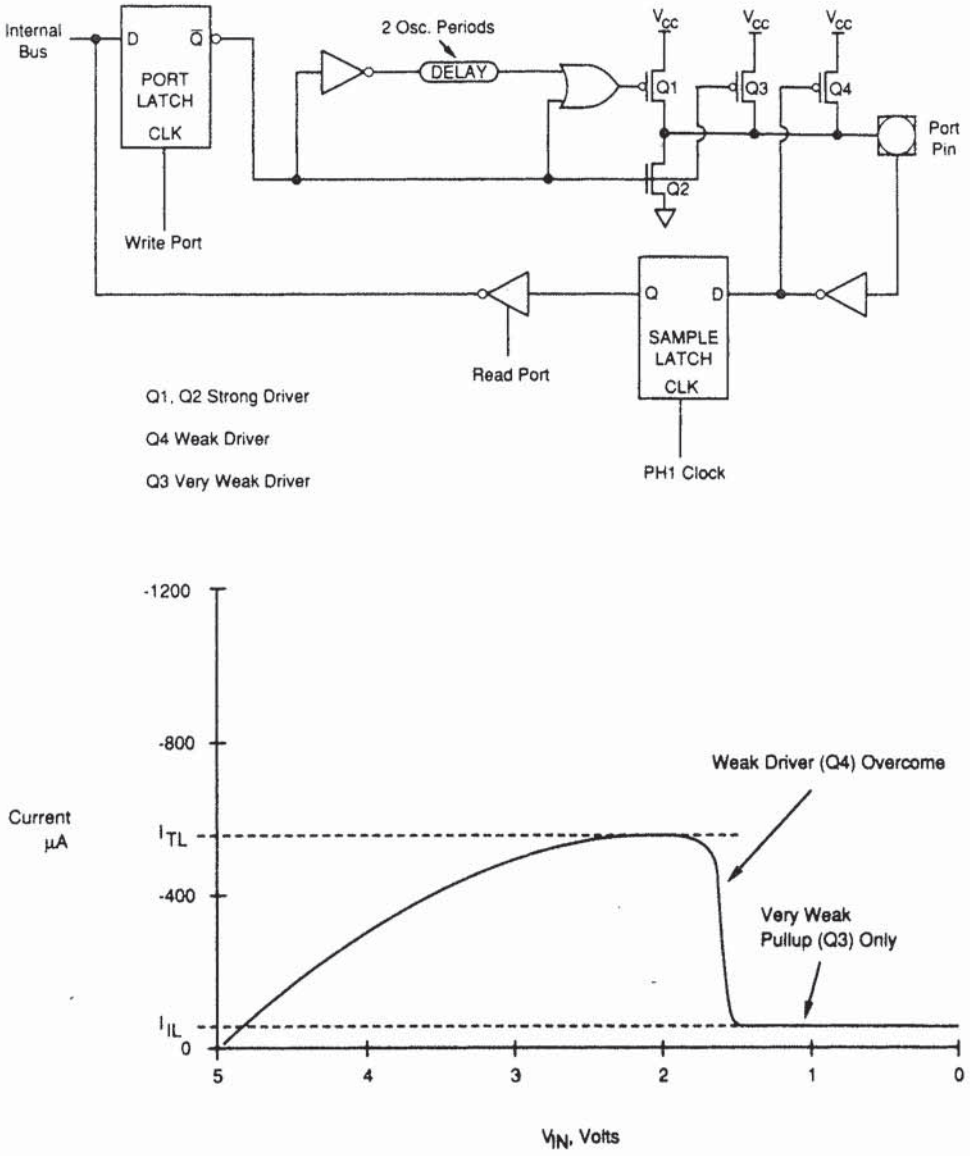


FIGURE 11.31 Quasi bidirectional port pin and transfer characteristic.

Px_MODE: "0" selects the pin as LSIO and disables weak pull-up.
Px_DIR: "1" disables operation of the N-channel transistor.
Px_REG: "1" disables the N-channel transistor.

To configure a given port pin for push-pull operation, the following values must be written to the corresponding bit within the port SFR.

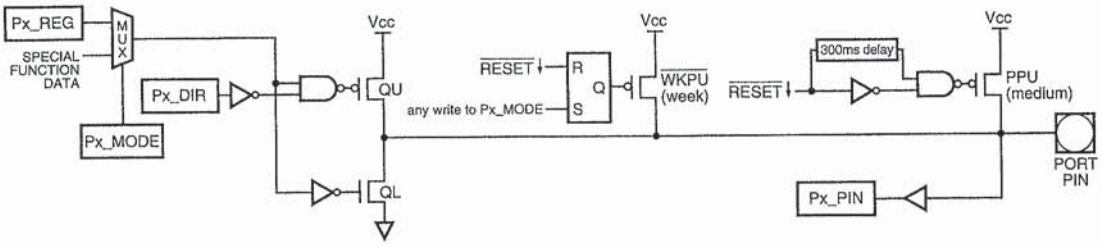


FIGURE 11.32 Bidirectional port structure example.

Px_MODE: "0" selects the pin as LSIO and disables weak pull-up transistor.
Px_DIR: "0" enables operation of both the N- and P-channel transistors.
Px_REG: "0" or "1" drives that value at the port pin.

To configure a port pin for open-drain operation, the user must write the following values to the corresponding bits within the port SFR.

Px_MODE: "0" selects the pin as LSIO and disables weak pull-up transistor.
Px_DIR: "1" disables operation of the N-channel transistor.
Px_REG: "1" disables the P-channel transistor / achieves Hi-Z state.
"0" enables the N-channel transistor / drives "0" at pin.

11.4 HIGH-SPEED I/O PORTS

Perhaps the most demanding of automotive microcontroller applications is electronic engine control and antilock braking/traction control. These applications both require the microcontroller to detect, process, and respond to external signals or "events" within relatively short periods of time. Sometimes referred to as a capture/compare module, a microcontroller's HSIO (high-speed input/output) peripheral allows the microcontroller to capture an event as it occurs. The term *capture* refers to a series of events that begins with the microcontroller detecting a rising or falling edge upon a high-speed input pin. At the precise moment this edge is detected, the value of a software timer is loaded into a time register and an interrupt is triggered. This gives the microcontroller the relative time at which the event occurred. An HSIO peripheral also provides compare functions by detecting an internal event, such as a timer reaching a particular count value. When the particular count value is detected, the HSIO unit will generate a specified event (rising or falling edge) on a port pin. This feature is ideal for generating PWM waveforms or synchronizing external events with internal events.

For example, consider a typical ABS microcontroller which must detect, capture, and calculate wheel speeds; respond with signals to hydraulic solenoids; and perform many other background tasks all within a loop time of about 5 ms. The wheel speed signals are input to the microcontroller as square waves with frequencies up to 7000 Hz (approximately one edge every 71 μs). The microcontroller must have the performance necessary to capture and process these edges on as many as four wheel speed inputs. HSIO peripherals, along with the interrupt structure, play a major role in the microcontroller's ability to perform this function.

Nearly every microcontroller manufacturer has its own proprietary HSIO peripheral. For purposes of this section, the event processor array (EPA) HSIO peripheral, which is used by Intel's 87C196KT automotive microcontroller, will be discussed.

11.4.1 High-Speed Input and Output Peripheral

High-speed input/output peripherals typically consist of a given number of capture/compare modules, a timer/counter structure, control and status SFRs, and an interrupt structure of some type. Figure 11.33 shows a block diagram of the EPA peripheral. The main components of the EPA are ten capture/compare channels, two compare only channels, and two timer/counters. The capture/compare channels are configured independently of each other. The two timer/counters are shared between the various capture/compare channels. Each capture/compare channel has its own dedicated SFR's: EPAx_TIME and EPAx_CON (x designates the channel number).

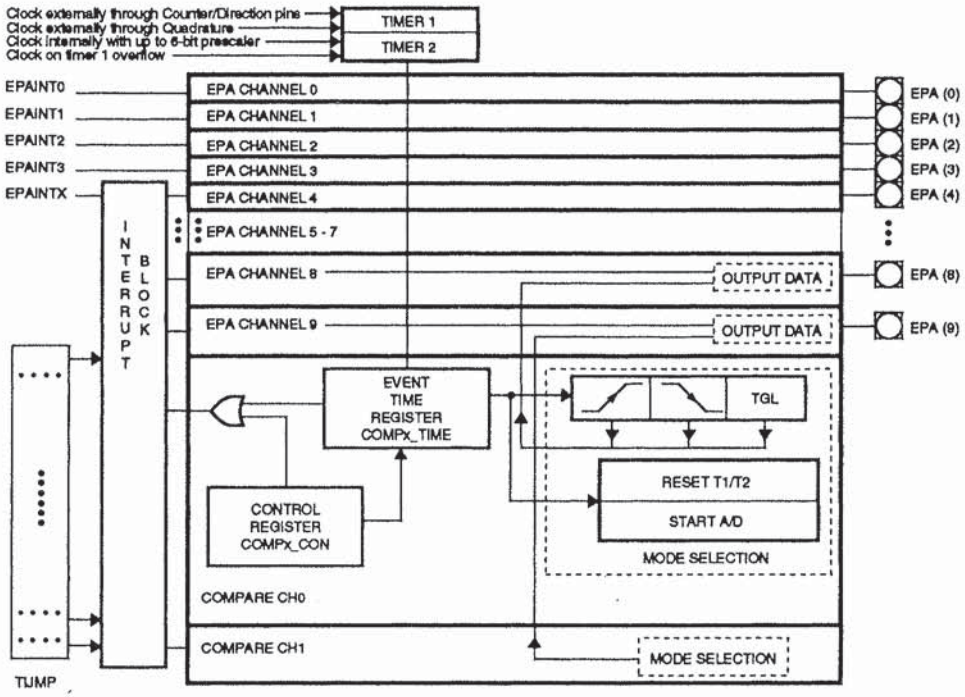


FIGURE 11.33 Example HSIO peripheral: Intel's EPA peripheral.

11.4.2 Timer/Counter Structures

High-performance microcontrollers typically integrate one or more timer/counters onto their silicon. A microcontroller's timer/counter structure provides a time base to which all HSIO events are referenced. Timers are clocked internally, whereas counters are clocked from an external clock source. Timers are often very flexible structures, in which programmers have the capability to configure the timer/counters to meet their application's particular needs. The 87C196KT has two 16-bit timer/counters referred to as TIMER1 and TIMER2. As 16-bit timer/counters, each timer has the capability of counting to 2^{16} or 65,536 before overflowing. The user has the option of triggering an interrupt upon overflow of a timer/counter. Each of these two timers can be independently configured using the TxCONTROL SFR as shown in Fig. 11.34, where x specifies either 1 or 2 for Timer1 or Timer2, respectively.

Bits number 3, 4, and 5 are the mode bits that allow the user to configure the clocking source and direction of each timer/counter. The clock rate can be based either upon the fre-

TxCONTROL SFR

7	6	5	4	3	2	1	0
CE	UD	M2	M1	M0	P2	P1	P0

CE: Count Enable: "0" = disable timer, "1" = enable timer

UD: Up/Down: "0" = count up, "1" = count down

MODE:	M2, M1, M0	Clock source	Direction determined by:
	0 0 0	XTAL/4	state of UD bit
	0 0 1	TxCLK pin	state of UD bit
	0 1 0	XTAL/4	state of TxDIR pin
	0 1 1	TxCLK pin	state of TxDIR pin
	1 0 0	Timer1 overflow	state of UD bit
	1 1 0	Timer1 overflow	same as Timer1
	1 1 1	Quadrature clocking using TxCLK and TxDIR pins	

Prescale:	P2, P1, P0	Clock prescale values
	0 0 0	÷ by 1 (250 ns @ 16 MHz xtal frequency)
	0 0 1	÷ by 2 (500 ns @ 16 MHz xtal frequency)
	0 1 0	÷ by 4 (1 μs @ 16 MHz xtal frequency)
	0 1 1	÷ by 8 (2 μs @ 16 MHz xtal frequency)
	1 0 0	÷ by 16 (4 μs @ 16 MHz xtal frequency)
	1 0 1	÷ by 32 (8 μs @ 16 MHz xtal frequency)
	1 1 0	÷ by 64 (16 μs @ 16 MHz xtal frequency)
	1 1 1	reserved

FIGURE 11.34 Timer control SFR example.

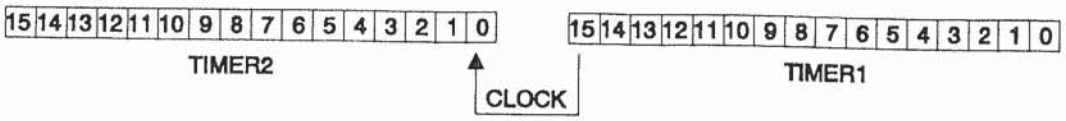


FIGURE 11.35 Cascading of timer/counters.

quency that the microcontroller is being clocked at the XTAL pins or upon the input frequency on another pin referred to as TxCLK. The user also has the option of either having the logic level of another pin (TxDIR) or the UD bits in TxCONTROL determine the direction (up/down) that the timer/counter is clocked.

For those applications that require a 32-bit timer/counter, the user has the option (using the mode bits) to direct the overflow of TIMER1 to clock TIMER2. This is known as cascading and essentially creates a 32-bit timer/counter as shown in Fig. 11.35.

11.4.3 Input Capture

Input capture refers to the process of capturing a current timer value when a specific type of event occurs. An excellent example of high-speed input capture can be illustrated with a basic automotive ABS input capture algorithm that calculates the frequency of a wheel speed input. The signals from the wheel speed sensors are input into the microcontroller's EPA pins as square waves. Consider the generic wheel speed input capture example shown in Fig. 11.36.

Two timers (1 and 2) are used in this example. Timer1 is used in conjunction with an EPA channel to provide a 5-ms software timer (this is a compare function that will be discussed in the next section). The 5 ms is the main loop time used in generic ABS algorithms. Timer2 is used in conjunction with one or more EPA channels to capture the relative times at which edges occur on wheel speed inputs. The EPA is configured to capture falling edges and initiate an interrupt, which stores the event time and increments an edge count. To simplify this example, we will consider only a single input channel.

The process starts by EPA interrupts being enabled after Timer1 starts a new 5-ms timer count. The first falling edge causes an interrupt that stores the event time (T2) into a variable *initial time* and increments an edge count. The next edge causes an interrupt in which the event time (T2+x) is stored into a variable called *final time* and increments the edge count.

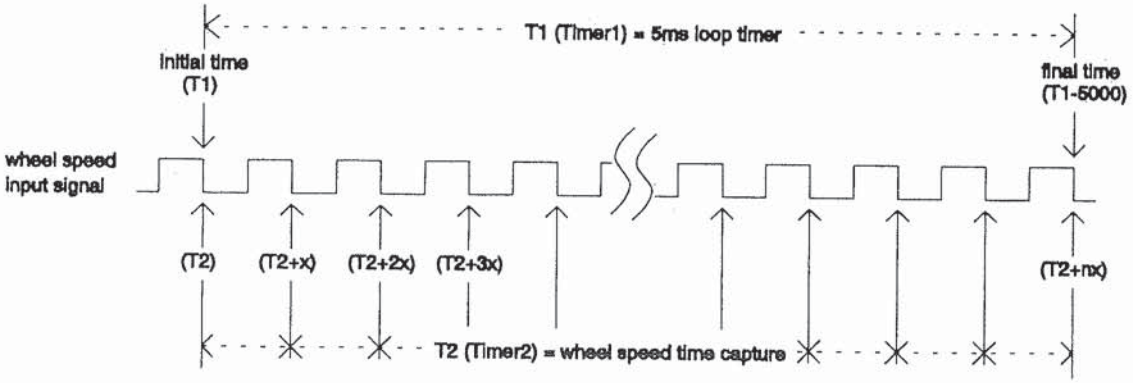


FIGURE 11.36 Input capture example using EPA peripheral.

Subsequent edges' event times are also stored into *final time* until Timer1's 5-ms count expires. At this point, final time contains the time at which the last edge to occur was captured. The average period of the input waveform can then be calculated with the following equation:

$$\text{input period} = (\textit{final time} - \textit{initial time}) / \text{edge count}$$

11.4.4 Output Compare

Output compare refers to the process of generating an event when a timer value matches a predetermined time value. The event may be to generate an interrupt, toggle an output pin, perform an A/D conversion, and so forth. Following is an example that shows the steps necessary to generate an event every 50 μs:

1. Enable the output compare channel's interrupt.
2. Initialize the timer to count up at 1 μs per timer tick.
3. Initialize the output compare channel to re-enable and reset the timer (to zero) when a timer match occurs.
4. Initialize the output compare channel to produce the desired event when a timer match occurs.
5. Write 32h (50 decimal) to the appropriate output compare channel's time register.
6. Enable the timer to start the process.
7. A compare channel interrupt will be generated every 50 μs.

Since the example re-enables and zeros the timer, the event will occur continuously until the user's program halts the process.

Software Timers. Software timers such as the 5-ms timer used in the ABS wheel speed capture example can be set up easily using a compare channel and a timer. The following software timer procedure is very similar to that used in the previous output compare example:

1. Enable the compare channel's interrupt.
2. Initialize the timer to count up at 1 us per timer tick.
3. Initialize the output compare channel to re-enable and reset the timer (to zero) when a timer match occurs.
4. Initialize the output compare channel to produce an interrupt (5-ms ISR) when a timer match occurs.
5. Write 1388h (5000 decimal) to the appropriate output compare channel's time register.
6. Enable the timer to start the process.
7. An compare channel interrupt will be generated every 5 ms.

11.4.5 Pulse-Width Modulation (PWM)

Pulse-width modulation (PWM) peripherals provide the user with the ability to generate waveforms that have specified frequencies and duty cycles. PWM waveforms are typically used to generate pulsed waveforms used for motor control or they may be filtered to produce a smooth analog signal. HSIO peripherals typically provide for PWM waveform generation, although the methods are not usually as efficient as dedicated PWM peripherals. A basic example of creating a PWM waveform using an HSIO peripheral's output compare function is described in Sec. 11.4.4.

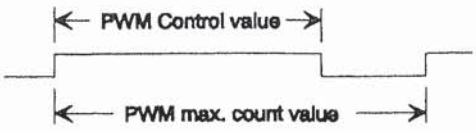


FIGURE 11.37 PWM waveform time values.

PWM Peripheral. The components of a basic automotive microcontroller's PWM peripheral include a counter (typically 8-bit), a comparator, a holding register, and a control register. The counter typically has a prescaler that allows the user to select the clock rate of the counter, which allows for selectable PWM frequencies. Without prescaling capability, an 8-bit counter would only allow for a period of 256 state times. The PWM control register determines how long the PWM output is held high during the pulse, effectively controlling the duty cycle as shown in Fig. 11.37. For an 8-bit PWM counter, the value written to the PWM control register can be from 0 to 255 (equating to 255 state times with no prescaling). Note that PWM peripherals do not typically allow for a 100 percent duty cycle because the output must be reset when the counter reaches zero.

The operation of a PWM peripheral is rather simple. The PWM control register's value (assume 8-bit for this example) is loaded into a holding register when the 8-bit counter overflows. The comparator compares the contents of the holding register to the counter value. When the counter value is equal to zero, the PWM output is driven high. It remains high until the counter value matches the value in the holding register, at which time the output is pulled low. When the counter overflows, the output is again switched high. Figure 11.38 shows typical PWM output waveforms.

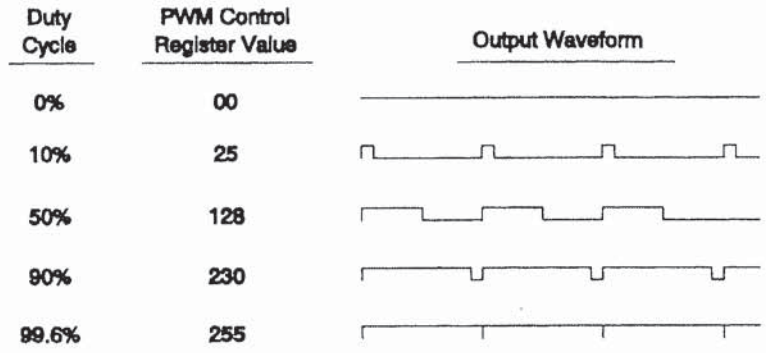


FIGURE 11.38 PWM output waveforms.

11.5 SERIAL COMMUNICATIONS

It is often necessary for automotive microcontrollers to have the capability to communicate with other devices both internal and external to the ECU. Within an ECU a microcontroller may have to communicate with other devices such as backup processors, shift registers, watchdog timers, and so forth. It is not uncommon for automotive microcontrollers to communicate with devices external to the ECU, such as other modules within the vehicle and even diagnostic computers at a service station. All of these communication examples require a large quantity of data to be transmitted/received in a short period of time. Also consider that this communication must utilize as few pins of the microcontroller as possible in order to save valuable PCB board space. These requirements all support the need for serial communications.

Serial communications provides for efficient transfer of data while utilizing a minimum number of pins. Serial communications is performed by transferring a group of data bits, one at a time, sequentially over a single data line. Each transmission of a group of bits (typically a

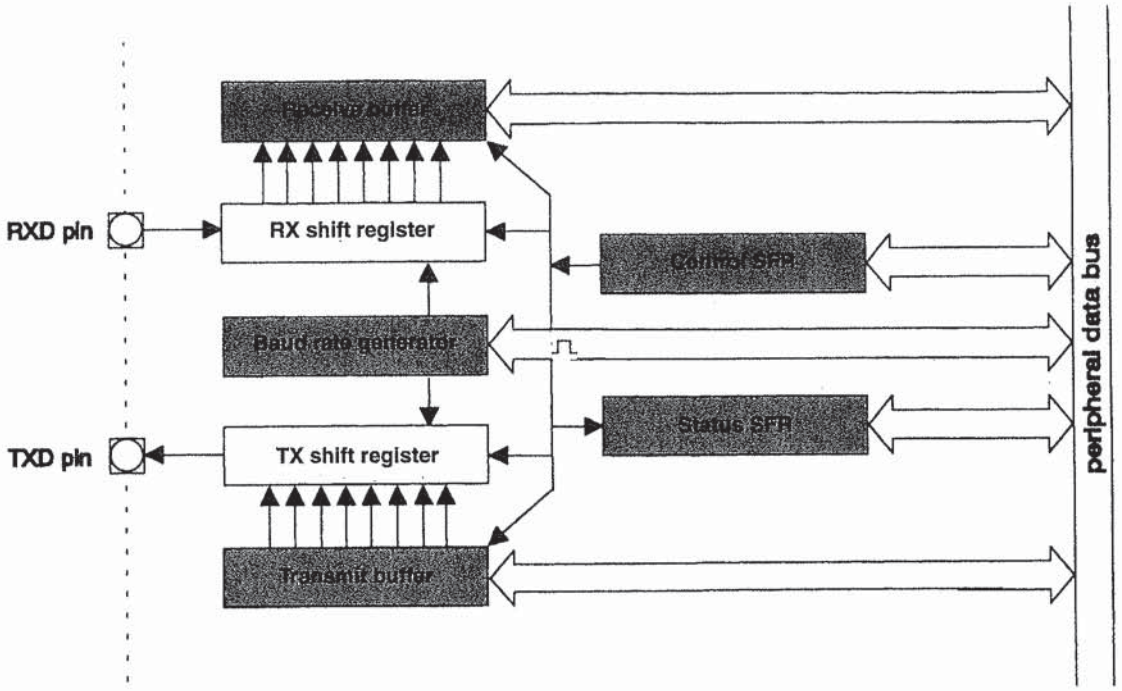


FIGURE 11.39 Serial port block diagram.

byte of data) is known as a data frame. This transfer of data takes place at a given speed, which is referred to as the baud rate and is typically specified in bits/second.

A typical microcontroller serial port consists of data buffers, data registers, and a baud rate generator. Interface to the outside world takes place via the transmit (TXD) and receive (RXD) pins. A block diagram for a typical serial port peripheral is shown in Fig. 11.39. By writing to the serial port control register, users are able to customize the operation of the serial port to their particular application's requirements.

The baud rate generator is used to provide the timing necessary for serial communications and determines the rate at which the bits are transmitted. In synchronous modes, the baud rate generator provides the timing reference used to create clock edges on the clock output pin. In asynchronous modes, the baud rate generator provides the timing reference used to latch data into the RX pin and clock it out of the TX pin.

11.5.1 Synchronous Serial Communications

Sometimes an application does not allow asynchronous serial communications to take place due to variations in clock frequency, which results in unacceptable baud rate error. Some applications simply require some sort of shift register I/O. Synchronous communication involves an additional clock pin, which is used to signal the other device that data being transferred are valid and ready to be read. Often when the user configures the serial port to work in a synchronous mode, the TXD pin automatically reverts to supplying the clock and the RXD pin automatically becomes the data pin. This configuration prevents an additional pin from having to be reserved for use as a serial clock pin. When a synchronous data transfer is initiated, a series of eight clock pulses is emitted from the clock pin at a predetermined baud rate as shown in Fig. 11.40.

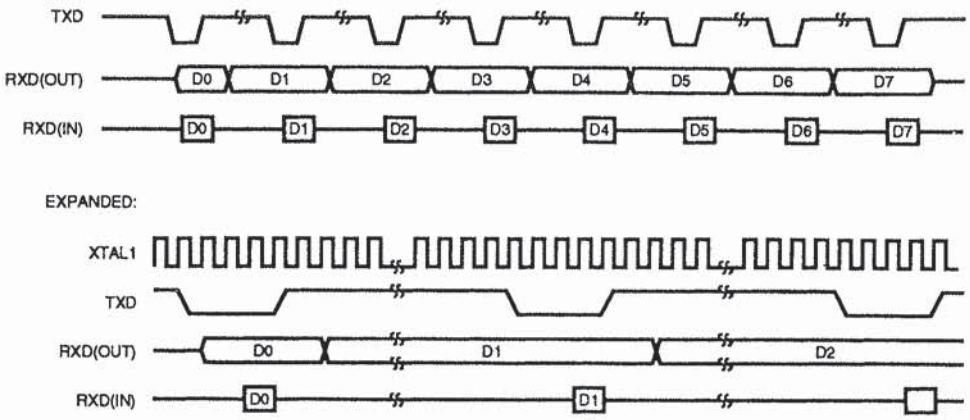


FIGURE 11.40 Synchronous serial mode data frame.

An example of synchronous serial communications is shown in Fig. 11.41. Assume that processor A is to transfer a byte of data to processor B. The program executing in processor A initiates a serial transmission by writing the data byte to be transmitted into the transmit buffer. Assuming microcontroller A's serial port is enabled for transmission, writing to the transmit buffer results in a series of eight clock pulses to be emitted from microcontroller A's clock pin. The first falling edge of the clock will signal to processor B that bit 0 (LSB) is ready to be read into its receive buffer. Microcontroller A will place the next data bit on the TXD pin with each rising clock edge. With B's serial port enabled for reception, each falling edge will result in another data bit being shifted into B's receive buffer. When B's receive buffer is full, the received data byte will be loaded into its receive register and will signal its CPU that the reception has been completed and the data is ready for use.

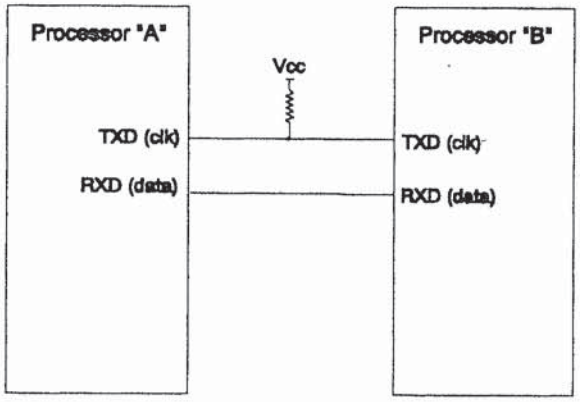


FIGURE 11.41 Synchronous serial communications example.

Shift Register Based I/O Expansion. A common application for synchronous serial transmission is shift register based I/O expansion as shown in Fig. 11.42. In this circuit, a 74HC164 8-bit serial-in/parallel-out shift register is used to provide eight parallel outputs with a single serial input. The 74HC165 8-bit parallel-in/serial out shift register shown provides a single serial input resulting from eight parallel input signals. This allows the system designer to

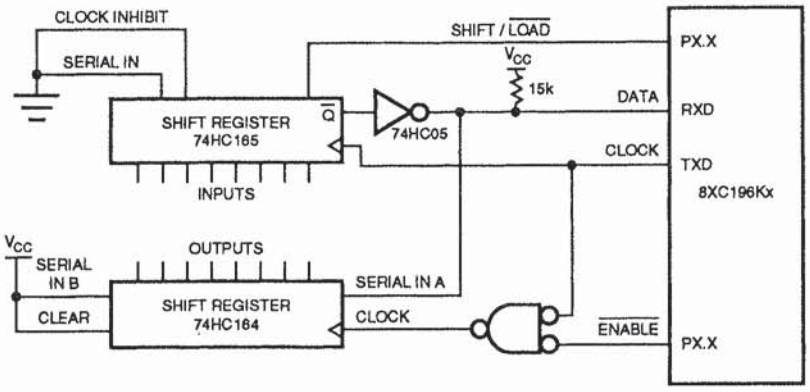


FIGURE 11.42 Shift register based I/O expansion example.

implement an additional 8-bit output port and additional 8-bit input port (16 signals total) using only four pins on the microcontroller. This expansion scheme allows a designer to achieve a greater number of I/O pins without having to upgrade to a microcontroller with a higher pin count.

To output data using this I/O expansion method, the user code simply writes a byte to the serial port transmit register to initiate data transfer. This causes the written byte to be shifted out of the microcontroller's RXD pin and into the 74HC164 one bit at a time. The data is reflected at the output pins of the 74HC164 as each bit is shifted in. For address/data bus emulation, another microcontroller pin may be utilized to indicate valid data to the intended receiving device.

To receive eight bits of data in parallel using this method, the user's code must latch the data on the 74HC165's input pins into its shift register by asserting the *shift/load* signal. After this is accomplished, the user's code simply needs to enable the serial port receive circuitry to receive the data one bit at a time into its receive buffer.

11.5.2 Asynchronous Serial Communications

The most common type of serial communications is asynchronous. As its name implies, asynchronous communication takes place between two devices without use of a clock line. Data is transmitted out the transmit buffer and received into the receive buffer independently at a speed determined by the baud rate generator. Most microcontrollers offer several modes of asynchronous serial communication.

Standard Asynchronous Mode. The standard asynchronous mode consists of 10 bits: a start bit, eight data bits (LSB first), and a stop bit, as shown in Fig. 11.43. After the user initiates a transmission, data is automatically transmitted from the TX pin at the specified baud rate.

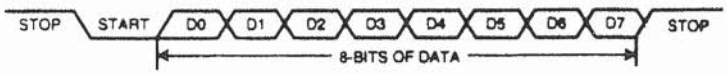


FIGURE 11.43 Standard asynchronous mode data frame.

A parity function is also implemented, which provides for a simple method of error-detection. Data transmitted will consist of either an odd or even number of logical "1"s. If even parity is enabled, the parity bit will either be set to a "1" or a "0" to make the number of "1"s in the data byte even. If odd parity is enabled, the parity bit will be set to the appropriate value to make the number of "1"s in the data byte odd. For instance, consider the data byte 11010010b. If even parity is enabled, the parity bit will be set to a "0" since there is already an even number of "1"s. If odd parity were enabled, the parity bit would be set to a "1" since another "1" would be needed to provide an odd number of "1"s. If the parity function is enabled (usually through a serial port control register), the parity bit is sent instead of the eighth data bit and parity is checked on reception. The occurrence of parity errors is typically flagged in a serial port status register to alert the microcontroller to corrupted data in the receive register.

Multiprocessor Asynchronous Serial Communications Modes. Two other common serial communications modes which are used on automotive microcontrollers are the asynchronous 9th-bit recognition mode and the asynchronous 9th-bit mode. These two modes are commonly used together for multiprocessor communications where selective selection on a data link is required. Both modes are similar to the standard asynchronous mode with the exception of an additional ninth data bit in the data frame as shown in Fig. 11.44.

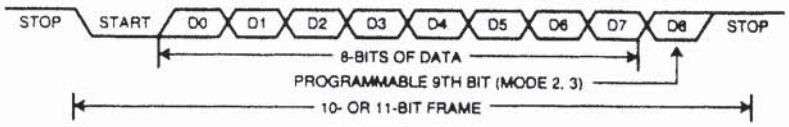


FIGURE 11.44 Asynchronous 9th-bit data frame.

The 9th-bit recognition mode consists of a start bit, nine data bits (LSB first), and a stop bit. For transmission, the ninth bit can be set to "1" by setting a corresponding bit in the serial port control register before writing to the transmit buffer. During reception, the receive interrupt bit is *not* set unless the ninth data bit being received is set to a logic "1".

The 9th-bit mode uses a data frame identical to that of the 9th-bit recognition mode. In this mode, a reception will always cause a receive interrupt, regardless of the state of the ninth data bit.

A multiprocessor data link is fairly simple to implement using these two modes. Microcontrollers within the system are connected as shown in Fig. 11.45. The master microcontroller is set to the 9th-bit recognition mode so that it is always interrupted by serial receptions. The slave microcontrollers are set to operate in the 9th-bit recognition mode so that they are interrupted on receptions only if the ninth data bit is set. Two types of data frames are used: address frames, which have the ninth bit set, and data frames, which have the ninth bit cleared. When the master processor wants to transmit a block of data to one of several slaves, it first sends out an address frame which identifies the target slave. Slaves in the 9th-bit recognition mode are not interrupted by a data frame, but an address frame interrupts all slaves. Each slave can examine the received byte and see if it is being addressed. The addressed slave then switches to the 9th-bit mode to receive data frames, while the slaves that were not addressed stay in the 9th-bit recognition mode and continue without interruption.

11.6 ANALOG-TO-DIGITAL CONVERTER

Analog-to-digital converter (A/D) peripherals allow automotive microcontrollers to sense and assign digital values to analog input voltages with considerable accuracy. An analog input

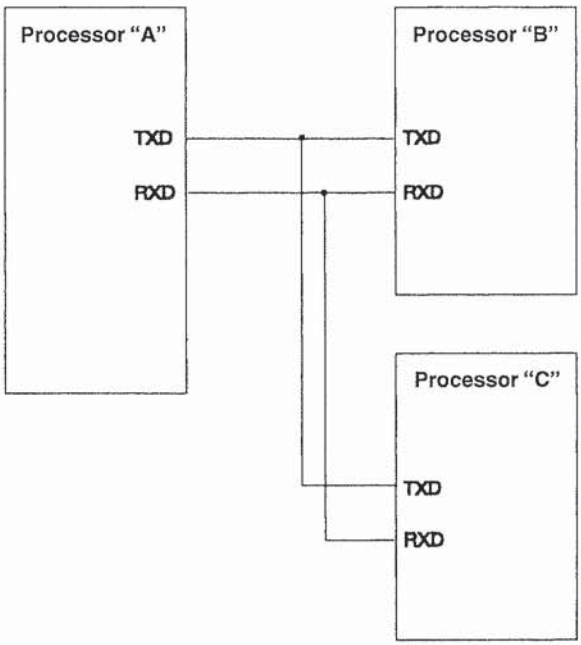


FIGURE 11.45 Asynchronous 9th-bit data frame.

may be defined as a voltage level that varies over a continuous range of values as opposed to the discrete values of digital signals.

11.6.1 Types of A/D Converters

The vast majority of A/D converters available on microcontrollers are of the successive approximation (S/A) type. Other types include flash A/D converters, in which conversions are completed in a parallel fashion and are performed at speeds measuring tens-of-nanoseconds. The drawback is that flash A/D converters require a great deal of die space when integrated on a microcontroller. It is because of their relatively large size that flash A/D converters are seldom offered on microcontrollers. Dual-slope A/D converters offer excellent A/D accuracy but typically take a relatively long period of time to complete a conversion. S/A A/D converters are very popular because they offer a compromise among accuracy, speed, and die-size requirements. The main drawback to successive approximation converters is that implementing the capacitor and resistor ladders takes a considerable amount of die space, although somewhat less than flash A/Ds. These converters are also somewhat susceptible to noise, although there are proven ways to reduce the effects of noise within a given application. The advantage of S/A converters is that they combine the best of other types of converters. They are relatively fast and do not take up excessive die space.

S/A converters typically consist of a resistor ladder, a sample capacitor, an input multiplexer, and a voltage comparator. A typical S/A converter is shown in Fig. 11.46. The resistor ladder is used to produce reference voltages for the input voltage comparison. A sample capacitor is utilized to capture the input voltage during a given period of time known as the sample time. Sample time can be defined as the amount of time that an A/D input voltage is applied to the sample capacitor.

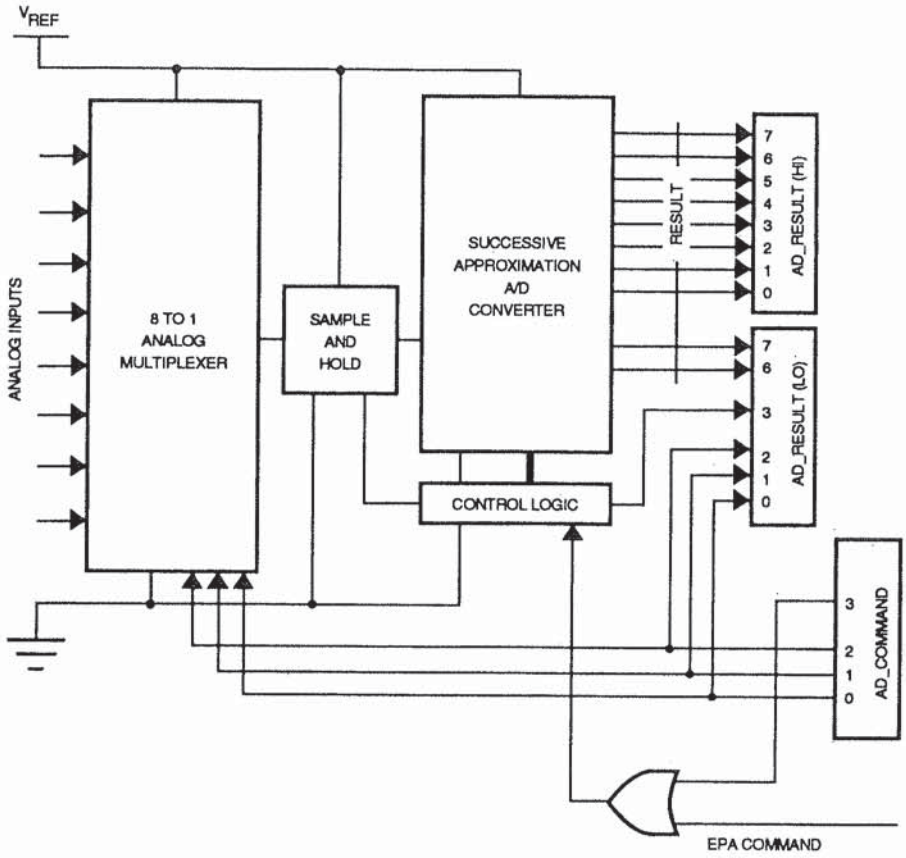


FIGURE 11.46 Typical successive approximation converter.

A successive approximation algorithm is used to perform the A/D conversion. A typical S/A converter consists of a 256-resistor ladder, a comparator, coupling capacitors, and a 10-bit successive approximation register (SAR), along with SFRs and logic to control the process. The resistor ladder provides 20-mV steps (with $V_{ref} = 5.12$ V), while capacitive coupling creates 5-mV steps within the 20-mV ladder voltages. Therefore, 1024 internal reference voltage levels are available for comparison against the analog input to generate a 10-bit conversion result. Eight-bit conversions use only the resistor ladder, providing 256 levels.

11.6.2 The A/D Conversion Process

The successive approximation conversion compares a reference voltage to the analog input voltage stored in the sampling capacitor. A binary search is performed for the reference voltage that most closely matches the input. The ½ full-scale reference voltage is the first tested. This corresponds to a 10-bit result in which the most significant bit is zero and all other bits are one (0111 1111 11b). If the analog input is less than the test voltage, bit 10 is left at zero and a new test voltage of ¼ full scale (0011 1111 11b) is tested. If this test voltage is less than the analog input voltage, bit 9 of the SAR is set and bit 8 is cleared for the next test (0101 1111

11b). This binary search continues until 8 or 10 tests have occurred, at which time the valid 8-bit or 10-bit result resides in the SAR where it can be read by software.

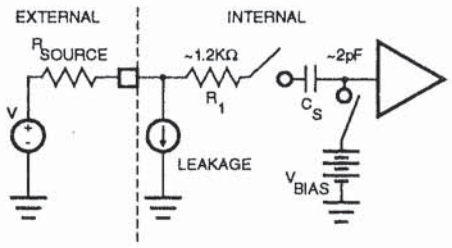


FIGURE 11.47 Idealized interface circuitry.

11.6.3 A/D Interfacing

The external interface circuitry to an analog input is highly dependent upon the application and can impact converter characteristics. Several important factors must be considered in the external interface design: input pin leakage, sample capacitor size, and multiplexer series resistance from the input pin to the sample capacitor. These factors are idealized in Fig. 11.47.

The following example is for a 1-μs sample time and a 10-bit conversion. The external input circuit must be able to charge a sample capacitor (C_S) through a series resistance (R_1) to an accurate voltage, given a dc leakage (I_L). For purposes of this example, assume C_S of 2 pf, R_1 of 1.2 kΩ, and I_L of 1 μA.

External circuits with source impedances of 1 kΩ or less can maintain an input voltage within a tolerance of about 0.2 LSB (1.0 kΩ × 1.0 μA = 1.0 mV) given the dc leakage. Source impedances above 5 kΩ can result in an external error of at least one LSB due to the voltage drop caused by the 1-μA leakage. In addition, source impedances above 25 kΩ may degrade converter accuracy because the internal sample capacitor will not charge completely during the sample time.

Typically, leakage is much lower than the maximum specification specified by the microcontroller manufacturer. Given typical leakage, source impedance may be increased substantially before a one-LSB error is apparent. However, a high source impedance may prevent the internal sample capacitor from fully charging during the sample window. This error can be calculated using the following formula:

$$\text{Error (LSBs)} = \left(e^{\frac{-T_{SAM}}{RC}} \right) \times 1024$$

where T_{SAM} = sample time, μs
$R = R_{SOURCE} + R_1$, Ω
$C = C_S$, μf

The effects of this error can be minimized by connecting an external capacitor C_{EXT} from the input pin to ANGND. The external signal will charge C_{EXT} to the source voltage. When the channel is sampled, a small portion of the charge stored in C_{EXT} will be transferred to the internal sample capacitor. The ratio of C_S to C_{EXT} causes the loss in accuracy. If C_{EXT} is .005 μf or greater, the maximum error will be –0.6 LSB.

Placing an external capacitor on each analog input also reduces the sensitivity to noise because the capacitor combines with series resistance in the external circuit to form a low-pass filter. In practice, one should include a small series resistance prior to the external capacitor on the analog input pin and choose the largest capacitor value practical, given the frequency of the signal being converted. This provides a low-pass filter on the input, while the resistor also limits input current during overvoltage conditions.

11.6.4 Analog References

To achieve maximum noise isolation, on-chip A/D converters typically separate the internal A/D power supply from the rest of the microcontroller's power supply lines. Separate supply

pins, V_{ref} and An_{gnd}, usually supply both the reference and digital voltages for the A/D converter. Keep in mind that V_{ref} and An_{gnd} are the reference for a large resistor ladder on successive approximation converters. Any variation in these supplies will directly affect the reference voltage taps within the ladder, which in turn directly affect A/D conversion accuracy.

If the on-chip A/D converter is not being used, or if accuracy is not a concern, the Vref and Angnd pins can simply be connected to V_{cc} and V_{ss}, respectively. However, since the reference supply levels strongly influence the absolute accuracy of the A/D converter, a precision, well-regulated reference should be used to supply V_{ref} to achieve the highest performance levels. It is also important to use bypass capacitors between V_{ref} and An_{gnd} to minimize any noise that may be present on these supplies. In noise-sensitive applications running at higher frequencies, the use of separate ground planes within the PCB (circuit board) should be considered, possibly as shown in Fig. 11.48. This will help minimize ground loops and provide for a stable A/D reference.

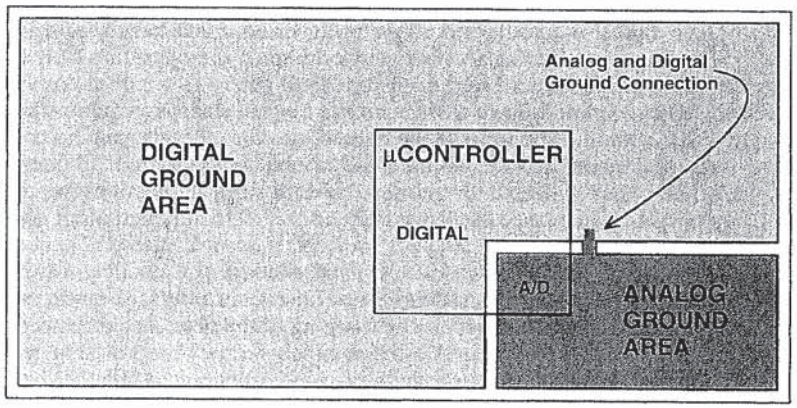


FIGURE 11.48 Example of separate analog and digital ground planes.

11.7 FAILSAFE METHODOLOGIES

The amount and complexity of automotive electronics incorporated into automobiles has increased at an incredible rate over the last decade. This trend has contributed significantly towards the impressive safety record of modern automobiles. Although microcontrollers are extremely reliable electronic devices, it is possible for failures to occur, either elsewhere in the module or within the microcontroller itself. It is critical that these failures be detected and responded to as quickly as possible in safety-related applications such as automotive antilock braking. If proper failsafe methodologies and good programming practices are followed, the chances of a failure going undetected are drastically reduced. The application of *failure mode and effect analysis* (FMEA) is an excellent tool for identifying potential failure modes, detection strategies, and containment methods. Used properly, FMEA will assist the designer in providing a high-quality, reliable automotive module. Although the scope of this chapter does not provide for a discussion on this topic, the author highly encourages the use of FMEA.

11.7.1 Hardware Failsafe Methods

Sometimes a hardware solution is required for detection of and response to certain failure modes. It is difficult for software alone to detect failures external to the device. As an exam-

read or drive an incorrect value. In this case, it can be difficult for software to detect because it would base its response on an incorrect value read from a pin.

Watchdog Timers (WDTs). An on-chip hardware watchdog is an excellent method of detecting failures which otherwise may go undetected. An example of this would be a microcontroller fetching either erroneous address or data (due to noise, etc.) and becoming "lost." WDTs commonly utilize a dedicated 16-bit counter, which provides for a count of 2^{16}(65,536) clocked at a rate of one tick per state time. If users wish to take advantage of this feature, they simply write to a register to enable the count. Once enabled, the user program must periodically clear the watchdog by writing a specific bit pattern to the Watchdog SFR. Clearing the WDT at least every 4.1 ms (65,535 * 1 state time at 16 MHz) will prevent the device from being reset. The strategy is that if the WDT initiates a reset, the assumption can be made that a failure has occurred and the microcontroller has became lost.

External Failsafe Devices. It is common for systems to incorporate an external failsafe device, such as another microcontroller or an *application-specific integrated circuit* (ASIC). The function of a failsafe device is to monitor the operation of the primary microcontroller and determine if it is operating properly.

The simplest failsafe devices output a signal such as a square wave for the microcontroller to detect and respond to. If the microcontroller doesn't respond correctly, a reset is typically asserted by the failsafe and the ECU reverts to a safe mode of operation. More complex failsafe devices will actually monitor several critical functions for failures such as low Vcc, stopped or decreased oscillator frequency, shorted/opened input signals, and so forth.

Oscillator Failure Detection. It is possible for the clocking source (typically an oscillator) to fail for various reasons. Since most microcontrollers are static devices, a particularly difficult failure mode to detect is the clocking of the device at a reduced frequency. To detect this failure, an *oscillator failure detection* circuit is often integrated upon the microcontroller. This circuitry will detect if the oscillator clock input signal falls below a specified frequency, in which case an interrupt will be generated or the device will reset itself.

Redundancy/Cross-checking. A common failsafe methodology is achieved by designing a redundant, or backup, processor into the module. In this case, the secondary microcontroller usually executes a subset of the main microcontroller's code. The secondary microcontroller typically processes critical input data and performs cross-checks periodically with the main microcontroller to insure proper operation. A failsafe routine is initiated if data exchanged between the two devices did not correlate.

11.7.2 Software Failsafe Techniques

Failsafe methodologies implemented in software are ideal for detecting failure modes that can interfere with proper program flow. Examples of these types of failures include noise glitches, which are notorious for causing external memory systems to fetch invalid addresses. ROM/EPROM memory corruption could cause an ISR start address to be fetched from an invalid interrupt vector location. Interrupts occurring at a rate faster than anticipated can cause problems such as an overflowing stack. Fortunately, failure modes such as these can be dealt with by implementing software failsafe methods. It is simply good programming practice to anticipate these types of failure modes and provide a failsafe strategy to deal with them. Following are several software strategies commonly used to deal with specific types of failure modes:

Checksum. One possible error that must be accounted for is ROM/EPROM memory corruption. An effective method of detecting these types of failures is through the calculation of

a checksum during the initialization phase of a user's program. A checksum is the final value obtained as the result of performing some arithmetic operation upon every ROM/EPROM memory location. The obtained checksum is then compared against a stored checksum. If the two match, the ROM/EPROM contents are intact. An error routine is called if the two checksums do not match. The most common arithmetic operation used to perform a checksum is addition. The checksum is calculated by adding the contents of all memory locations. When the addition is performed, the carry is ignored which provides for a byte or word checksum. The final result is then used as the checksum.

Unused Interrupt Vectors. It is a rare occasion when all interrupt sources are enabled within an application. If, for some unforeseen reason, the program should vector to an unused interrupt source, some sort of failsafe routine should be implemented to respond to the failure. The failsafe routine could be as simple as vectoring to a reset instruction or it can be as complicated as the programmer wishes.

Unused Memory Locations. A strategy should be in place to detect if, for some unforeseen reason, the program sequence should begin to execute in an unused area of ROM/EPROM. It is uncommon for the user's code to fill the entire ROM/EPROM array of a microcontroller. It is good programming practice to fill any unused locations with the opcode of an instruction such as *Reset.* On the MCS-96 family, executing the opcode FFh (which happens to be the blank state of EPROM) will initiate a reset sequence. Other microcontroller families have similar instructions.

Unimplemented Opcode Interrupt Vectors. Microcontrollers often dedicate one or more interrupt vectors for failsafe purposes. An *unimplemented opcode* interrupt is designed to detect corrupted instruction fetches. The corresponding interrupt service routine is executed whenever an unsupported opcode is fetched for execution. The interrupt service routine contains the user's failsafe routine, which is tailored to address this failure for the specific application.

11.8 FUTURE TRENDS

There are several significant trends developing in automotive electronics as ECU manufacturers strive to meet the challenges of a demanding automotive electronics market. The challenges that are bringing about these trends are: decreasing cost targets, decreasing form-factor goals, increasing performance requirements, and increasing system-to-system communication requirements. As the most significant component of an ECU, microcontrollers are bearing the brunt of these demands. This section will discuss these challenges and provide some insight into some of the ways microcontroller manufacturers are addressing these trends.

11.8.1 Decreasing Cost Targets

Microcontroller manufacturers are aproaching cost reduction in two ways: indirectly and directly. *Indirect* cost reductions are achieved by integrating features onto the microcontroller which allow the system designer to reduce cost elsewhere in the system. The key to this approach being successful is in the microcontroller manufacturer's ability to integrate the feature cheaper than the cost of providing an external solution. Integration is not always the cheaper solution, therefore each feature must be evaluated individually to determine the feasability of integration. An example of an indirect cost reduction would be the integration of watchdog and failsafe functions onto the microcontroller. This would eliminate the need for external watchdog components and thus reduce cost.

Another example would be through the integration of communications protocols such as CAN (Controller Area Network) or J1850 onto the same piece of silicon as the microcontroller. This will reduce the system chip count (and thus cost) by at least one integrated circuit device (the CAN chip) and several interfacing components. In most cases, a reduced chip count will translate into a PCB size decrease and a cost savings.

By *directly* addressing decreasing cost targets, microcontroller manufacturers actually reduce the manufacturing cost of the microcontroller itself. An example of this would be utilizing smaller geometry processes for manufacturing. Process geometry refers to the transistor channel width that is implanted onto a piece of silicon for a given fabrication process. Smaller processes allow for a higher transistor density on an integrated circuit. Higher densities allow for smaller die sizes which relate to lower costs. Most automotive microcontrollers manufactured today are fabricated with a 1.0-micron, or larger, process. As technology advances, future automotive microcontrollers will be manufactured upon submicron processes, such as 0.6 micron.

11.8.2 Increasing Performance Requirements

Automotive applications, such as ABS and engine control, require the processing of a substantial amount of data within a limited period of time. Higher-performance microcontrollers are required as system complexity increases and new features, such as traction control and vehicle dynamics, are incorporated into the ECU.

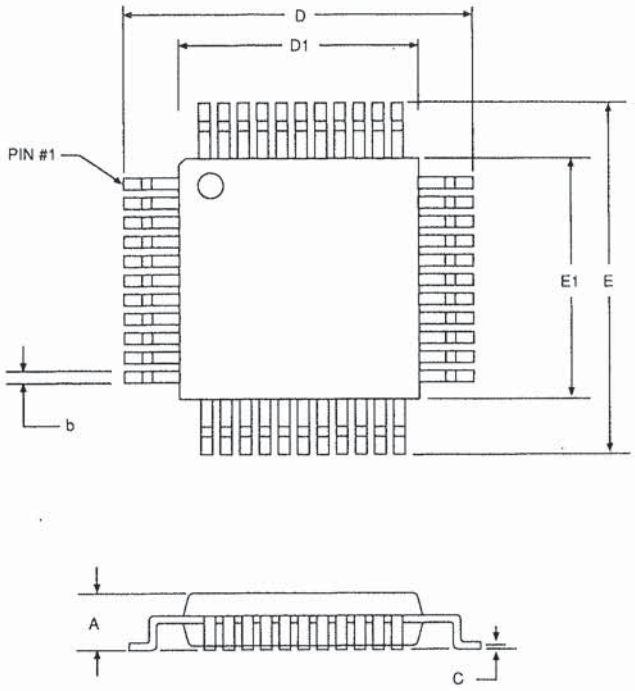
Microcontroller performance can be directly related to speed. Therefore, a rather straightforward approach to increased performance is through increasing clock speed. Today, most automotive microcontrollers have the capability to operate at frequencies of 16 MHz with speeds up to 20 MHz becoming common. Future microcontrollers will have the ability to be operated at frequencies of 24 or even 32 MHz. This allows more code to be executed in the same amount of time, and thus improves performance.

The method of increasing performance is not limited to just increasing the clock frequency. Microcontrollers can also achieve higher performance by enhancing existing peripherals for more efficient operation. This may be in the form of improved data handling or new features which suit the needs of a specific automotive application.

11.8.3 Increasing System-to-System Communication Requirements

The increasing complexity of automotive electronics requires that an increasing amount of information (diagnostics, etc.) be shared between various ECUs within an automobile. To fulfill this need, high-speed data links are utilized to transfer messages between multiple ECUs utilizing protocols such as Bosch's Controller Area Network (CAN) and SAE's J1850. To provide further size and cost savings, it is becoming more and more common to see these protocols supported or integrated onto automotive microcontrollers as opposed to separate integrated circuits.

The theory of centralized body computing is also receiving a closer look due to increased government regulations concerning fuel economy and diagnostics. A centralized body computer would link all ECUs (ABS and traction control, engine, transmission, suspension, instrumentation, etc.) together over a high-speed, in-vehicle serial network. One common scenario would have the central computer (possibly a microprocessor as opposed to a microcontroller) performing the more intense data-crunching tasks, while peripheral microcontrollers located in each individual ECU would perform system I/O functions. These communication protocols provide for efficient two-wire, high-speed serial communications between multiple ECUs utilizing protocols such as CAN and J1850. Supporting these protocols places additional loading upon the microcontroller. Increased microcontroller performance is necessary to manage this loading.



SHRINK QUAD FLATPACK				
SYMBOL	DESCRIPTION	MIN.	NOM.	MAX.
N	Lead Count		80	
A	Overall Height			1.66
A1	Stand Off	0.00	-	
b	Lead Width	0.14	0.20	0.26
c	Lead Thickness	0.117	0.127	0.177
D	Terminal Dimension	13.70	14.00	14.30
D1	Package Body		12.0	
E	Terminal Dimension	13.70	14.00	14.30
E1	Package Body		12.0	
e1	Lead Pitch	0.40	0.50	0.60
L1	Foot Length	0.35	0.50	0.70
T	Lead Angle	0.0°		10.0°
Y	Coplanarity			0.10

FIGURE 11.49 Shrink quad flat pack (SQFP) package.

11.8.4 Decreasing Form Factor Goals

Automobile manufacturers striving to build compact, more fuel efficient automobiles are putting pressure upon ECU suppliers to build smaller, lighter modules.

ECU size is directly affected by PCB size. The easiest way to achieve a smaller PCB is through integration and utilization of smaller integrated circuit packages. To support this demand, automotive microcontroller manufacturers are beginning to offer smaller, fine-pitch packages. A package commonly used today is the 68-lead plastic leaded chip carrier (PLCC) which has its pins placed on 1.27-mm centers and a body that is 24.3 mm^2. An example of a possible automotive package solution for the future would be the 80-lead shrink quad flat pack (SQFP, Fig. 11.49) which has pins on 0.50-mm centers and a body that is 12.0 mm^2. It is relatively easy to see that the SQFP package offers 12 additional pins in a package that is half the size of the PLCC. This high pin density, fine-pitch packaging allows for a smaller package to be utilized for the same size microcontroller die.

Another technology that is quickly becoming popular for automotive applications is referred to as *multichip modules* (MCMs). An MCM is a collection of unpackaged integrated circuit die (from various manufacturers) which are mounted upon a common substrate and packaged together. The advantage of MCMs is that they require much less PCB space than if the ICs were packaged separately.

GLOSSARY

Accumulator A register within a microcontroller that holds data, particularly data on which arithmetic or logic operations are to be performed.

Arithmetic logic unit (ALU) The part of a microcontroller that performs arithmetic and logic operations.

Analog-to-digital converter An electronic device that produces a digital result that is proportional to the analog input voltage.

Assembly language A low-level symbolic programming language closely resembling machine language.

Central processing unit (CPU) The portion of a computer system or microcontroller that controls the interpretation and execution of instructions and includes arithmetic capability.

EPROM Erasable and programmable read-only memory.

High-speed input/output unit (HSIO) A microcontroller peripheral which has the capability to either capture the time at which a certain input event occurs or create an output event at a predetermined time, both relative to a common clock. HSIO events are configured by the programmer to occur automatically.

Interrupt service routine (ISR) A predefined portion of a computer program which is executed in response to a specific event.

Low-speed input/output The input/output of a digital signal by "manually" reading or writing a register location in software.

Machine language A set of symbols, characters, or signs used to communicate with a computer in a form directly usable by the computer without translation.

Program counter (PC) A microcontroller register which holds the address of the next instruction to be executed.

Program status word (PSW) A microcontroller register that contains a set of boolean flags which are used to retain information regarding the state of the user's program.

Pulse-width modulation (PWM) The precise and timely creation of negative and positive waveform edges to achieve a waveform with a specific frequency and duty cycle.

Random access memory (RAM) A memory device which has both read and write capabilities so that the stored information (write) can be retrieved (reread) and be changed by applying new information to the inputs.

Read-only memory (ROM) A memory that can only be read and not written to. Data is either entered during the manufacturing process or by later programming; once entered, it is unalterable.

Register/arithmetic logic unit (RALU) A component of register-direct microcontroller architectures that allows the ALU to operate directly upon the entire register file.

Serial input/output (SIO) A method of digital communication in which a group of data bits is transferred one at a time, sequentially over a single data line.

Special function register (SFR) A microcontroller RAM register which has a specific, dedicated function assigned to it.

BIBLIOGRAPHY

ASM96 Assembler User's Manual, Intel Corp., 1992.

Automotive Electrics/Electronics, Robert Bosch GmbH, 1988.

Automotive Handbook, Intel Corporation, 1994.

Automotive Handbook, 2d ed., Robert Bosch GmbH, 1986.

Corell, Roger J., "How are semiconductor suppliers responding to the growing demand for automotive safety features?," *Intel Corp.,* 1993.

Davidson, Lee S., and Robert M. Kowalczyk, "Microcontroller technology enhancements to meet ever-increasing engine control requirements," Intel Corp., 1992.

Fink, Donald G., and Donald Christiansen, *Electronics Engineers' Handbook,* 3d ed. McGraw-Hill, 1989.

iC-96 Compiler User's Manual, Intel Corp., 1992.

Introduction to MOSFETS and EPROM Memories, Intel Corp., 1990.

MCS®-51 Microcontroller Family User's Manual, Intel Corp., 1993.

Millman, Jacob, and Arvin Grabel, *Microelectronics,* McGraw-Hill, 1987.

Packaging Handbook, Intel Corporation, 1994.

Ribbens, William B., *Understanding Automotive Electronics,* Howard Sams Company, Carmel, Ind. 1992.

8XC196Kx User's Manual, Intel Corporation, 1992.

8XC196KC/8XC196KD User's Manual, Intel Corp., 1992.

ABOUT THE AUTHOR

David S. Boehmer is currently a senior technical marketing engineer for the Automotive Operation of Intel's Embedded Microprocessor Division located in Chandler, Ariz. He is a member of SAE.