

# Rabbit 2000™ Microprocessor User's Manual

**Revision A**



# **Rabbit 2000 Microprocessor User's Manual**

Part Number 019-0069 • Revision A

Last revised on November 16, 1999 • Printed in U.S.A.

## **Copyright**

© 1999 Rabbit Semiconductor • All rights reserved.

Rabbit Semiconductor reserves the right to make changes and improvements to its products without providing notice.

## **Trademarks**

- Dynamic C® is a registered trademark of Z-World
- Z80/Z180™ is a trademark of Zilog, Inc.

## **Notice to Users**

Rabbit Semiconductor products are not authorized for use as critical components in life-support devices or systems unless a specific written agreement regarding such intended use is entered into between the customer and Rabbit Semiconductor prior to use. Life-support devices or systems are devices or systems intended for surgical implantation into the body or to sustain life, and whose failure to perform, when properly used in accordance with instructions for use provided in the labeling and user's manual, can be reasonably expected to result in significant injury.

No complex software or hardware system is perfect. Bugs are always present in a system of any size. In order to prevent danger to life or property, it is the responsibility of the system designer to incorporate redundant protective mechanisms appropriate to the risk involved.

## **Company Address**

### **Rabbit Semiconductor**

2932 Spafford Street  
Davis, California 95616-6800  
USA

Telephone: (530) 757-8400

Facsimile: (530) 757-8402

Web site: <http://www.rabbitsemiconductor.com>

# Table of Contents

1.	Introduction.....	7
1.1	Features and Specifications .....	7
1.2	Summary of Rabbit Advantages .....	11
2.	Rabbit Design Features .....	13
2.1	The Rabbit 8-bit Processor vs. 16-bit and 32-bit Processors .....	13
2.2	Overview of On-Chip Peripherals .....	14
2.2.1	Serial Ports .....	14
2.2.2	System Clock.....	14
2.2.3	Time/Date Oscillator .....	15
2.2.4	Parallel I/O.....	15
2.2.5	Slave Port .....	15
2.2.6	Timers .....	16
2.3	Design Standards .....	18
2.3.1	Programming Port .....	18
2.3.2	Standard BIOS.....	18
2.4	Dynamic C Software Support for the Rabbit.....	18
3.	Details on Rabbit Microprocessor Features .....	19
3.1	Processor Registers .....	19
3.2	Memory Mapping .....	20
3.2.1	Extended Code Space.....	24
3.2.2	Practical Memory Considerations .....	25
3.3	Instruction Set Outline .....	26
3.3.1	Instructions to Load Immediate Data To a Register .....	27
3.3.2	Instructions to Load or Store Data from or to a Constant Address .....	27
3.3.3	Instructions to Load or Store Data Using an Index Register .....	28
3.3.4	Register to Register Move Instructions .....	29
3.3.5	Register Exchanges .....	29
3.3.6	Push and Pop Instructions .....	30
3.3.7	16-bit Arithmetic and Logical Operations.....	30
3.3.8	Input/Output Instructions .....	33
3.4	How to Do It in Assembly Language—Tips and Tricks.....	33
3.4.1	Zero HL in 4 Clocks .....	33
3.4.2	Exchanges Not Directly Implemented.....	34
3.4.3	Manipulation of Boolean Variables .....	34
3.4.4	Comparisons of Integers.....	35
3.4.5	Atomic Moves from Memory to I/O Space.....	37
3.5	Interrupt Structure .....	38
3.5.1	Interrupt Priority .....	38
3.5.2	Multiple External Interrupting Devices.....	40
3.5.3	Privileged Instructions, Critical Sections and Semaphores.....	40
3.5.4	Critical Sections .....	41
3.5.5	Semaphores Using Bit B,(HL) .....	41
3.5.6	Computed Long Calls and Jumps.....	42

4.	Rabbit Capabilities.....	43
4.1	Precisely Timed Output Pulses.....	43
4.1.1	Pulse Width Modulation to Reduce Relay Power.....	44
4.2	Open-Drain Outputs Used for Key Scan.....	46
4.3	Cold Boot .....	46
4.4	The Slave Port .....	47
4.4.1	Slave Rabbit As A Protocol UART .....	48
5.	Pin Assignments and Functions .....	49
5.1	Package Schematic and Pin Names.....	49
5.2	Package Mechanical Dimensions.....	50
5.3	Rabbit Pin Descriptions.....	52
5.4	Bus Timing.....	58
5.5	Description of Pins with Alternate Functions .....	59
5.6	Register and Interrupt Vector Summary.....	61
6.	Rabbit Internal I/O Registers .....	63
7.	Miscellaneous I/O Functions .....	67
7.1	Rabbit Oscillators and Clocks.....	67
7.2	Clock Doubler .....	69
7.3	Controlling Power Consumption.....	70
7.4	Output Pins CLK, STATUS, /WDTOUT, /IOBEN .....	71
7.5	Time/Date Clock (Real-Time Clock).....	71
7.6	Watchdog Timer .....	73
7.7	System Reset .....	74
7.8	Rabbit Interrupt Structure.....	75
7.8.1	External Interrupts .....	77
7.9	Bootstrap Operation .....	79
8.	Rabbit Memory Mapping and Interface.....	81
8.1	Memory-Mapping Unit .....	81
8.2	Memory Interface Unit.....	83
8.3	Memory Bank Control Register Functions .....	83
8.3.1	Optional A16, A19 Inversions by Segment (/CS1 Enable) .....	84
8.4	Allocation of Extended Code and Data.....	84
8.5	How the Compiler Compiles to Memory.....	85
9.	Parallel Ports .....	87
9.1	Parallel Port A .....	87
9.2	Parallel Port B .....	88
9.3	Parallel Port C .....	88
9.4	Parallel Port D.....	89
9.5	Parallel Port E.....	92
10.	I/O Bank Control Registers.....	95

11.	Timers .....	97
11.1	Timer A .....	98
11.1.1	Timer A I/O Registers .....	99
11.1.2	Practical Use of Timer A .....	100
11.2	Timer B .....	101
11.2.1	Using Timer B .....	102
12.	Rabbit Serial Ports .....	105
12.1	Register Layout Serial Port .....	105
12.2	Serial Port Interrupt .....	108
12.3	Transmit Serial Data Timing .....	109
12.4	Receive Serial Data Timing .....	110
12.5	Clocked Serial Ports .....	110
12.6	Serial Port Software Suggestions .....	112
12.6.1	Controlling an RS-485 Driver and Receiver .....	114
12.6.2	Transmitting Dummy Characters .....	114
12.6.3	Transmitting and Detecting a Break .....	114
12.6.4	Using A Serial Port to Generate a Periodic Interrupt .....	115
12.6.5	Working With Two Stop Bits or a Parity Bit .....	115
12.6.6	Data Framing/Modbus .....	115
13.	Rabbit Slave Port .....	117
13.1	Hardware Design of Slave Port Interconnection .....	120
13.2	Slave Port Registers .....	121
13.3	Applications and Communications Protocols for Slaves .....	123
13.3.1	Slave Applications .....	123
13.3.2	Master-Slave Messaging Protocol .....	123
14.	Rabbit Hardware Design and Development .....	127
14.1	RS-485 Communication Interface .....	127
14.2	RS-232 Communication Interface .....	127
14.3	Analog-to-Digital Converters .....	127
14.4	Digital-to-Analog Converters .....	127
14.5	High-Voltage Drivers .....	127
14.6	Clocks .....	127
14.7	Low-Power Design .....	128
14.8	Basic Memory Design .....	129
14.8.1	Memory Access Time .....	129
14.8.2	Precautions for Unprogrammed Flash Memory .....	129
14.9	PC Board Layout and Memory Line Permutation .....	131
15.	AC Timing Specifications .....	133
15.1	Current Consumption .....	137
16.	Rabbit Software .....	139
16.1	Reading and Writing I/O Registers and Shadow Registers .....	139
16.2	Shadow Registers .....	140
16.3	Timer and Clock Usage .....	141
16.4	WatchDog Support Software .....	142
16.4.1	The Watchdog Hardware .....	143
16.4.2	The Virtual Watchdog System .....	143

17.	Rabbit Standard BIOS.....	145
17.1	The BIOS—More Details.....	145
17.2	BIOS Assumptions.....	146
17.3	Periodic Interrupt and Real-Time Clock BIOS Services.....	146
17.3.1	Real-Time Clock Support .....	147
17.3.2	Watchdog Timer Support .....	147
17.3.3	Power Management Support.....	147
17.3.4	Flash Memory Write Support .....	148
18.	Rabbit Instructions .....	149
18.1	Load Immediate Data .....	149
18.2	Load and Store to an Immediate Address .....	150
18.3	8-bit Indexed Load and Store .....	150
18.4	16-bit Indexed Loads and Stores .....	150
18.5	16-bit Load and Store 20-bit Address .....	150
18.6	Register to Register Moves .....	151
18.7	Exchange Instructions .....	151
18.8	Stack Manipulation Instructions.....	152
18.9	16-bit Arithmetic and Logical Operations .....	152
18.10	8-bit Arithmetic and Logical Operations .....	153
18.11	8-bit Bit Set, Reset and Test Instructions .....	154
18.12	8-bit Increment and Decrement.....	154
18.13	8-bit Fast A register Operations .....	154
18.14	8-bit Shifts and Rotates .....	155
18.15	Instruction Prefixes .....	156
18.16	Block Move Instructions .....	156
18.17	Control Instructions - Jumps and Calls .....	156
18.18	Miscellaneous Instructions.....	157
18.19	Privileged Instructions.....	157
19.	Differences Rabbit vs. Z80/Z180 Instructions.....	159
20.	Instructions in Alphabetical Order With Binary Encoding.....	161
	Appendix A.....	167
A.1	Rabbit Programming Port.....	167
A.1.1	Use of the Programming Port as a Diagnostic/Setup Port .....	167
A.1.2	Alternate Programming Port.....	168
A.2	Suggested Rabbit Crystal Frequencies.....	169
	Legal Notice.....	171

# 1. Introduction

Rabbit Semiconductor was formed expressly to design a better microprocessor for use in small and medium-scale controllers. The first product is the *Rabbit 2000* microprocessor. The Rabbit 2000 designers have had years of experience using Z80, Z180 and HD64180 microprocessors in small controllers. The Rabbit shares a similar architecture and a high degree of compatibility with these microprocessors, but it is a vast improvement.

The Rabbit has been designed in close cooperation with Z-World, Inc., a long-time manufacturer of low-cost single-board computers. Z-World's products are supported by an innovative C-language development system (Dynamic C). Z-World is providing the software development tools for the Rabbit.

The Rabbit is easy to use. Hardware and software interfaces are as uncluttered and are as foolproof as possible. The Rabbit has outstanding computation speed for a microprocessor with an 8-bit bus. This is because the Z80-derived instruction set is very compact and the design of the memory interface allows maximum utilization of the memory bandwidth. The Rabbit races through instructions.

Traditional microprocessor hardware and software development is simplified for Rabbit users. In-circuit emulators are not needed and will not be missed by the Rabbit developer. Software development is accomplished by connecting a simple interface cable from a PC serial port to the Rabbit-based target system.

## 1.1 Features and Specifications

- 100-pin PQFP package. Operating voltage 2.7 V to 5 V. Clock speed to 30 MHz. All specifications are given for both industrial and commercial temperature and voltage ranges. Rabbit microprocessors cost under \$10 in moderate quantities.  
Industrial specifications are for a voltage variation of 10% and a temperature range from  $-40^{\circ}\text{C}$  to  $+85^{\circ}\text{C}$ . Commercial specifications are for a voltage variation of 5% and a temperature range from  $0^{\circ}\text{C}$  to  $70^{\circ}\text{C}$ .
- 1-megabyte code space allows C programs with up to 50,000+ lines of code. The extended Z80-style instruction set is C-friendly, with short and fast instructions for most common C operations.
- Four levels of interrupt priority make a fast interrupt response practical for critical applications. The maximum time to the first instruction of an interrupt routine is about  $1\ \mu\text{s}$  at a clock speed of 25 MHz.
- Access to I/O devices is accomplished by using memory access instructions with an I/O prefix. Access to I/O devices is thus faster and easier compared to processors with a restricted I/O instruction set.

- The hardware design rules are simple. Up to six static memory chips (such as RAM and flash EPROM) connect directly to the microprocessor with no glue logic. Even larger amounts of memory can be handled by using parallel I/O lines as high-order address lines. The Rabbit runs with no wait states at 24 MHz with a memory having an access time of 70 ns. There are two clocks per memory access. Most I/O devices may be connected without glue logic.

The memory cycle is two clocks long. A clean memory and I/O cycle completely avoid the possibility of tri-state fights. Peripheral I/O devices can usually be interfaced in a glueless fashion using pins programmable as I/O chip selects, I/O read strobes or I/O write strobe pins. A built-in clock doubler allows ½-frequency crystals to be used to reduce radiated emissions.

- The Rabbit may be cold-booted via a serial port or the parallel access slave port. This means that flash program memory may be soldered in unprogrammed, and can be reprogrammed at any time without any assumption of an existing program or bios. A Rabbit that is slaved to a master processor can operate entirely with volatile RAM, depending on the master for a cold program boot.
- There are 40 parallel I/O lines (shared with serial ports). Some I/O lines are timer synchronized, which permits precisely timed edges and pulses to be generated under combined hardware and software control.
- There are four serial ports. All four serial ports can operate asynchronously in a variety of customary operating modes; two of the ports can also be operated synchronously to interface with serial I/O devices. The baud rates can be very high—1/32 the clock speed for asynchronous operation and 1/8 the clock speed in synchronous mode. In asynchronous mode, the Rabbit, like the Z180, supports sending flagged bytes to mark the start of a message frame. The flagged bytes have 9 data bits rather than 8 data bits; the extra bit is located after the first 8 bits, where the stop bit is normally located, and marks the start of a message frame.
- A slave port allows the Rabbit to be used as an intelligent peripheral device slaved to a master processor. The 8-bit slave port has six 8-bit registers, 3 for each direction of communication. Independent strobes and interrupts are used to control the slave port in both directions. Only a Rabbit and a RAM chip are needed to construct a complete slave system if the clock and reset are shared with the master processor
- The built-in battery-backable time/date clock uses an external 32.768 kHz crystal. The time/date clock can also be used to provide periodic interrupts every 488 μs. Typical battery current consumption is 25 μA with the suggested battery circuit. An alternative circuit provides means for substantially reducing this current.
- Numerous timers and counters (six all together) can be used to generate interrupts, baud rate clocks, and timing for pulse generation.



- The built-in main clock oscillator uses an external crystal or more usually a ceramic resonator. Typical resonator frequencies are in the range of 1.8 MHz to 29.5 MHz. Since precision timing is available from the separate 32.768 kHz oscillator, a low-cost ceramic resonator with ½ percent error is generally satisfactory. The clock can be doubled or divided by 8 to modify speed and power dynamically. The I/O clock, which clocks the serial ports, is divided separately so as not to affect baud rates and timers when the processor clock is divided or multiplied. For ultra low power operation, the processor clock can be driven from the separate 32.768 kHz oscillator and the main oscillator can be powered down. This allows the processor to operate at approximately 100 µA and still execute instructions at the rate of approximately 10,000 instructions per second. This is a powerful alternative to sleep modes of operation used by other processors. The current is approximately 65 mA at 25 MHz and 5 V. The current is proportional to voltage and clock speed—at 3.3 V and 7.68 MHz the current would be 13 mA, and at 1 MHz the current is reduced to less than 2 mA. Flash memory with automatic power down (from AMD) should be used for operation at the lowest power.
- The excellent floating-point performance is due to a tightly coded library and powerful processing capability. For example, a 25 MHz clock takes 14 µs for a floating add, 13 µs for a multiply, and 40 µs for a square root. In comparison, a 386EX processor running with an 8-bit bus at 25 MHz and using Borland C is about 10 times slower.
- There is a built-in watchdog timer.
- The standard 10-pin programming port eliminates the need for in-circuit emulators. A very simple 10 pin connector can be used to download and debug software using Z-World's Dynamic C and a simple connection to a PC serial port. The incremental cost of the programming port is extremely small.

Figure 1 shows a block diagram of the Rabbit.

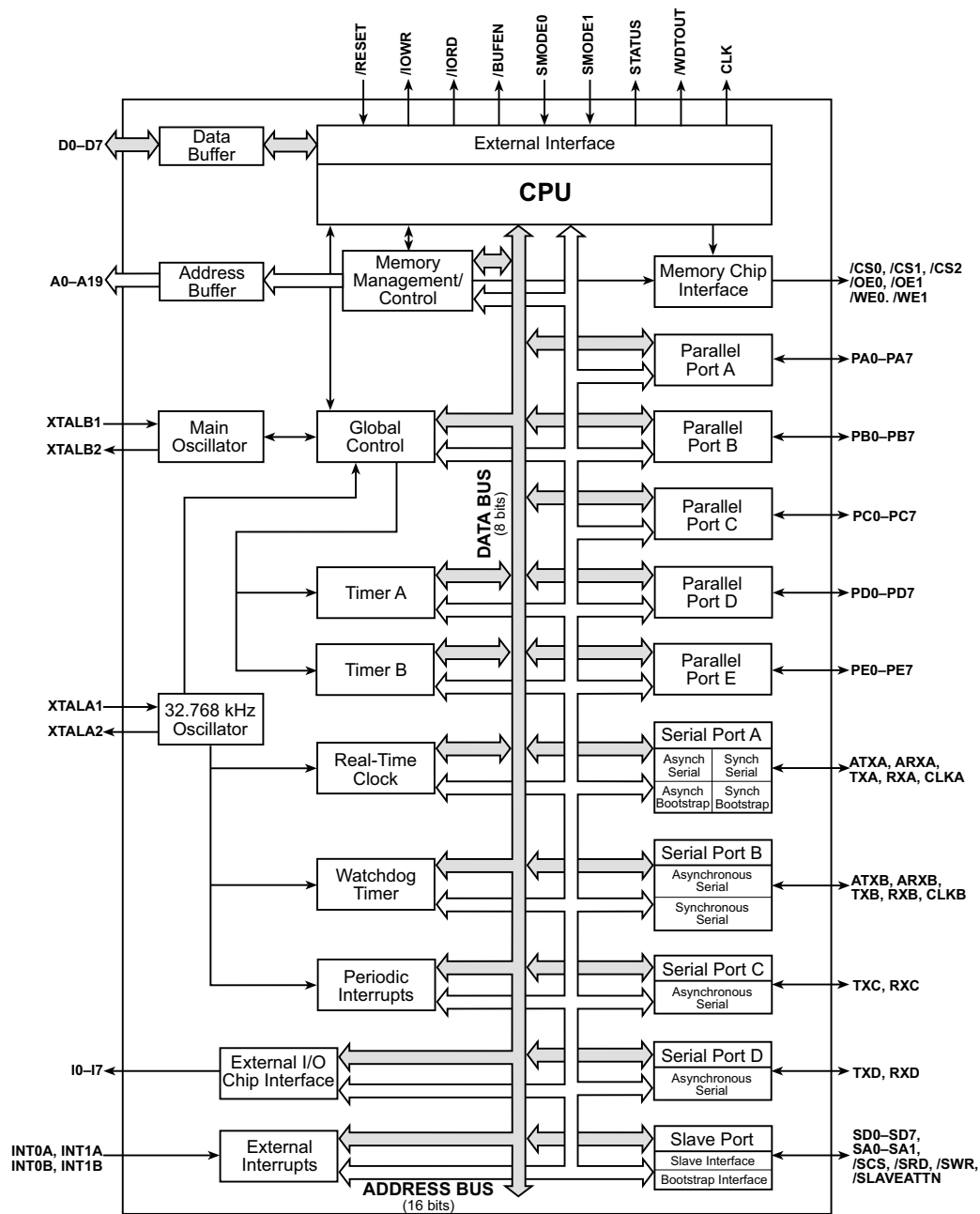


Figure 1. Block Diagram of the Rabbit Microprocessor

## 1.2 Summary of Rabbit Advantages

- The glueless architecture makes it is easy to design the hardware system.
- There are a lot of serial ports and they can communicate very fast.
- Precision pulse and edge generation is a standard feature.
- Interrupts can have multiple priorities.
- Processor speed and power consumption are under program control.
- The ultra low power mode can perform computations and execute logical tests since the processor continues to execute, albeit at 32 kHz.
- The Rabbit may be used to create an intelligent peripheral or a slave processor. For example, protocol stacks can be off loaded to a Rabbit slave. The master can be any processor.
- The Rabbit can be cold booted so unprogrammed flash memory can be soldered in place.
- You can write serious software, be it 1,000 or 50,000 lines of C code. The tools are there and they are low in cost.
- If you know the Z80 or Z180, you know most of the Rabbit.
- A simple 10-pin programming interface replaces in-circuit emulators and PROM programmers.
- The battery backable time/date clock is included.
- The standard Rabbit chip is made to industrial temperature and voltage specifications.



## 2. Rabbit Design Features

The Rabbit is an evolutionary design. The instruction set and the register layout is that of the Z80 and Z180. The instruction set has been augmented by a substantial number of new instructions. Some obsolete or redundant Z180 instructions have been dropped to make available efficient 1-byte opcodes for important new instructions. (see “Differences Rabbit vs. Z80/Z180 Instructions” on page 159.) The advantage of this evolutionary approach is that users familiar with the Z80 or Z180 can immediately understand the Rabbit. Existing source code can be assembled or compiled for the Rabbit with minimal changes.

Changing technology has made some features of the Z80/Z180 family obsolete, and these have been dropped. For example, the Rabbit has no special support for dynamic RAM but it has extensive support for static memory. This is because the price of static memory has decreased to the point that it has become the preferred choice for medium-scale embedded systems. The Rabbit has no support for DMA (direct memory access) because most of the uses for which DMA is traditionally used do not apply to embedded systems, or they can be accomplished better in other ways, such as fast interrupt routines, external state machines or slave processors.

Our experience in writing C compilers has revealed the shortcomings of the Z80 instruction set for executing the C language. The main problem is the lack of instructions for handling 16-bit words and for accessing data at a computed address, especially when the stack contains that data. New instructions correct these problems.

Another problem with many 8-bit processors is their slow execution and a lack of number-crunching ability. Good floating-point arithmetic is an important productivity feature in smaller systems. It is easy to solve many programming problems if an adequate floating-point capability is available. The Rabbit’s improved instruction set provides fast floating-point and fast integer math capabilities.

The Rabbit supports four levels of interrupt priorities. This is an important feature that allows the effective use of super fast interrupt routines for real-time tasks.

### 2.1 The Rabbit 8-bit Processor vs. 16-bit and 32-bit Processors

The Rabbit is a 16-bit processor with an 8-bit data bus. Because it make the most of its 8-bit bus and because it has a compact instructions set, its performance is as good as many 16-bit processors.

We hesitate to compare the Rabbit to 32-bit processors, but there are undoubtedly occasions where the user can use a Rabbit instead of a 32-bit processor and save a vast amount of money. Many Rabbit instructions are 1 byte long. In contrast, the minimum instruction length on most 32-bit RISC processors is 32 bits.

## 2.2 Overview of On-Chip Peripherals

The on-chip peripherals were chosen based on our experience as to what types of peripheral devices are most useful in small embedded systems. The major on-chip peripherals are the serial ports, system clock, time/date oscillator, parallel I/O, slave port, and timers. These are described below.

### 2.2.1 Serial Ports

There are four serial ports designated ports A, B, C, and D. All four serial ports can operate in an asynchronous mode up to a baud rate of the system clock divided by 32. The asynchronous ports can handle 7 or 8 data bits. A 9th bit address scheme, where an additional bit is sent to mark the first byte of a message, is also supported. The software can tell when the last byte of a message has finished transmitting from the output shift register - correcting an important defect of the Z180. This is important for RS-485 communication because the line driver cannot have the direction of transmission reversed until the last bit has been sent. In many UART's, including those on the Z180, it is difficult to generate an interrupt after the last bit is sent. Parity bits and multiple stop bits are not supported directly by the Rabbit, but can be accomplished with appropriate driving software.

Serial ports A and B can be operated alternately in the clocked serial mode. In this mode, a clock line synchronously clocks the data in or out. Either device of the 2 devices communicating can supply the clock. When the Rabbit provides the clock, the baud rate can be as fast as 1/8th of the system clock frequency, or more than 3,000,000 bits per second.

Serial port A has special features. It can be used to cold boot the system after reset. Serial port A is the normal port that is used for software development under Dynamic C.

### 2.2.2 System Clock

The main oscillator uses an external crystal with a frequency typically in the range from 1.8 MHz to 29.5 MHz. The processor clock is derived from the oscillator output by either doubling the frequency, using the frequency directly, or dividing the frequency by 8. The processor clock can also be driven by the 32.768 kHz oscillator for very low power operation, in which case the main oscillator can be shut down under software control.

Table 1 provides preliminary estimates of the operating power for selected clock speeds.

**Table 1. Preliminary Operating Power Estimates at Selected Clock Speeds**

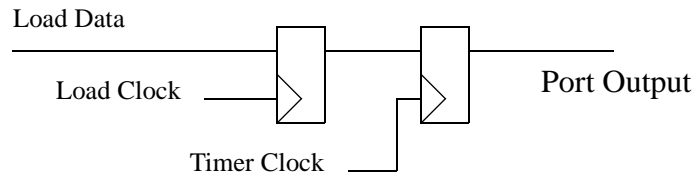
Clock Speed (MHz)	Voltage (V)	Current (mA)	Power (mW)	Clock Speed (MHz)	Voltage (V)	Current (mA)	Power (mW)
25.0	5.0	80	400	6.0	2.5	10	25
12.5	5.0	40	200	3.0	2.5	5	12
12.5	3.3	26	87	1.5	2.5	2.5	6
6.0	3.3	13	42	0.032	2.5	0.054	0.135

### 2.2.3 Time/Date Oscillator

The 32.768 kHz oscillator drives an external 32.768 kHz quartz crystal. The 32.768 kHz clock is used to drive a battery-backable (there is a separate power pin) internal 48-bit counter that serves as a real-time clock (RTC). The counter can be set and read by software and is intended for keeping the date and time. There are enough bits to keep the date for more than 100 years. The 32.768 kHz oscillator is also used to drive the watchdog timer and to generate the baud clock for serial port A during the cold boot sequence.

### 2.2.4 Parallel I/O

There are 40 parallel input/output lines divided among five 8-bit ports designated A through E. Most of the port lines have alternate functions, such as serial data or chip select strobes. Parallel ports D and E have the capability of timer-synchronized outputs. The output registers are cascaded.



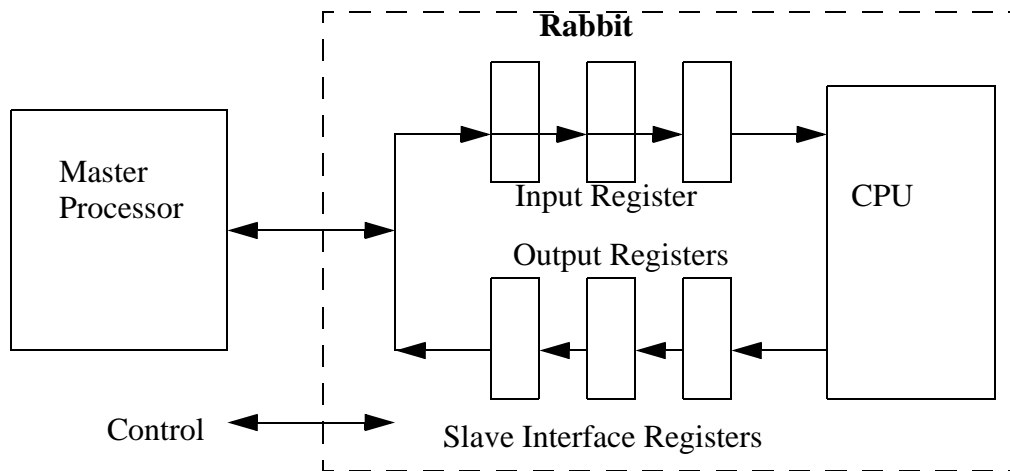
**Figure 2. Cascaded Output Registers for Parallel Ports D and E**

Stores to the port are loaded in the first-level register. That register in turn is transferred to the output register on a selected timer signal. The timer signal can also cause an interrupt that can be used to set up the next bit to be output on the next timer pulse. This feature can be used to generate precisely controlled pulses whose edges are positioned with high accuracy in time. Applications include communications signaling, pulse width modulation and driving stepper motors.

### 2.2.5 Slave Port

The slave port is designed to allow the Rabbit to be a slave to another processor, which could be another Rabbit. The port is shared with parallel port A and is a bidirectional data port. The master can read any of three registers selected via two select lines that form the register address and a read strobe that causes the register contents to be output by the port. These same registers can be written as I/O registers by the Rabbit slave. Three additional registers transmit data in the opposite direction. They are written by the master by means of the two select lines and a write strobe.

Figure 3 shows the data paths in the slave port.



**Figure 3. Slave Port Data Paths**

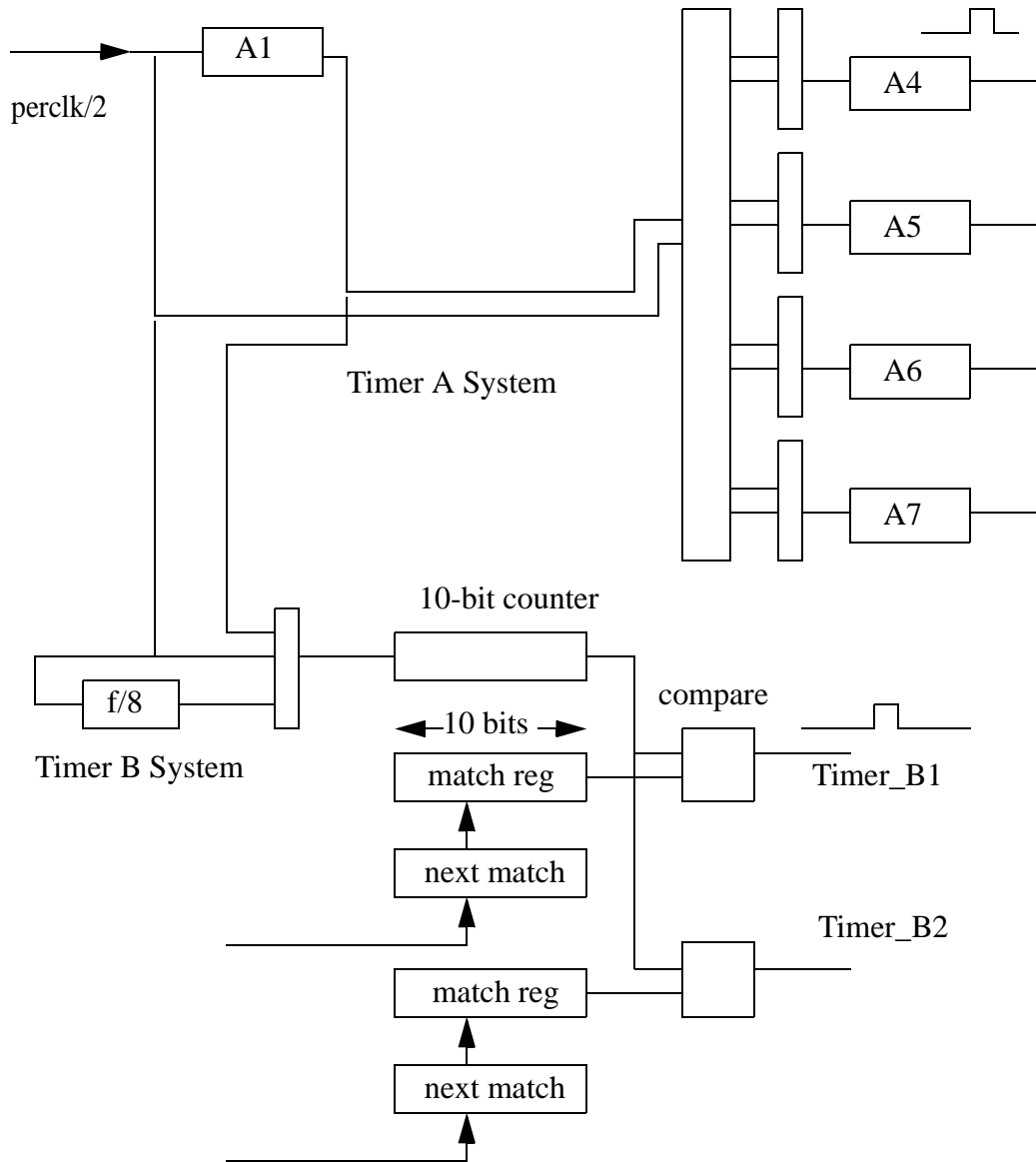
The slave Rabbit can read the same registers as I/O registers. When incoming data bits are written into one of the registers, status bits indicate which registers have been written, and an optional interrupt can be programmed to take place when the write occurs. When the slave writes to one of the registers carrying data bits outward, an attention line is enabled so that the master can detect the data change and be interrupted if desired. One line tells the master that the slave has read all the incoming data. Another line tells the master that new outgoing data bits are available and have not yet been read by the master. The slave port can be used to direct the master to perform tasks using a variety of communication protocols over the slave port.

### 2.2.6 Timers

The Rabbit has several timer systems. The periodic interrupt is driven by the 32.768 kHz oscillator divided by 16, giving an interrupt every 488  $\mu$ s if enabled. This is intended to be used as a general-purpose clock interrupt. Timer A consists of five 8-bit countdown and reload registers that can be cascaded up to two levels deep. Each countdown register can be set to divide by any number between 1 and 256. The output of four of the timers is used to provide baud clocks for the serial ports. Any of these registers can also cause interrupts and clock the timer-synchronized parallel output ports. Timer B consists of a 10-bit counter that can be read but not written. There are two 10-bit match registers and comparators. If the match register matches the counter, a pulse is output. Thus the timer can be programmed to output a pulse at a predetermined count in the future. This pulse can be used to clock the timer-synchronized parallel-port output registers as well as cause an interrupt. Timer B is convenient for creating an event at a precise time in the future under program control.



Figure 4 illustrates the Rabbit timers.



**Figure 4. Rabbit Timers**

## **2.3 Design Standards**

The same functionality can be accomplished in many ways using the Rabbit. By publishing design standards, or standard ways to accomplish common objectives, software and hardware support become easier.

### **2.3.1 Programming Port**

Rabbit Semiconductor publishes a specification for a standard programming port (see: “Rabbit Programming Port” on page 167) and provides a converter cable that may be used to connect a PC serial port to the standard programming interface. The interface is implemented using a 10-pin connector with two rows of pins on 2 mm. centers. The port is connected to Rabbit serial port A, to the startup mode pins on the Rabbit, to the Rabbit reset pin, and to a programmable output pin that is used to signal the PC that attention is needed. With proper precautions in design and software, it is possible to use serial port A as both a programming port and as a user-defined serial port, although this will not be necessary in most cases.

Rabbit Semiconductor supports the use of the standard programming port and the standard programming cable as a diagnostic and setup port to diagnosis problems or set up systems in the field.

### **2.3.2 Standard BIOS**

Rabbit Semiconductor provides a standard BIOS for the Rabbit. The BIOS is a software program that manages startup and shutdown, and provides basic services for software running on the Rabbit.

## **2.4 Dynamic C Software Support for the Rabbit**

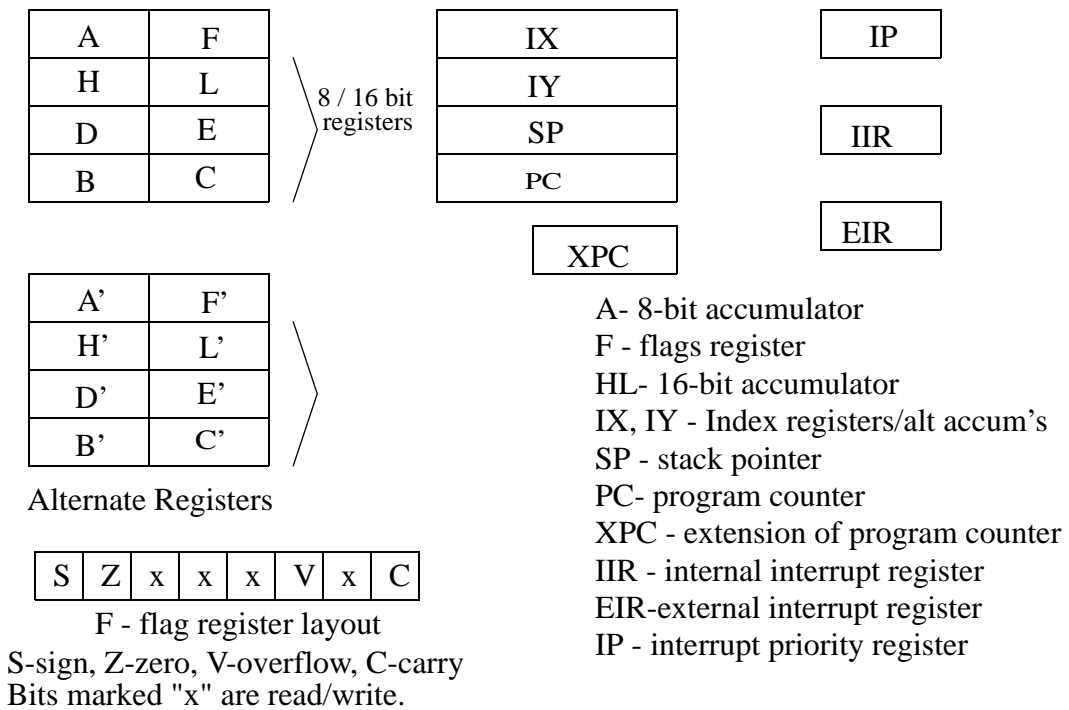
Dynamic C is Z-World’s interactive C language development system. Dynamic C runs on a PC under Windows 95/98 or Windows NT. It provides a combined compiler, editor and debugger. The usual method for debugging a target system based on the Rabbit is to implement the 10-pin programming connector that connects to the PC serial port via a standard converter cable. Dynamic C libraries contain highly perfected software to control the Rabbit. These includes drivers, utility and math routines and the debugging BIOS for Dynamic C.

In addition, the internationally known real-time operating system uC/OS-II is being ported to the Rabbit and will be available with some versions of Dynamic C.

### 3. Details on Rabbit Microprocessor Features

#### 3.1 Processor Registers

The Rabbit's registers are nearly identical to those of the Z180 or the Z80. Figure 5 shows the register layout. The XPC and IP registers are new. The EIR register is the same as the Z80 I register, and is used to point to a table of interrupt vectors for the externally generated interrupts. The IIR register occupies the same logical position in the instruction set as the Z80 R register, but its function is to point to an interrupt vector table for internally generated interrupts.



**Figure 5. Rabbit Registers**

The Rabbit (and the Z80/Z180) processor has two accumulators—the A register serves as an 8-bit accumulator for 8-bit operations such as *add* or *and*. The 16-bit register HL register serves as an accumulator for 16-bit operations such as *add hl,de*, which adds the 16-bit register DE to the 16-bit accumulator HL. For many operations IX or IY can substitute for hl as accumulators.

The register marked F is the flags register or status register, and it holds a number of flags that provide information about the last operation performed. The flag register cannot be accessed directly except by using the *pop af* and *push af* instructions. Normally the flags are tested by conditional jump instructions. The flags are set to mark the results of arithmetic and logic operations according to rules that are specified for each instruction. There are four unused read/write bits in the flag register that are available to the user via the *push*

af and pop af instructions. These bits should be used with caution since new-generation Rabbit processors could use these bits for new purposes.

The registers IX, IY and HL can also serve as index registers. They point to memory addresses from which data bits are fetched or stored. Although the Rabbit can address a megabyte or more of memory, the index registers can only directly address 64K of memory (except for certain extended addressing "ldp" instructions). The addressing range is expanded by means of the memory mapping hardware (see "Memory Mapping" on page 20) and by special instructions. For most embedded applications, 64K of *data* memory (as opposed to *code* memory) is sufficient. The Rabbit can handle a megabyte of code space in an efficient manner.

The register SP points to the stack that is used for subroutine and interrupt linkage as well as general-purpose storage.

A feature of the Rabbit (and the Z80/Z180) is the *alternate register set*. Two special instructions swap the alternate registers with the regular registers. The instruction `ex af,af'` exchanges the contents of AF with AF'. The instruction `exx` exchanges HL, DE, and BC with HL', DE', and BC'. Communication between the regular and alternate register set in the original Z80 architecture was difficult because the exchange instructions provided the only means of communication between the regular and alternate register sets. The Rabbit has new instructions that greatly improve communication between the regular and alternate register set. This effectively doubles the number of registers that are easily available for the programmer's use. It is not intended that the alternate register set be used to provide a separate set of registers for an interrupt routine, and Dynamic C does not support this usage because it uses both registers sets freely.

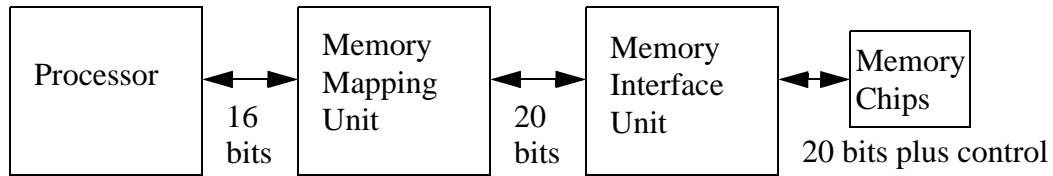
The IP register is the interrupt priority register. It contains four 2-bit fields that hold a history of the processor's interrupt priority. The Rabbit supports four levels of processor priority, something that exists only in a very restricted form in the Z80 or Z180.

## 3.2 Memory Mapping

Except for a handful of special instructions (see "16-bit Load and Store 20-bit Address" on page 150), the Rabbit instructions directly address a 64K data memory space. This means that the address fields in the instructions are 16 bits long and that the registers that may be used as pointers to memory addresses (index registers (ix, iy), program counter and stack pointer (sp)) are also 16 bits long.

Because Rabbit instructions use 16-bit addresses, the instructions are shorter and can execute much faster than, for example, 32-bit addresses. The executable code is also very compact. Even though these 16-bit addresses are a valuable asset, they do create some complications because a memory-mapping unit is needed in order to access a reasonable amount of memory for modern C programs.

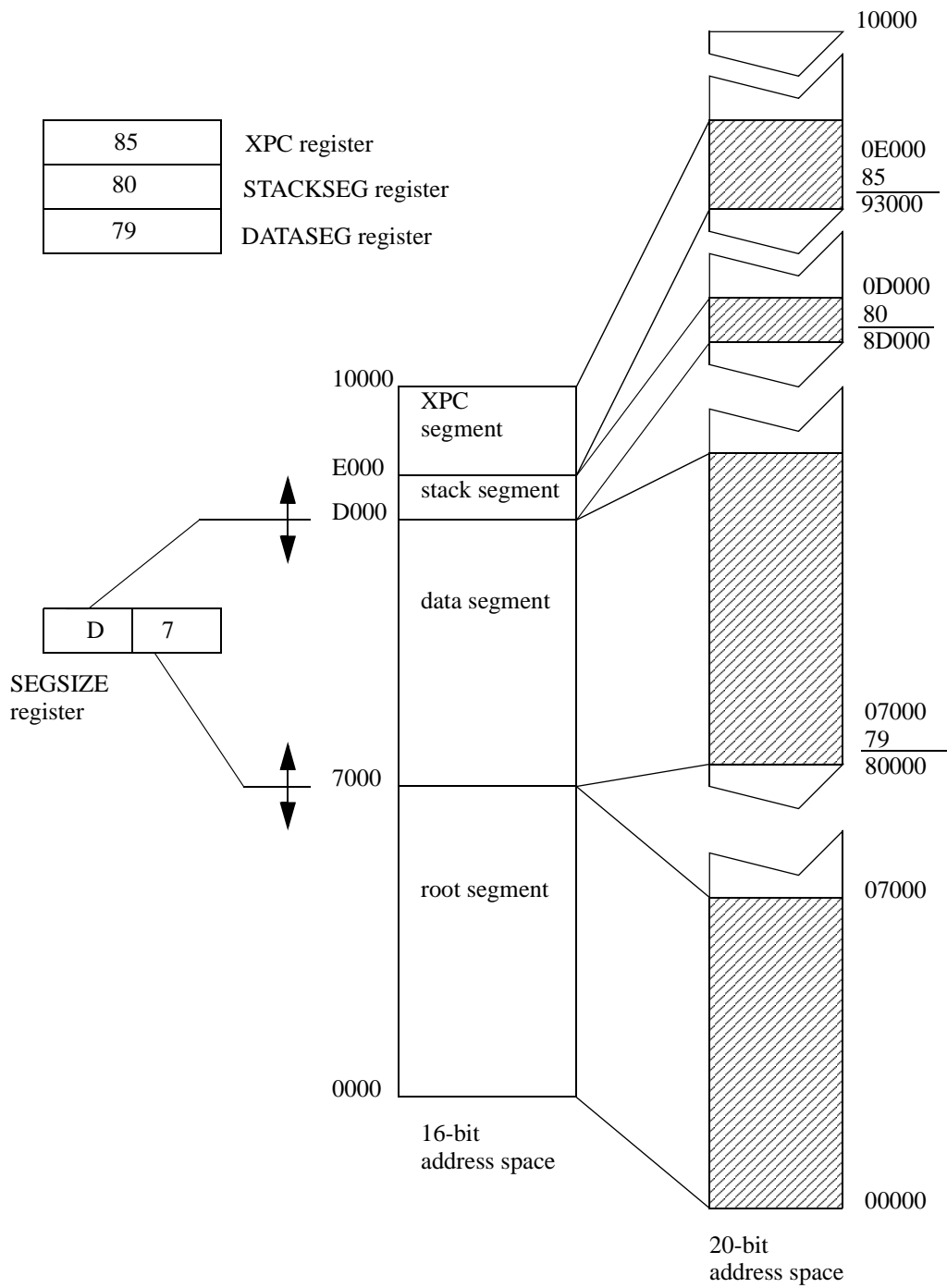
The Rabbit memory-mapping unit is similar to, but more powerful than, the Z180 memory-mapping unit. Figure 6 illustrates the relationship among the major components related to addressing memory.



**Figure 6. Addressing Memory Components**

The memory-mapping unit receives 16-bit addresses as input and outputs 20-bit addresses. The processor (except for certain ldp instructions) sees only a 16-bit address space. That is, it sees 65536 distinctly addressable bytes that its instructions can manipulate. Three segment registers are used to map this 16-bit space into a 1-megabyte space. The 16-bit space is divided into four separate zones. Each zone, except the first or root zone, has a segment register that is added to the 16-bit address within the zone to create a 20-bit address. The segment register has eight bits and those eight bits are added to the upper four bits of the 16-bit address, creating a 20-bit address. Thus, each separate zone in the 16-bit memory becomes a window to a segment of memory in the 20-bit address space. The relative size of the four segments in the 16-bit space is controlled by the SEGSIZE register. This is an 8-bit register that contains two 4-bit registers. This controls the boundary between the first and the second segment and the boundary between the second and the third segment. The location of the two movable segment boundaries is determined by a 4-bit value that specifies the upper four bits of the address where the boundary is located. These relationships are illustrated in Figure 7.

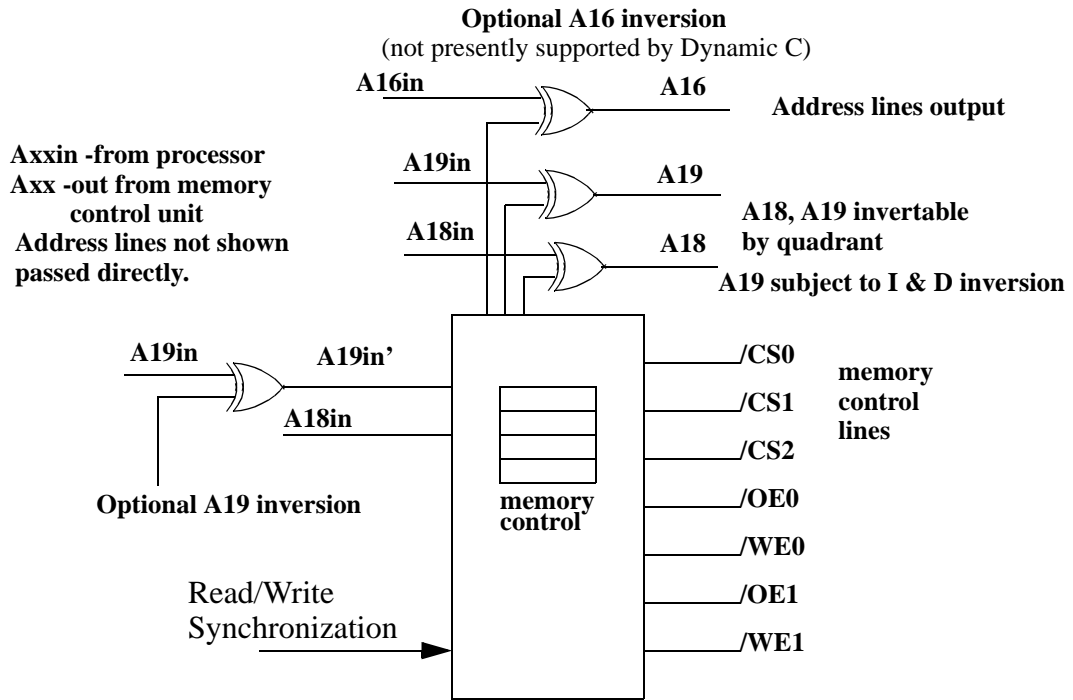
The names given to the segments in the figure are evocative of the common uses for each segment. The *root segment* is mapped to the base of flash memory and contains the startup code as well as other code that may happen to be stored there. The *data segment* usage varies depending on the overall strategy for setting up memory. It may be an extension of the root segment or it may contain data variables. The *stack segment* is normally 4K long and it holds the system stack. The *XPC segment* is normally used to execute code that is not stored in the root segment or the data segment. Special instructions support executing code that is visible in the XPC segment.



**Figure 7. Example of Memory Mapping Operation**

The memory interface unit receives the 20-bit addresses generated by the memory-mapping unit. The memory interface unit conditionally modifies address lines A16, A18 and A19. The other address lines of the 20-bit address are passed unconditionally. The memory interface unit provides control signals for external memory chips. These interface signals are chip selects (/CS0, /CS1, /CS2), output enables (/OE0, /OE1), and write enables (/WE0, /WE1). These signals correspond to the normal control lines found on static memory chips (chip select or /CS, output enable or /OE, and write enable or /WE). In order to generate these memory control signals, the 20-bit address space is divided into four quadrants of 256K each. A *bank control register* for each quadrant determines which of the chip selects and which pair of output enables, and write enables (if any) is enabled when a memory read or write to that quadrant takes place. For example, if a 512K x 8 flash memory is to be accessed in the first 512K of the 20-bit address space, then /CS0, /WE0, /OE0 could be enabled in both quadrants.

Figure 8 shows a memory interface unit.



**Figure 8. Memory Interface Unit**

### 3.2.1 Extended Code Space

A crucial element of the Rabbit memory mapping scheme is the ability to execute programs containing up to a megabyte of code in an efficient manner. This ability is absent in a pure 16-bit address processor, and it is poorly supported by the Z180 through its memory mapping unit. On paged processors, such as the 8086, this capability is provided by paging the code space so that the code is stored in many separate pages. On the 8086 the page size is 64K, so all the code within a given page is accessible using 16-bit addressing for jumps, calls and returns. When paging is used, a separate register (CS on the 8086) is used to determine where the active page currently resides in the total memory space. Special instructions make it possible to jump, call or return from one page to another. These special instructions are called long calls, long jumps and long returns to distinguish them from the same operations that only operate on 16-bit variables.

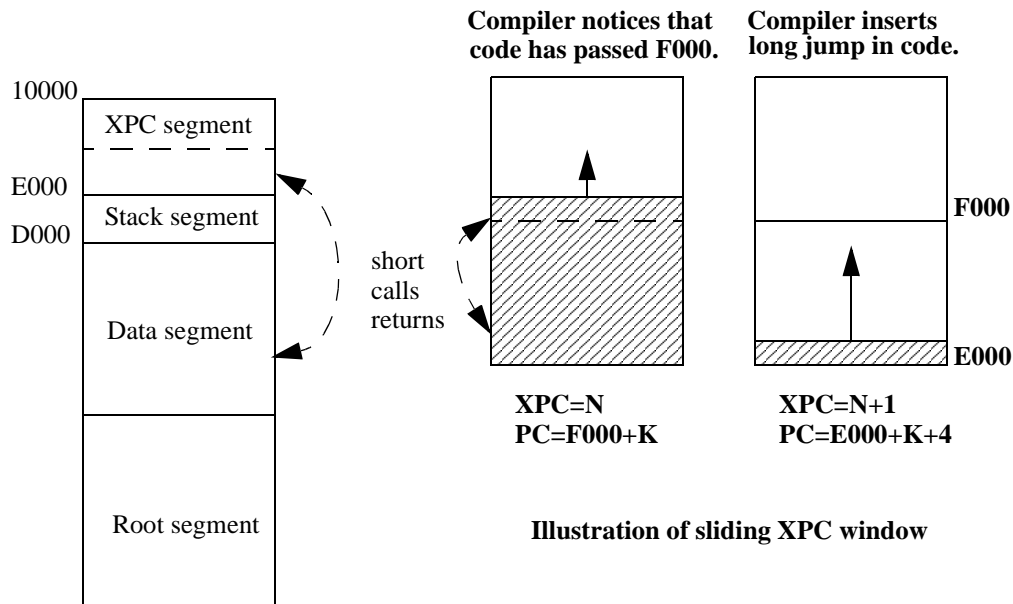
The Rabbit also uses a paging scheme to expand the code space beyond the reach of a 16-bit address. The Rabbit paging scheme uses the concept of a sliding page, which is 8K long. This is the XPC segment. The 8-bit XPC register serves as a page register to specify the part of memory where the window points. When a program is executed in the XPC segment, normal 16-bit jumps, calls and returns are used for most jumps within the window. Normal 16-bit jumps, calls and returns may also be used to access code in the other three segments in the 16-bit address space. If a transfer of control to code outside the window is required, then a long jump, long call or long return is used. These instructions modify both the program counter (PC) and the XPC register, causing the XPC window to point to a different part of memory where the target of the long jump, call or return is located. The XPC segment is always 8K long. The granularity with which the XPC segment can be positioned in memory is 4K. Because the window can be slid by one-half of its size, it is possible to compile continuously without unused gaps in memory.

As the compiler generates code resident in the XPC window, the window is slid down by 4K when the code goes beyond F000. This is accomplished by a long jump that repositions the window 4K lower. This is illustrated by Figure 9 on page 25. The compiler is not presented with a sharp boundray at the end of the page because the window does not run out of space when code passes F000 unless 4K more of code is added before the window is slid down. All code compiled for the XPC window has a 24-bit address consisting of the 8-bit XPC and the 16-bit address. Short jumps and calls can be used, provided that the source and target instructions both have the same XPC address. Generally this means that each instruction belongs to a window that is approximately 4K long and has a 16-bit address between E000+n and F000+m, where n and m are on the order of a few dozen bytes, but can be up to 4096 bytes in length. Since the window is limited to no more than 8K, the compiler is unable to compile a single expression that requires more than 8K or so of code space. This is not a practical consideration since expressions longer than a few hundred bytes are in the nature of stunts rather than practical programs.

Program code can reside in the root segment or the XPC segment. Program code may also be resident in the data segment. Code can be executed in the stack segment, but this is usually restricted to special situations. Code in the root, meaning any of the segments other than the XPC segment, can call other code in the root using short jumps and calls.



Code in the XPC segment can also call code in the root using short jumps and calls. However, a long call must be used when code in the XPC segment is called. Functions located in the root have an efficiency advantage because a long call and a long return require 32 clocks to execute, but a short call and a short return require only 20 clocks to execute. The difference is small, but significant for short subroutines.



**Figure 9. Use of XPC Segment**

### 3.2.2 Practical Memory Considerations

The simplest Rabbit configurations have one flash memory chip interfaced using /CS0 and one RAM memory chip interfaced using /CS1. The smallest practical amount of flash is 128K and the smallest practical amount of RAM is 32K. Smaller chips could be supported, but such small static memories are obsolete parts, so no support is offered.

Although the Rabbit can support code size approaching a megabyte, it is anticipated that the great majority of applications will use less than 250K of code, equivalent to approximately 10,000–20,000 C statements. This reflects both the compact nature of Rabbit code and the typical size of embedded applications.

Directly accessible C variables are limited to approximately 44K of memory, split between data stored in flash and RAM. This will be more than adequate for many embedded applications. Some applications may require large data arrays or tables that will require additional data memory. For this purpose Dynamic C supports a type of extended data memory that allows the use of additional data memory, even extending far beyond a megabyte.

Requirements for stack memory depend on the type of application and particularly whether preemptive multitasking is used. If preemptive multitasking is used, then each task requires its own stack. Since the stack has its own segment in 16-bit address space, it is easy to use available RAM memory to support a large number of stacks. When a preemptive change of context takes place, the STACKSEG register can be changed to map the stack segment to the portion of RAM memory that contains the stack associated with the new task that is to be run. Normally the stack segment is 4K, which is typically large enough to provide space for several (typically four) stacks. It is possible to enlarge the stack segment if stacks larger than 4K are needed. If only one stack is needed, then it is possible to eliminate the stack segment entirely and place the single stack in the data segment. This option is attractive for systems with only 32K of RAM that don't need multiple stacks.

### 3.3 Instruction Set Outline

“Instructions to Load Immediate Data To a Register” on page 27

“Instructions to Load or Store Data from or to a Constant Address” on page 27

“Instructions to Load or Store Data Using an Index Register” on page 28

“Register to Register Move Instructions” on page 29

“Register Exchanges” on page 29

“Push and Pop Instructions” on page 30

“16-bit Arithmetic and Logical Operations” on page 30

“Input/Output Instructions” on page 33

In the discussion that follows, we give a few example instructions in each general category and contrast the Z80/ Z180 with the Rabbit. For a detailed description of every instruction, See “Rabbit Instructions” on page 149.

The Rabbit executes instructions in fewer clocks than the Z80 or Z180. The Z180 usually requires a minimum of four clocks for 1-byte opcodes or three clocks for each byte for multi-byte op codes. In addition, three clocks are required for each data byte read or written. Many instructions in the Z180 require a substantial number of additional clocks. The Rabbit usually requires two clocks for each byte of the op code and for each data byte read. Three clocks are needed for each data byte written. One additional clock is required if a memory address needs to be computed or an index register is used for addressing. Only a few instructions don't follow this pattern. An example is *mul*, a 16 x 16 bit signed two's complement multiply. *mul* is a 1-byte op code, but requires 12 clocks to execute. Compared to the Z180, not only does the Rabbit require fewer clocks, but in a typical situation it has a higher clock speed and its instructions are more powerful.

The most important instruction set improvements in the Rabbit over the Z180 are in the following areas.

- Fetching and storing data, especially 16-bit words, relative to the stack pointer or the index registers IX, IY, and HL.
- 16-bit arithmetic and logical operations, including 16-bit and's, or's, shifts and 16-bit multiply.

- Communication between the regular and alternate registers and between the index registers and the regular registers is greatly facilitated by new instructions. In the Z180 the alternate register set is difficult to use, while in the Rabbit it is well integrated with the regular register set.
- Long calls, long returns and long jumps facilitate the use of 1M of code space. This removes the need in the Z180 to utilize inefficient memory banking schemes for larger programs that exceed 64K of code.
- Input/output instructions are now accomplished by normal memory access instructions prefixed by an op code byte to indicate access to an I/O space. There are two I/O spaces, internal peripherals and external I/O devices.

Some Z80 and Z180 instructions have been deleted and are not supported by the Rabbit (see Section 19, “Differences Rabbit vs. Z80/Z180 Instructions,” on page 159). Most of the deleted instructions are obsolete or are little-used instructions that can be emulated by several Rabbit instructions. It was necessary to remove some instructions to free up 1-byte op codes needed to implement new instructions efficiently. The instructions were not re-implemented as 2-byte op codes so as not to waste on-chip resources on unimportant instructions. Except for the instruction `ex (sp),hl`, the original Z180 binary encoding of op codes is retained for all Z180 instructions that are retained.

### 3.3.1 Instructions to Load Immediate Data To a Register

A constant that follows the op code in the instruction stream can generally be loaded to any register, except PC, AF, IP and F. (Load to the PC is a jump instruction.) This includes the alternate registers on the Rabbit, but not on the Z180. Some example instructions appear below.

```
ld a,3
ld hl,456
ld bc',3567 ; not possible on Z180
ld h',4Ah ; not possible on Z180
ld ix,1234
ld c,54
```

Byte loads require four clocks, word loads require six clocks. Loads to IX, IY or the alternate registers generally require two extra clocks because the op code has a 1-byte prefix.

### 3.3.2 Instructions to Load or Store Data from or to a Constant Address

```
ld a,(mn) ; loads 8 bits from address mn
ld a',(mn) ; not possible on Z180
ld (mn),a
ld hl,(mn);load 16 bits from the address specified by mn
ld hl',(mn) ; to alternate register, not possible Z180
ld (mn),hl
```

Similar 16-bit loads and stores exist for DE, BC, SP, IX and IY.

It is possible to load data to the alternate registers, but it is not possible to store and alternate register directly to memory.

```
ld a',(mn) ; allowed
** ld (mn),a' ; **** not a legal instruction!
** ld (mn),de' ;**** not a legal instruction!
```

### 3.3.3 Instructions to Load or Store Data Using an Index Register

An index register is a 16-bit register, usually IX, IY, SP or HL, that is used for the address of a byte or word to be fetched from or stored to memory. Sometimes an 8-bit offset is added to the address either as a signed or unsigned number. The 8-bit offset is a byte in the instruction word. BC and DE can serve as index registers only for the special cases below.

```
ld a,(bc)
ld a',(bc)
ld (bc),a
ld a,(de)
ld a',(de)
ld (de),a
```

Other 8-bit loads and stores are the following.

```
ld r,(hl) ; r is any of 7 registers A, B, C, D, E, H, L
ld r',(hl) ; same but alternate register destination
ld (hl),r ; r is any of the 7 registers above
;or an immediate data byte
** ld (hl),r' ;**** not a legal instruction!
ld r,(ix+d) ; r is any of 7 registers, d is -128 to +127 offset
ld r',(ix+d) ; same but alternate destination
ld (ix+d),r ; r is any of 7 registers or an immediate data byte
ld (iy+d),r ; ix or iy can have offset d
```

The following are 16-bit indexed loads and stores. None of these instructions exists on the Z180 or Z80. The only source for a store is hl. The only destination for a load is hl or hl'.

```
ld hl,(sp+d) ; d is an offset from 0 to 255.
; 16-bits are fetched to hl or hl'
ld (sp+d),hl ; corresponding store
ld hl,(hl+d) ; d is an offset from -128 to +127,
; uses original hl value for addressing
; l=(hl+d), h=(hl+d+1)
ld hl',(hl+d)
ld (hl+d),hl
ld (ix+d),hl ; store hl at address pointed to
; by ix plus -128 to +127 offset
ld hl,(ix+d)
ld hl',(ix+d)
ld (iy+d),hl ; store hl at address pointed to
; by iy plus -128 to +127 offset
ld hl,(iy+d)
ld hl',(iy+d)
```

### 3.3.4 Register to Register Move Instructions

Any of the 8-bit registers, A, B, C, D, E, H, and L, can be moved to any other 8-bit register, for example:

```
ld a,c
ld d,b
ld e,l
```

The alternate 8-bit registers can be a destination, for example:

```
ld a',c
ld d',b
```

These instructions are unique to the Rabbit and require 2 bytes and four clocks because of the required prefix byte. Instructions such as ld a,d' or ld d',e' are not allowed.

Several 16-bit register-to-register move instructions are available. Except as noted, these instructions all require 2 bytes and four clocks. The instructions are listed below.

```
ld dd',bc    ; where dd' is any of hl', de', bc' (2 bytes, 4 clocks)
ld dd',de
ld ix,hl
ld iy,hl
ld hl,iy
ld hl,ix
ld sp,hl    ; 1-byte, 2 clocks
ld sp,ix
ld sp,iy
```

Other 16-bit register moves can be constructed by using 2-byte moves.

### 3.3.5 Register Exchanges

Exchange instructions are very powerful because two (or more) moves are accomplished with one instruction. The following register exchange instructions are implemented.

```
ex af,af'    ; exchange af with af'
exx          ; exchange hl, de, bc with hl', de', bc'
ex de,hl     ; exchange de and hl
```

The following instructions are unique to the Rabbit.

```
ex de',hl    ; 1 byte, 2 clocks
ex de, hl'   ; 2 bytes, 4 clocks
ex de', hl'  ; 2 bytes, 4 clocks
```

The following special instructions (Rabbit and Z180/Z80) exchange the 16-bit word on the top of the stack with the HL register. These three instructions are each 2 bytes and 15 clocks.

```
ex (sp),hl
ex (sp),ix
ex (sp),iy
```

### 3.3.6 Push and Pop Instructions

There are instructions to push and pop the 16-bit registers AF, HL, DC, BC, IX, and IY. The registers AF', HL', DE', and BC' can be popped. Popping the alternate registers is exclusive to the Rabbit, and is not allowed on the Z80 / Z180.

Examples

```
pop hl
push bc
push ix
push af
pop de
pop de'
pop hl'
```

### 3.3.7 16-bit Arithmetic and Logical Operations

The HL register is the primary 16-bit accumulator. IX and IY can serve as alternate accumulators for many 16-bit operations. The Z180/Z80 has a weak set of 16-bit operations, and as a practical matter the programmer has to resort to combinations of 8-bit operations in order to perform many 16-bit operations. The Rabbit has many new op codes for 16-bit operations, removing some of this weakness.

The basic Z80/Z180 16-bit arithmetic instructions are

```
add hl,ww    ; where ww is HL, DE, BC, SP
adc hl,ww    ; add and add carry
sbc hl,ww    ; sub and sub carry
inc ww       ; increment the register (without affecting flags)
```

In the above op codes, IX or IY can be substituted for HL. The add and adc instructions can be used to left-shift HL with the carry. An alternate destination prefix (altd) may be used on the above instructions. This causes the result and its flags to be stored in the corresponding alternate register. If the altd flag is used when IX or IY is the destination register, then only the flags are stored in the alternate flag register.

The following new instructions have been added for the Rabbit.

```
;Shifts
rr hl ; rotate hl right with carry, 1 byte, 2 clocks
; note use adc hl,hl for left rotate, or add hl,hl if
; no carry in is needed.
rr de ; 1 byte, 2 clocks
rl de ; rotate de left with carry, 1-byte, 2 clocks
rr ix ; rotate ix right with carry, 2 bytes, 4 clocks
rr iy ; rotate iy right with carry

;Logical Operations
and hl,de ; 1 byte, 2 clocks
and ix,de ; 2 bytes, 4 clocks
and iy,de
or hl,de ; 1 byte, 2 clocks
or ix,de ; 2 bytes, 4 clocks
or iy,de
```

The *bool* instruction is a special instruction designed to help test the HL register. *bool* sets HL to the value 1 if HL is non zero, otherwise, if HL is zero its value is not changed. The flags are set according to the result. *bool* can also operate on IX and IY.

```
bool hl ; set hl to 1 if non- zero, set flags to match hl
bool ix
bool iy
altd bool hl ; set hl' an f' according to hl
altd bool iy ; modify iy and set f' with flags of result
```

The *sbc* instruction can be used in conjunction with the *bool* instruction for performing comparisons. The *sbc* instruction subtracts one register from another and also subtracts the carry bit. The carry out is inverted compared to the carry that would be expected if the number subtracted was negated and added. The following examples illustrate the use of the *sbc* and *bool* instructions.

```
; Test if hl>=de - hl and de unsigned numbers 0-65535
or a ; clear carry
sbc hl,de ; if C==0 then hl>=de else if C==1 then hl<de

; convert the carry bit into a boolean variable in hl
;
sbc hl,hl ; sets hl==0 if C==0, sets hl==0ffffh if C==1
bool hl ; hl==1 if C was set, otherwise hl==0
;
; convert not carry bit into boolean variable in hl
sbc hl,hl ; hl==0 if C==0 else hl==ffff if C=1
inc hl ; hl==1 if C==0 else hl==0 if C==1
; note carry flag set, but zero / sign flags reversed
```

In order to compare signed numbers using the *sbc* instruction, the programmer can map the numbers into an equivalent set of unsigned numbers by inverting the sign bit of each number before performing the comparison. This maps the most negative number 08000h to the smallest unsigned number 0000h, and the most positive signed number 07FFFh to the largest unsigned number 0FFFFh. Once the numbers have been converted, the com-

parison can be done as for unsigned numbers. This procedure is faster than using a jump tree that requires testing the sign and overflow bits.

```

; example - test for hl>=de where hl and de are signed numbers
; invert sign bits on both
add hl,hl ; shift left
ccf ; invert carry
rr hl ; rotate right
rl de
ccf
rr de ; invert de sign
sbc hl,de ; no carry if hl>=de
; generate boolean variable true if hl>=de
sbc hl,hl ; zero if no carry else -1
inc hl ; 1 if no carry, else zero
bool ; use this instruction to set flags if needed

```

The *sbc* instruction can also be used to perform a sign extension.

```

; extend sign of l to hl
ld a,l
rla ; sign to carry
sbc a,a ; a is all 1's if sign negative
ld h,a ; sign extended

```

The multiply instruction performs a signed multiply that generates a 32-bit signed result.

```

mul ; signed multiply of bc and de,
;result in hl:bc - 1 byte, 12 clocks

```

If a 16-bit by 16-bit multiply with a 16-bit result is performed, then only the low part of the 32-bit result (bc) is used. This (counter intuitively) is the correct answer whether the terms are signed or unsigned integers. The following method can be used to perform a 16 x 16 bit multiply of two unsigned integers and get an unsigned 32-bit result. This uses the fact that if a negative number is multiplied the sign causes the other multiplier to be subtracted from the product. The method shown below adds double the number subtracted so that the effect is reversed and the sign bit is treated as a positive bit that causes an addition.

```

ld bc,n1
ld hl',bc ; save bc in hl'
ld de,n2
ld a,b ; save sign of bc
mul ; form product in hl:bc
or a ; test sign of bc multiplier
jr p,x1 ; if plus continue
add hl,de ; adjust for negative sign in bc
x1:
rl de ; test sign of de
jr nc,x2 ; if not negative
; subtract other multiplier from hl
ex de,hl'
add hl,de
x2:
; final unsigned 32 bit result in hl:bc

```



This method can be modified to multiply a signed number by an unsigned number. In that case only the unsigned number has to be tested to see if the sign is 0, n and in that case the signed number is added to the upper part of the product.

The multiply instruction can also be used to perform left or right shifts. A left shift of n positions can be accomplished by multiplying by the unsigned number  $2^n$ . This works for  $n \leq 15$ , and it doesn't matter if the numbers are signed or unsigned. In order to do a right shift by n ( $0 < n < 16$ ), the number should be multiplied by the unsigned number  $2^{(16-n)}$ , and the upper part of the product taken. If the number is signed, then a signed by unsigned multiply must be performed. If the number is unsigned or is to be treated as unsigned for a logical right shift, then an unsigned by unsigned multiply must be performed. The problem can be simplified by excluding the case where the multiplier is  $2^{15}$ .

### 3.3.8 Input/Output Instructions

The Rabbit uses an entirely different scheme for accessing input/output devices. Any memory access instruction may be prefixed by one of two prefixes, one for internal I/O space and one for external I/O space. When so prefixed, the memory instruction is turned into an I/O instruction that accesses that I/O space at the I/O address specified by the 16-bit memory address used. For example

```
ioi ld a,(85h)    ; loads A register with contents
                  ; of internal I/O register at location 85h.

ld iy,4000h
ioe ld hl,(iy+5) ; get word from external I/O location 4005h
```

By using the prefix approach, all the 16-bit memory access instructions are available for reading and writing I/O locations. The memory mapping is bypassed when I/O operations are executed.

I/O writes to the internal I/O registers require only two clocks, rather than the minimum of three clocks required for writes to memory or external I/O devices.

## 3.4 How to Do It in Assembly Language—Tips and Tricks

### 3.4.1 Zero HL in 4 Clocks

```
bool hl ; 2 clocks, clears carry, hl is 1 or 0
rr hl   ; 2 clocks, 4 total - get rid of possible 1
```

This sequence requires four clocks compared to six clocks for `ld hl,0`.

### 3.4.2 Exchanges Not Directly Implemented

HL<->HL' - eight clocks

```
ex de',hl ; 2 clocks
ex de',hl' ; 4 clocks
ex de',hl ; 2 clocks, 8 total
```

DE<->DE' - six clocks

```
ex de',hl ; 2 clocks
ex de,hl ; 2 clocks
ex de',hl ; 2 clocks, 6 total
```

BC<->BC' - 12 clocks

```
ex de',hl ; 2 clocks
ex de,hl' ; 4
ex de,hl ; 2
exx ; 2
ex de,hl ; 2
```

Move between IX, IY and DE, DE'

IX/IY->DE / DE->IX/IY

```
;ix, ix --> de
ex de,hl
ld hl,ix/iy / ld ix/iy,hl
ex de,hl ; 8 clocks total

; de --> ix/ iy
ex de,hl
ld ix/iy,hl
ex de,hl ; 8 clocks total
```

### 3.4.3 Manipulation of Boolean Variables

Logical operations involving HL when HL is a logical variable with a value of 1 or 0—this is important for the C language where the least bit of a 16-bit integer is used to represent a logical result

Logical not operator—invert bit 0 of HL in four clocks (also works for IX, IY in eight clocks)

```
dec hl ; 1 goes to zero, zero goes to -1
bool hl ; -1 to 1, zero to zero. 4 clocks total
```

Logical xor operator—xor hl,de when HL/DE are 1 or 0.

```
add hl,de
res 1,1 ; 6 clocks total, clear bit 1 result of if 1+1=2
```

### 3.4.4 Comparisons of Integers

Unsigned integers may be compared by testing the zero and carry flags after a subtract operation. The zero flag is set if the numbers are equal. With the SBC instruction the carry cleared is set if the number subtracted is less than or equal to the number it is subtracted from. 8-bit unsigned integers span the range 0–255. 16-bit unsigned integers span the range 0–65535.

```
or a ; clear carry
sbc hl,de ; hl=A and de=B
```

```
A>=B    !C
A<B     C
A==B    Z
A>B     C & !Z
A<=B    C v Z
```

If A is in hl and B is in de these operations can be performed as follows assuming that the object is to set hl to 1 or 0 depending on whether the compare is true or false.

```
; compute hl<de
; unsigned integers
; ex de,hl ; uncomment for de<hl
or a ; clear carry
sbc hl,de ; C set if hl<de
sbc hl,hl ; hl-hl-C -- -1 if carry set
bool hl ; set to 1 if carry, else zero
; else result == 0
;unsigned integers
; compute hl>=de or de>=hl - check for !C
; ex de,hl ; uncomment for de<=hl
or a ; clear carry
sbc hl,de ; !C if HL>=DE
sbc hl,hl ; hl-hl-C - zero if no carry, -1 if C
inc hl ; 14 / 16 clocks total -if C after first sbc result 1,
; else 0
; 0 if C , 1 if !C
;
: compute hl==de
or a ; clear carry
sbc hl,de ; zero is equal
bool hl ; force to zero, 1
dec hl ; invert logic
bool hl ; 12 clocks total -logical not, 1 for inputs equal
;
```

Some simplifications are possible if one of the unsigned numbers being compared is a constant. Note that the carry has a reverse sense from sbc.

```

;test for hl>B B is constant
ld de,(65535-B)
add hl,de ; carry set if hl>B
sbc hl,hl ; hl-hl-C - result -1 if carry set, else zero
bool hl ; 14 total clocks - true if hl>B

; hl>=B B is constant not zero
ld de,(65536-B)
add hl,de
sbc hl,hl
bool hl ; 14 clocks

; hl>=B and B is zero
ld hl,1 ; 6 clocks

; hl<B B is a constant, not zero (if B==0 always false)
ld de,(65536-B)
add hl,de ; not carry if hl<B
sbc hl,hl ; -1 if carry, else 0
inc hl ; 14 clocks --0 if carry, else 1 if no carry
;
; hl <= B B is constant not zero
ld de,(65535-B)
add hl,de ; ~C if hl<=B
ccf ; C if true
sbc hl,hl ; if C -1 else 0
inc hl ; 16 clocks -- 1 if true, else 0
;
; hl <= B B is zero - true if hl==0
bool hl ; result in hl
;
; hl==B and B is a constant not zero
ld de,(65536-B)
add hl,de ; zero if equal
bool hl
inc hl
res 1,1 ; 16 clocks

; hl==B and B==0
bool hl
inc hl
res 1,1 ; 8 clocks

```

For signed integers the conventional method to look at the zero flag, the minus flag and the overflow flag. Signed 8-bit integers span the range -128 to +127 (80h to 7Fh). Signed 16-bit integers span the range -32768 to + 32767 (8000h to 7FFFh). The sign and zero flag tell which is the larger number after the subtraction unless the overflow is set, in which case the sign flag needs to be inverted in the logic, that is, it is wrong.

```

A>B    (!S & !V & !Z) v (S & V)
A<B    (S & !V) v (!S & V & !Z)
A==B
A>=B
A<=B

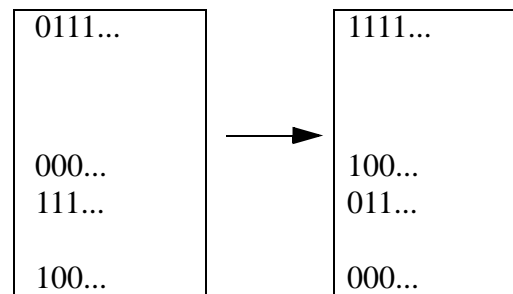
```

Another method of doing signed compare is to first map the signed integers onto unsigned integers by inverting bit 15. This is shown in Figure 10 on page 37. Once the mapping has been performed by inverting bit 15 on both numbers, the comparisons can be done as if the numbers were unsigned integers. This avoids having to construct a jump tree to test the overflow and sign flags. An example is shown below.

```

; test hl>5 for signed integers
ld de,65535-(5+08000h) ; 5 mapped to unsigned integers
ld bc,08000h
add hl,bc ; invert high bit
add hl,de ; 16 clocks to here
; carry now set if hl>5 - opportunity to jump on carry
subc hl,hl ; hl-hl-C ; if C on result is -1, else zero
bool hl ; 22 clocks total - true if hl>5 else false

```



**Figure 10. Mapping Signed Integers to Unsigned Integers by Inverting Bit 15**

### 3.4.5 Atomic Moves from Memory to I/O Space

To avoid disabling interrupts while copying a shadow register to its target register, it is desirable to have an atomic move from memory to I/O space. This can be done using LDD or LDI instructions.

```

ld hl,sh_PDDDR ; point to shadow register
ld de,PDDDR ; set de to point to I/O reg
set 5,(hl) ; set bit 5 of shadow register
; use ldd instruction for atomic transfer
ioi ldd ; (io de)<-(hl) hl--, de--

```

When the LDD instruction is prefixed with an I/O prefix, the destination becomes the I/O address specified by DE. The decrementing of HL and DE is a side effect. If the repeating instructions LDIR and LDDR are used, interrupts can take place between successive iterations. Word stores to I/O space can be used to set two I/O registers at adjacent addresses with a single noninterruptable instruction.

## 3.5 Interrupt Structure

When an interrupt occurs on the Rabbit, the return address is pushed on the stack, and control is transferred to the address of the interrupt service routine. The address of the interrupt service routine has two parts: the upper byte of the address comes from a special register and the lower byte is fixed by hardware for each interrupt. There are separate registers for internal interrupts (IIR) and external interrupts (EIR) to specify the high byte of the interrupt service routine address. These registers are accessed by special instructions.

```
ld a,iir
ld iir,a
ld a,eir
ld eir,a
```

Interrupts are initiated by hardware devices or by certain 1-byte instructions called reset instructions.

```
rst 10
rst 18
rst 20
rst 28
rst 38
```

The RST instructions are similar to those on the Z80 and Z180, but certain ones have been removed from the instruction set (00, 08, 30). The RST interrupts are not inhibited regardless of the processor priority. The user is advised to exercise caution when using these instructions as they are mostly reserved for the use of Dynamic C for debugging. Unlike the Z80 or Z180, the IIR register contributes the upper byte of the service routine address for RST interrupts.

Since interrupt routines do not affect the XPC, interrupt routines must be located in the root code space. However, they can jump to the extended code space after saving the XPC on the stack.

### 3.5.1 Interrupt Priority

The Z80 and Z180 have two levels of interrupt priority: maskable and nonmaskable. The nonmaskable interrupt cannot be disabled and has a fixed interrupt service routine address of 66h. The Rabbit, in contrast, has three levels of interrupt priority and four priority levels at which the processor can operate. If an interrupt is requested, and the priority of the interrupt is higher than that of the processor, the interrupt will take place after the execution of the current instruction is complete (except for privileged instructions)

Multiple interrupt priorities have been established to make it feasible for the embedded systems programmer to have extremely fast interrupts available. *Interrupt latency* refers to the time required for an interrupt to take place after it has been requested. Generally, interrupts of the same priority are disabled when an interrupt service routine is entered. Sometimes interrupts must stay disabled until the interrupt service routine is completed, other times the interrupts can be re-enabled once the interrupt service routine has at least

disabled its own cause of interrupt. In any case, if several interrupt routines are operating at the same priority, this introduces interrupt latency while the next routine is waiting for the previous routine to allow more interrupts to take place. If a number of devices have interrupt service routines, and all interrupts are of the same priority, then pending interrupts can not take place until at least the interrupt service routine in progress is finished, or at least until it changes the interrupt priority. As a rule of thumb, Z-World usually suggests that 100  $\mu$ s be allowed for interrupt latency on Z180-based controllers. This can result if, for example, there are five active interrupt routines, and each turns off the interrupts for at most 20  $\mu$ s.

The intention in the Rabbit is that most interrupting devices will use priority 1 level interrupts. Devices that need extremely fast response to interrupts will use priority level 2 or 3 interrupts. Since code that runs at priority level 0 or 1 never disables level 2 and level 3 interrupts, these interrupts will take place within about 20 clocks, the length of the longest instruction or longest sensible sequence of privileged instructions followed by an unprivileged instruction. It is important that the user be careful not to overdisable interrupts in critical code sections. *The processor priority should not be raised above level 1 except in carefully considered situations.*

The effect of the processor priority on interrupts is shown in Table 2. The priority of the interrupt is usually established by bits in an I/O control register associated with the hardware that creates the interrupt. The 8-bit interrupt register (IR) holds the processor priority in the least significant 2 bits. When an interrupt takes place the IR register is shifted left 2 positions and the lower 2 bits are set to equal the priority of the interrupt that just took place. This means that an interrupt service can only be interrupted by an interrupt service routine for an interrupt of higher priority (unless the priority is explicitly set lower by the programmer). The IR register serves as a 4-word stack to save and restore interrupt priority. It can be shifted right, restoring the previous priority by a special instruction (ipres). Since only the current processor priority and 3 previous priorities can be saved in IP instructions are also provided to push and pop IP from using the regular stack. A new priority can be pushed into the IP register with special instructions (IP 0, IP 1, IP 2, IP 3).

**Table 2. Effect of Processor Priorities on Interrupts**

Processor Priority	Effect on interrupts
0	All interrupts, priority 1,2 and 3 take place after execution of current non privileged instruction.
1	Only interrupts of priority 2 and 3 take place.
2	Only interrupts of priority 3 take place.
3	All interrupt are suppressed (except RST instruction).

### 3.5.2 Multiple External Interrupting Devices

The Rabbit has two distinct external interrupt request lines. If there are more than two external causes of interrupts, then these lines must be shared between multiple devices. The interrupt line is edge sensitive, meaning that it requests an interrupt only when a rising or falling edge, whichever is specified in the setup registers, takes place. The state of the interrupt line(s) can always be read by reading parallel port E since they share pins with parallel port E.

If several lines are to share interrupts with the same port, the individual interrupt requests would normally be or'ed together so that any device can cause an interrupt. If several devices are requesting an interrupt at the same time, only one interrupt results because there will be only one transition of the interrupt request line. To resolve the situation and make sure that the separate interrupt routines for the different devices are called, a good method is to have a interrupt dispatcher in software that is aided by providing separate attention request lines for each device. The attention request lines are basically the interrupt request lines for the separate devices before they are or'ed together. The interrupt dispatcher calls the interrupt routines for all devices requesting interrupts in priority order so that all interrupts are serviced.

### 3.5.3 Privileged Instructions, Critical Sections and Semaphores

Normally an interrupt happens at the end of the instruction currently executing. However, if the instruction executing is *privileged*, the interrupt cannot take place at the end of the instruction and is deferred until a nonprivileged instruction is executed, usually the next instruction. Privileged instructions are provided as a handy way of making a certain operation *atomic* because there would be a software problem if an interrupt took place after the instruction. Turning off the interrupts explicitly may be too time consuming or not possible because the purpose of the privileged instruction is to manipulate the interrupt controls. For additional information on privileged instructions, see Section 18.19, "Privileged Instructions," on page 157.

The privileged instructions to load the stack are listed below.

```
ld sp,h1
ld sp,iy
ld sp,ix
```

The following instructions to load SP are privileged because they are frequently followed by an instruction to change the stack segment register. If an interrupt occurs between these two instructions and the following instruction, the stack will be ill-defined.

```
ld sp,h1
ioi ld sseg,a
```



The privileged instructions to manipulate the IP register are listed below.

```
ip 0      ; shift ip left and set priority 00 in bits 1,0
ip 1
ip 2
ip 3
ipres     ; rotate ip right 2 bits, restoring previous priority
reti      ; pops IP from stack and then pops return address
pop ip    ; pop IP register from stack
```

### 3.5.4 Critical Sections

Certain library routines may need to disable interrupts during a critical section of code. Generally these routines are only legal to call if the processor priority is either 0 or 1. A priority higher than this implies custom hand-coded assembly routines that do not call general-purpose libraries. The following code can be used to disable priority 1 interrupts.

```
ip 1      ; save previous priority and set priority to 1
....critical section...
ipres     ; restore previous priority
```

This code is safe if it is known that the code in the critical section does not have an embedded critical section. If this code is nested, there is the danger of overflowing the IP register. A different version that can be nested is the following.

```
push ip
ip 1      ; save previous priority and set priority to 1
....critical section...
pop ip    ; restore previous priority
```

The following instructions are also privileged.

```
ld a,xpc
ld xpc,a
bit b,(hl)
```

### 3.5.5 Semaphores Using Bit B,(HL)

The bit B,(HL) instruction is privileged to allow the construction of a semaphore by the following code.

```
bit b,(hl) ; test a bit in the byte at (hl)
set b,(hl) ; make sure bit set, does not affect flag
; if zero flag set the semaphore belongs to us;
; otherwise someone else has it
```

A semaphore is used to gain control of a resource that can only belong to one task or program at a time. This is done by testing a bit to see if it is on, in which case someone else is using the resource, otherwise setting the bit to indicate ownership of the resource. No interrupt can be allowed between the test of the bit and the setting of the bit as this might allow two different program to both think they own the resource.

### 3.5.6 Computed Long Calls and Jumps

The instruction to set the XPC is privileged so that a computed long call or jump can be made. This would be done by the following sequence.

```
ld xpc,a
jp (hl)
```

In this case, A has the new XPC, and HL has the new PC. This code should normally be executed in the root segment so as not to pull the memory out from under the JP (HL) instruction.

A call to a computed address can be performed by the following code.

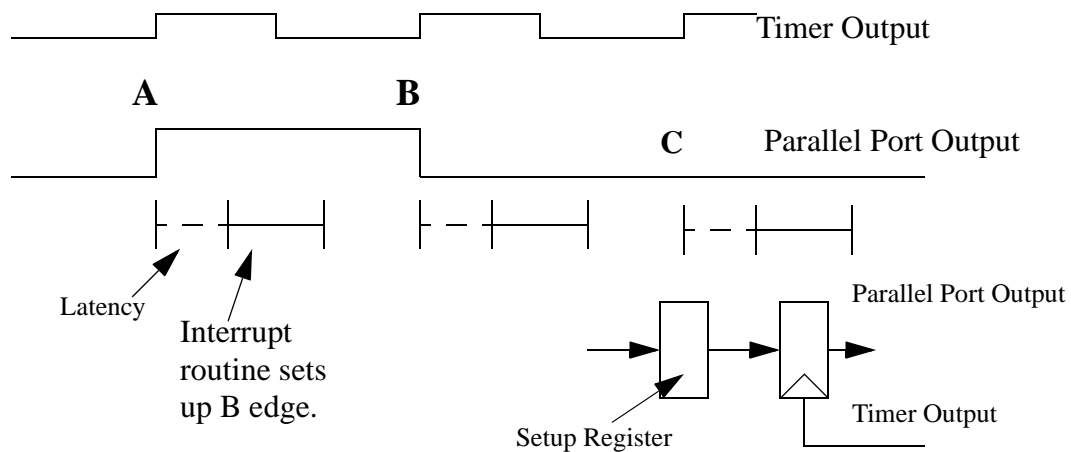
```
; A=xpc, IY=address
;
ld a,newxpc
ld iy,newaddress
lcall docall ; call utility routine in the root
;
; The docall routine
docall:
ld xpc,a ;set xpc
jp (iy) ; go to the routine
```

## 4. Rabbit Capabilities

This section describes the various capabilities of the Rabbit that may not be obvious from the technical description.

### 4.1 Precisely Timed Output Pulses

The Rabbit can output precise pulses under software control. The effect of interrupt latency is avoided because the interrupt always prepares a future pulse edge that is clocked into the output registers on the next clock. This is shown in Figure 11.



**Figure 11. Timed Output Pulses**

The timer output in Figure 11 is periodic. As long as the interrupt routine can be completed during one timer period, an arbitrary pattern of synchronous pulses can be output from the parallel port.

The interrupt latency depends on the priority of the interrupt and the amount of time that other interrupt routines of the same or higher priority inhibit interrupts. The first instruction of the interrupt routine will start executing within 30 clocks of the interrupt request for the highest priority interrupt routine. This includes 19 clocks for the longest instruction to complete execution and 10 clocks for the interrupt to execute. Pushing registers requires 10–12 clocks per 16-bit register. Popping registers requires 7–9 clocks. Return from interrupt requires 7 clocks. If three registers are saved and restored, and 20 instructions averaging 5 clocks are executed, an entire interrupt routine will require about 200 clocks, or 10  $\mu$ s with a 20 MHz clock. Given this timing, the following capabilities become possible.

Pulse width modulated output—The minimum pulse width is 10  $\mu$ s. If the repetition rate is 10 ms, then a new pulse with 1000 different widths can be generated at the rate of 100 times per second.

Asynchronous communications serial output—Asynchronous output data can be generated with a new pulse every 10  $\mu$ s. This corresponds to a baud rate of 100,000 bps.

Asynchronous communications serial input—To capture asynchronous serial input, the input must be polled faster than the baud rate, a minimum of three times faster, with five times being better. If five times polling is used, then asynchronous input at 20,000 bps could be received.

Generating pulses with precise timing relationships—The relationship between two events can be controlled to within 10  $\mu$ s to 20  $\mu$ s.

Using a timer to generate a periodic clock allows events to be controlled to a precision of approximately 10  $\mu$ s. However, if Timer B is used to control the output registers, a precision approximately 100 times better can be achieved. This is because Timer B has a match register that can be programmed to generate a pulse at a specified future time. The match register has two cascaded registers, the match register and the next match register. The match register is loaded with the contents of the next match register when a pulse is generated. This allows events to be very close together, one count of Timer B. Timer B can be clocked by `sysclk/2` divided by a number in the range of 1–256. Timer B can count as fast as 10 MHz with a 20 MHz system clock, allowing events to be separated by as little as 100 ns. Timer B and the match registers have 10 bits.

Using Timer B, output pulses can be positioned to an accuracy of `clk/2`. Timer B can also be used to capture the time at which an external event takes place in conjunction with the external interrupt line. The interrupt line can be programmed to interrupt on either rising, falling or both edges. To capture the time of the edge, the interrupt routine can read the Timer B counter. The execution time of the interrupt routine up to the point where the timer is read can be subtracted from the timer value. If no other interrupt is of the same or higher priority, then the uncertainty in the position of the edge is reduced to the variable time of the interrupt latency, or about one-half the execution time of the longest instruction. This uncertainty is approximately 10 clocks, or 0.5  $\mu$ s for a 20 MHz clock. This enables pulse width measurements for pulses of any length, with a precision of about 1  $\mu$ s. If multiple pulses need to be measured simultaneously, then the precision will be reduced, but this reduction can be minimized by careful programming.

#### 4.1.1 Pulse Width Modulation to Reduce Relay Power

Typically relays need far less current to hold them closed than is needed to initially close them. For example, if the driver is switched to a 75% duty cycle using pulse width modulation after the initial period when the relay armature is picked, the holding current will be approximately 75% of the full duty-cycle current and the power consumption will be about 56% as great.

The pulse width modulation rate may be from 5 kHz to 20 kHz. If a periodic interrupt is established that interrupts every 50  $\mu$ s, then a 50% duty cycle could be set up for a 100  $\mu$ s period. A 25%, 50% or 75% duty cycle could operate on a 200  $\mu$ s period. A 250  $\mu$ s pe-

riod would allow duty cycles of 20%, 40%, 60% or 80%. The code for such an interrupt routine might appear as follows.

```

push af ; 10
push hl
push de
ld hl,(ptr) ; 11 get pointer to location in array
ld a,(maskand) ; 9 get mask
and a,(hl) ; 5 get current output
ld e,a ; 2
ld a,(maskor) ; 9
or a,e ; 2
ioi ld (port),a ; 13 store in port
inc hl ; 2 point to next
ld a,(hl) ; 5 check for end of array
or a,a ; 2
jr nz,step2 ; 2
ld hl,(beginptr) ; 11 reset hl to start of array
step2:
ld (ptr),hl ; 13 save hl
pop de ;7
pop hl
pop af
reti ; 7 return from interrupt

; 153 clocks total worst case - 7.5 us at 20 MHz

```

This routine would take approximately 15% of the processor's compute time assuming 50  $\mu$ s between interrupts. This routine could be speeded up, but at the expense becoming more complicated. Instead of "and" and "or" masks, a higher level routine could modify the array directly, and the end of the array could be detected by testing a bit pattern in HL. The higher level routine would have to suppress the interrupt while changing the bit pattern in the array, or otherwise prevent erratic outputs while the array is being changed. If the relay emits a whistle at the period of the modulation, the acoustic energy can be spread out over the spectrum by periodically missing an "off" pulse, creating a phase shift of 180°. A faster routine that executes in two-thirds the time is shown below.

```

push af ;10
push hl
ld hl,(ptr) ;11
ld a,(hl) ;5
ioi ld (port),a ; 13 output data
inc hl
ld a,0fh ;4
and l ; see if hl at end of cycle
jr z,step2
ld (ptr),hl
pop hl
pop af
reti
step2:
ld a,(beginptr)
ld l,a

```

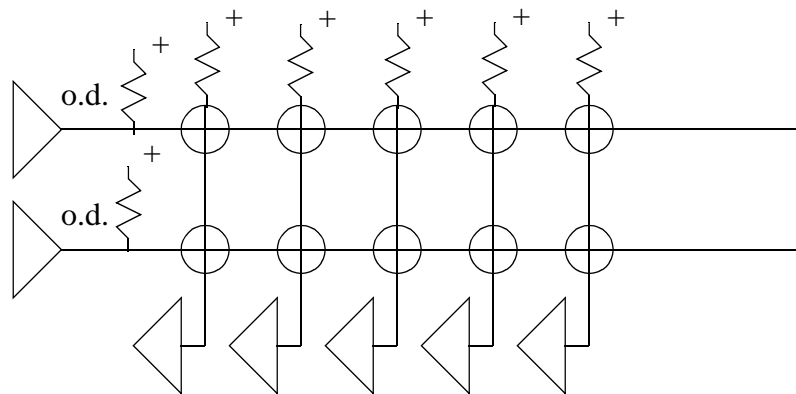
```

ld (ptr),hl ;13
pop hl ;7
pop af
reti
; 103 clocks total

```

## 4.2 Open-Drain Outputs Used for Key Scan

The parallel port D outputs can be individually programmed to be open drain. This is useful for scanning a switch matrix, as shown in Figure 12. A row is driven low, then the columns are scanned for a low input line, which indicates a key is closed. This is repeated for each row. The advantage of using open-drain outputs is that if two keys in the same column are depressed, there will not be a fight between a driver driving the line high and another driver driving it low.



**Figure 12. Using Open-Drain Outputs for Key Scan**

## 4.3 Cold Boot

Most microprocessors start executing at a fixed address, often address zero, after a reset or power-on condition. The Rabbit has two mode pins (SMODE0, SMODE1—see Figure 13 on page 49). The logic state of these two pins determines the startup procedure after a reset. If both pins are grounded, then the Rabbit starts executing instructions at address zero. On reset, address zero is defined to be the start of the memory connected to the memory control lines /CS0, and /OE0. However, three other startup modes are available. These alternate methods all involve accepting a data stream via a communications port that is used to store a boot program in a RAM memory, which in turn can be used to start any further secondary boot process, such as downloading a program over the same communications port. (For a detailed description, see Section 7.9 on page 79.)

Three communication channels may be used for the bootstrap, either serial port A in asynchronous mode at 2400 bps, serial port A in synchronous mode with an external clock, or the (parallel) slave port.

The cold-boot protocol accepts groups of three bytes that define an address and a data byte. Each triplet causes a write of the data byte to either memory or to internal I/O space. The high bit of the address is set to specify the I/O space, and thus writes are limited to the first 32K of either space. The cold boot is terminated by a store to an address in I/O space, which causes execution to begin at address zero. Since any memory chip can be remapped to address zero by storing in the I/O space, RAM can be temporarily be mapped to zero to avoid having to deal with the more complicated write protocol of flash memory, which is the usual default memory located at address zero.

The following are the advantages of the cold-boot capability.

- Flash memory can be soldered to the microprocessor board and programmed via a serial port or a parallel port. This avoids having to socket the part or program it with a BIOS or boot program before soldering.
- Complete reprogramming of the flash memory can be accomplished in the field. This is particularly useful during software development when the development platform can perform a complete reload of software regardless of the state of the existing software in the processor. The standard programming cable for Dynamic C allows the development platform to reset and cold boot the target, a Rabbit-based microprocessor board.
- If the Rabbit is used as a slave processor, the master processor can cold boot it over via the slave port. This means the slave can operate without any nonvolatile memory. Only RAM is required.

## 4.4 The Slave Port

The slave port allows a Rabbit to act as a slave to another processor, which can also be a Rabbit. The slave has to have only a processor chip, a RAM chip, and clock and reset signals that can be supplied by the master. The master can cold boot and download a program to the slave. The master does not have to be a Rabbit processor, but can be any type of processor capable of reading and writing standard registers.

For a detailed description, See “Rabbit Slave Port” on page 117.

The slave processor’s slave port is connected to the master processor’s data bus. Communication between the master and the slave takes place via three registers, implemented in the Rabbit, for each direction of communication, for a total of six data registers. In addition, there is a slave port status register that can be read by either the master or the slave (see Figure 33 on page 117). Two slave address lines are used by the master to select the register to be read or written. The registers that carry data from the master to the slave appear as write registers to the master and as read registers to the slave. The registers that operate in the opposite direction appear as read registers to the master and as write registers to the slave. These registers appear as read-write registers on both sides, but are not true read-write registers since different data may be read from what is written. The master provides the clock or strobe to store data in the three write registers under its control. The master also can do a write to the status register, which is used as a signaling device and does not actually write to the status register. The three registers that the master can write

appear as read registers to the slave Rabbit. The master provides an enable strobe to read the three read data registers and the status register. These registers are write registers to the Rabbit.

The first register or the three pairs of registers is special in that writing can interrupt the other processor in the master-slave communications link. An output line from the slave is asserted when the slave writes to slave register zero. This line can be used to interrupt the master. Internal circuits in the slave can be setup up to interrupt the slave when the master writes to slave register zero.

The status register that is available to both sides keeps score on all the registers and reports if a potential interrupt is requested by either side. The status register keeps track of the "full-empty" status of each register. A register is considered full when one side of the link writes to it. It becomes empty if the other side reads it. In this way either side can test if the other side has modified a register or whether either side has even stored the same information to a register.

The master-slave communication link makes possible "set and forget" communication protocols. Either side can issue a command or request by storing data in some register and then go about its business while the other side takes care of the request according to its own time schedule. The other side can be alerted by an interrupt that takes place when a store is made to register zero, or it can alert itself by a periodic poll of the status register.

Of the three registers seen by each side for each direction of communication, the first register, slave register zero, has a special function because an interrupt can only be generated by a write to this register, which then causes an interrupt to take place on the other side of the link if the interrupt is enabled. One type of protocol is to store data first in registers 1 and 2, and then as the last step store to register 0. Then 24 bits of data will be available to the interrupt routine on the other side of the link.

Bulk data transfers across the link can take place by an interrupt for each byte transferred, similar to a typical serial port or UART. In this case, a full-duplex transfer can take place, similar to what can be done with a UART. The overhead for such an interrupt-driven transfer will be on the order of 100 clocks per byte transferred, assuming a 20-instruction interrupt routine. (In order to keep the interrupt routine to 20 instructions, the interrupt routine needs to be very focused as opposed to general purpose.) Several methods are available to cater to a faster transfer with less computing overhead. There are enough registers to transfer two bytes on each interrupt, thus nearly halving the overhead. If a rendezvous is arranged between the processors, data can be transferred at approximately 25 clocks per byte. Each side polls the status register waiting for the other side to read/write a data register, which is then written/read again by the other side.

#### **4.4.1 Slave Rabbit As A Protocol UART**

A prime application for the Rabbit used as a slave is to create a 4-port UART that can also handle the details of a communication protocol. The master sends and receives messages over the slave port. Error correction, retransmission, etc., can be handled by the slave.



# 5. Pin Assignments and Functions

## 5.1 Package Schematic and Pin Names

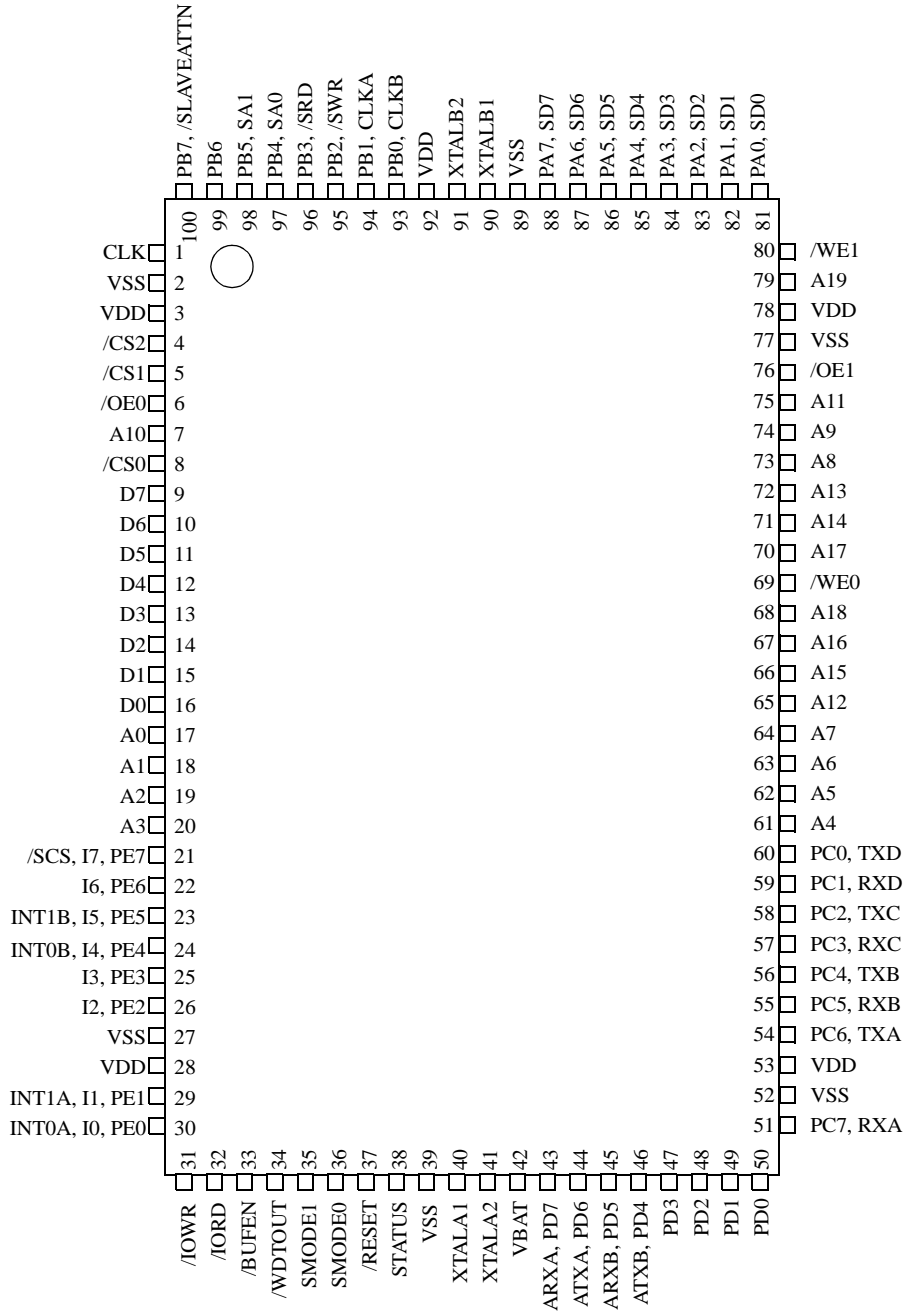
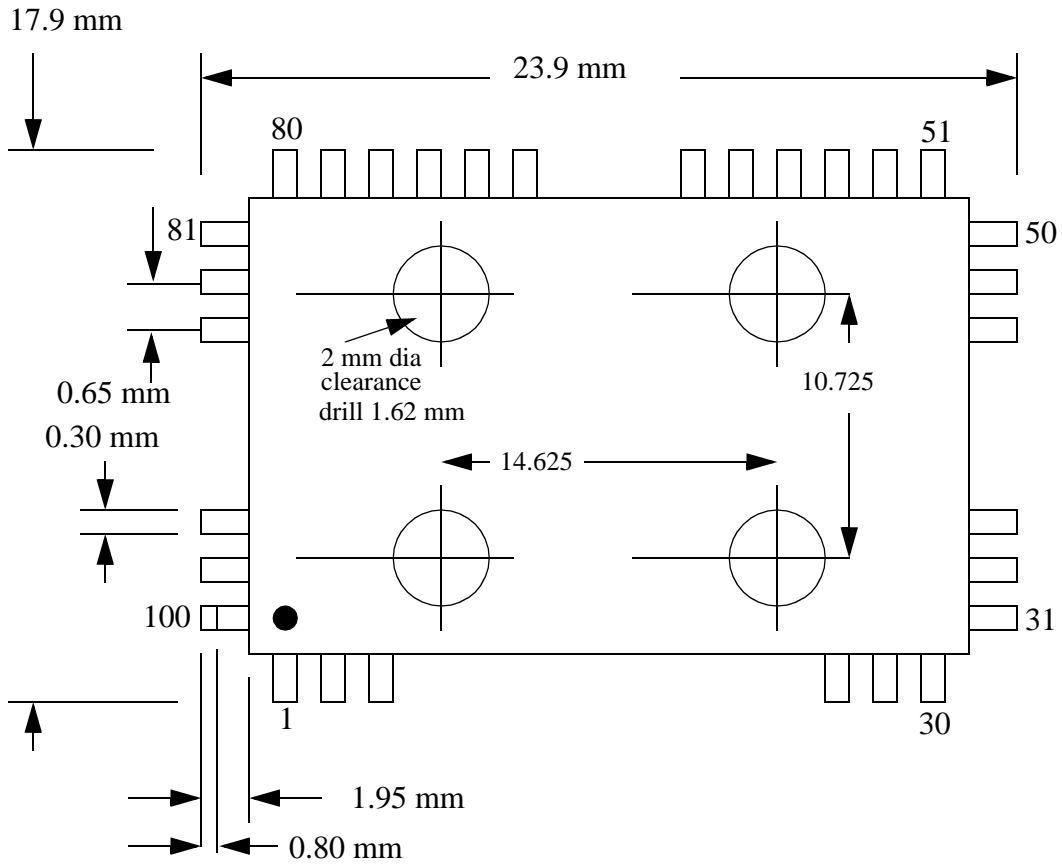


Figure 13. Package Outline and Pin Assignments

## 5.2 Package Mechanical Dimensions

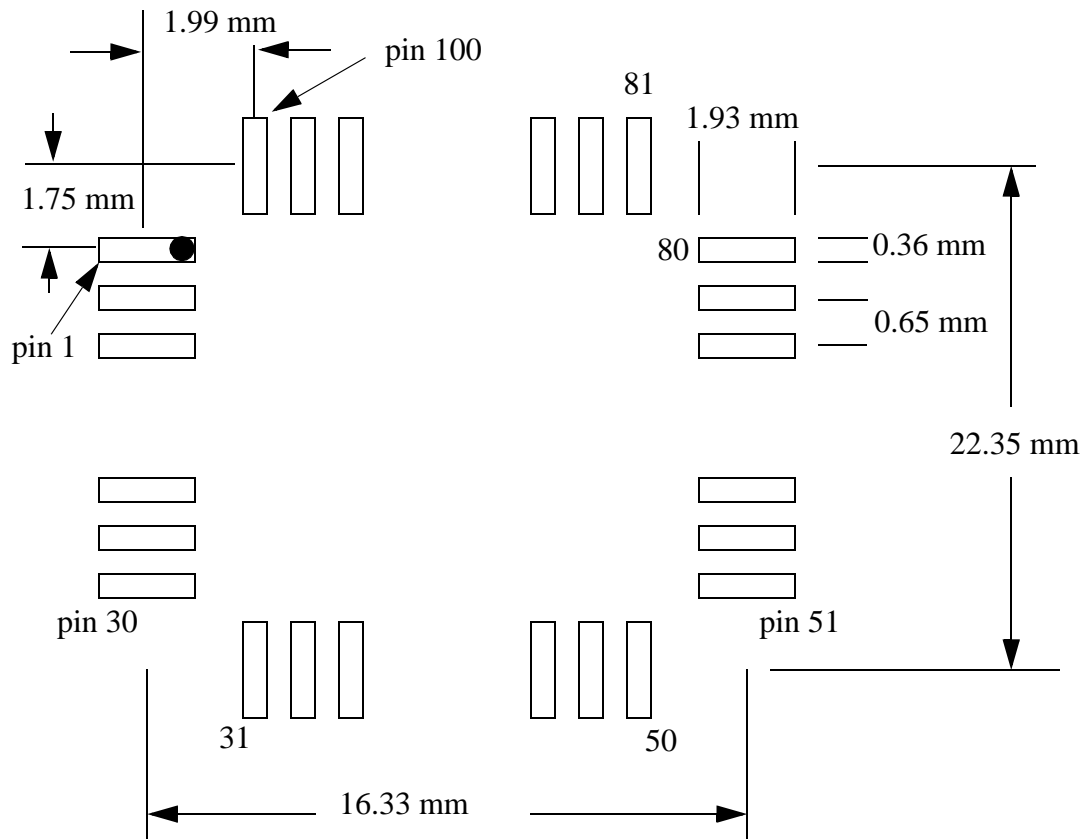
Figure 14 shows the mechanical dimensions of the Rabbit PQFP package.



100-pin PQFP Physical Dimensions (Top View)  
 Holes are optional to aid in attaching substitute connector  
 for processor.

**Figure 14. Mechanical Dimensions Rabbit PQFP Package**

Figure 15 shows the PC board footprint for the Rabbit 100-pin PQFP.



**Figure 15. PC Board Footprint for Rabbit 100-pin PQFP**

### 5.3 Rabbit Pin Descriptions

Table 3 lists all the pins on the device, along with their direction, function, and pin number on the package.

**Table 3. Rabbit Pin Assignments**

Pin Group	Pin Name	Direction	Function	Pin Numbers
Hardware	CLK	Output	Processor clock output. This signal is derived from the main system oscillator and may be divided by 8, or doubled, or both, by internal programmable circuitry. This signal is enabled after reset. Under program control this pin can output the full internal clock frequency, or 1/2 the internal frequency, or it can be used as a general-purpose output pin under software control. See Table 9, “Global Output Control Register (GOOCR = 0Eh),” on page 71.	1
	/RESET	Input	Master reset.	37
	XTALA1	Input	Quartz crystal for 32 kHz clock oscillator. Lines to the crystal should be short and shielded from crosstalk. If an external clock is used, this pin should be driven by the external clock.	40
	XTALA2	Output	Quartz crystal for 32 kHz crystal oscillator. Do not connect if an external clock is used.	41
	XTALB1	Input	Quartz crystal for main system oscillator. Lines to the crystal should be short and shielded from crosstalk. If an external clock is used this pin should be driven by the external clock.	90
	XTALB2	Output	Quartz crystal for main system oscillator. Do not connect if an external clock is used.	91
CPU Buses	A0–A19	Output	Address bus.	7, 17–20, 61–68, 70–75, 79
	D0–D7	Bidirectional	Data bus.	9–16
Status/Control	/WDTOUT	Output	WDT timeout—outputs a pulse when the internal watchdog times out. May also be used to output a 30 μs pulse.	34
Status	STATUS	Output	Programmable for functions: 1. driven low on first opcode fetch cycle 2. driven low during interrupt acknowledge cycle 3. to serve as a general-purpose output. See Table 9, “Global Output Control Register (GOOCR = 0Eh),” on page 71.	38

**Table 3. Rabbit Pin Assignments**

	SMODE1 SMODE0	Input	Startup mode select (SMODE1 = pin 35, SMODE0 = pin 36) to determine bootstrap procedure. (SMODE1 = 0, SMODE0 = 0) start executing at address zero. (0,1) cold boot from slave port. (1,0) cold boot from clocked serial port A. (1,1) cold boot from asynchronous serial port A at 2400 bps. The smode pins can be used as general input pins once the cold boot is complete.	35–36 (1:0)
Chip Selects	/CS0	Output	Memory Chip Select 0—connects directly to static memory chip chip select pin. Normally this pin is used to select base flash memory that holds the program.	8
	/CS1	Output	Memory Chip Select 1—normally this pin is connected directly to static RAM chip select. /CS1 can be optionally forced continuously low under software control, a feature that aids in the use of battery-backed RAM when the chip select must pass through a controller that may have a slow propagation time.	5
	/CS2	Output	Memory Chip Select 2—connect to static memory chip. Use this chip select last.	4
Output Enables	/OE0	Output	Memory Output Enable 0—connect directly to static memory chip.	6
	/OE1	Output	Memory Output Enable 1—alternate memory output enable allows chip selects to be shared between two memory chips.	76
Write Enables	/WE0	Output	Memory Write Enable 0—connect directly to static memory chip. This pin may be disabled under software control to write protect the chip.	69
	/WE1	Output	Memory Write Enable 1—connect directly to static memory chip. This pin may be disabled under software control to write protect the chip.	80
I/O Control	/BUFEN	Output	I/O Buffer Enable—this signal is driven low during an external I/O cycle and may be used to control 3-state enable on the bus buffer. The purpose is to save power by not driving the I/O address or data lines on every bus cycle.	33

**Table 3. Rabbit Pin Assignments**

I/O Read Strobe	/IORD	Output	I/O read strobe. Driven low on an external I/O bus cycle. May be used to drive glue logic concerned with I/O expansion, such as the direction pin on a bidirectional bus buffer. See also programmable strobes in port E.	32
I/O Write Strobe	/IOWR	Output	I/O write strobe. Driven low as a write strobe during external I/O read cycles. See also programmable strobes in port E.	31
I/O Port A	PA0–PA7	Input/Output	These 8 bits serve as general-purpose input output or they serve as the data port for the slave port. On reset these pins are set to inputs and they float.	81–88
I/O Port B	PB0–PB7	4 In/2 Out/ 2 I/O	I/O Port B. When used as parallel I/O, PB7 and PB6 are outputs only. PB0–PB5 are inputs only. PB0 and PB1 can be outputs when set up as the clock for the clocked serial ports. On reset, outputs are set to zero. If the slave port is enabled, the following alternate assignments apply: PB7—/SLAVEATTN: slave requests attention. PB5, PB4—address lines (SA1, SA0) for slave registers. PB3—slave negative write strobe from master. PB2—slave negative read strobe from master. If serial port A is enabled in clocked mode, then PB1 is the bidirectional clock line. If serial port B is enabled in clocked mode, then PB0 is the bidirectional clock line.	93–100
I/O port C	PC0–PC7	4 In/4 Out	I/O Port C. When used as a parallel port Bits 1, 3, 5, 7 are inputs and bits 0, 2, 4, 6 are outputs. Bits 0, 2, 4, 6 can alternately be selectively enabled to serve as the serial data output for serial ports D, C, B, A respectively. Bits 1, 3, 5, 7 serve as the serial data inputs for serial ports D, C, B, A. These inputs can also be read from the parallel port register when they are being used by the serial port UART.	51, 52, 54–60

**Table 3. Rabbit Pin Assignments**

I/O Port D	PD0–PD7	Input/Output/output open drain	I/O Port D. Each bit may be individually selected to be an input or output. Each output may be selected to be high-low drive or open drain. Outputs are buffered by timer-synchronizable registers for precision edge control. PD6 can be programmed to be an optional serial output for serial port A. PD4 can be programmed to be an optional serial output for serial port B. PD7 and PD5 can be used as alternate serial inputs by serial ports A and B, in which case these pins should be programmed as inputs.	43–50
I/O Port E	PE7–PE0	Input/Output	I/O Port E. Each bit may be individually selected to be an input or output. Outputs are buffered by timer-synchronizable registers for precision edge control. Each of the port lines can be individually selected to be an I/O control signal instead of a parallel I/O line. Each of the 8 possible I/O control signals is a strobe energized on an external I/O cycle to 1/8th of the 64K external I/O space. Each strobe can be programmed to be a chip select, a write strobe, a read strobe or a combined read and write strobe. Any port bit used as an I/O control strobe must be programmed as an output bit. If the slave port is enabled, PE7 is used as the slave register chip select signal (negative active). PE7 should be programmed as an input for the slave register chip select function to work. If PE7 is programmed as an output and set low, then the slave register chip select will always be activated. PE0 and PE4 serve as alternate inputs for external interrupt 0. PE1 and PE5 serve as alternate inputs for external interrupt 1. If PE0 is enabled, then PE1 must also be enabled and similarly for PE4 and PE5. The interrupt is triggered programmably on fall, rising or both edges. If both interrupts are enabled, they are or'ed together after edge detection has been performed on each input individually. The port bits must be set up as inputs for the to use them as interrupt request inputs.	21–30
Power	VDD VBAT		+3.3 V or +5.0 V +3.0 V (battery backup), +3.3 V or +5.0 V	3, 28, 42, 53, 78, 92
	VSS		Ground	2, 27, 39, 52, 77, 89

**Table 3. Rabbit Pin Assignments**

Serial Ports	CLKA	Input/Output	Clock for serial port A when operating in synchronous mode. Alternate assignment for PB1.	94
Serial Ports	CLKB	Input/Output	Clock for serial port A when operating in synchronous mode. Alternate assignment for PB0.	93
Serial Ports	RXA, TXA, RXB, TXB, RXC, TXC, RXD, TXD	RX—input TX—output	Serial inputs and output for serial ports A–D. These are alternate pin assignments for parallel port C.	51, 54–60
Serial Ports	ARXA, ATXA, ARXB, ATXB	RX—input TX—output	Alternate serial inputs and output for serial ports A and B. These are alternate pin assignments for parallel port D, PD4–PD7.	43–46
Slave Port	SD0-SD7	Bidirectional	Slave port data bus. An alternate assignment for parallel port A.	81–88
Slave Port	/SLAVEATTN	Output	/SLAVEATTN—Slave is requesting attention from the master. An alternate pin assignment for parallel port B, bit 7.	100
Slave Port	/SRD	Input	Strobe used to read one of the slave registers. An alternate pin assignment for parallel port B, bit 3.	96
Slave Port	/SWR	Input	Strobe used to write a slave register. An alternate pin assignment for parallel port B, bit 2.	95
Slave Port	SA0, SA1	Input	Address lines to address slave registers. An alternate pin assignment for parallel port B, bits 4 and 5.	97,98
Slave Port	/SCS	Input	Chip select for slave port, active low. An alternate pin assignment for parallel port E, bit 7.	21
I/O Strobes	/I0,/I1, /I2, /I3, /I4, /I5, /I6, /I7	Outputs	I/O strobes. Each strobe uses 1/8th of the I/O space or 8K addresses. Each strobe can be programmed as: chip select, read, write, combined read or write. These are alternate pin assignments for parallel port E, bits 0–7. Each pin may be individually re-assigned from parallel port to strobe functionality.	21–26, 29, 30

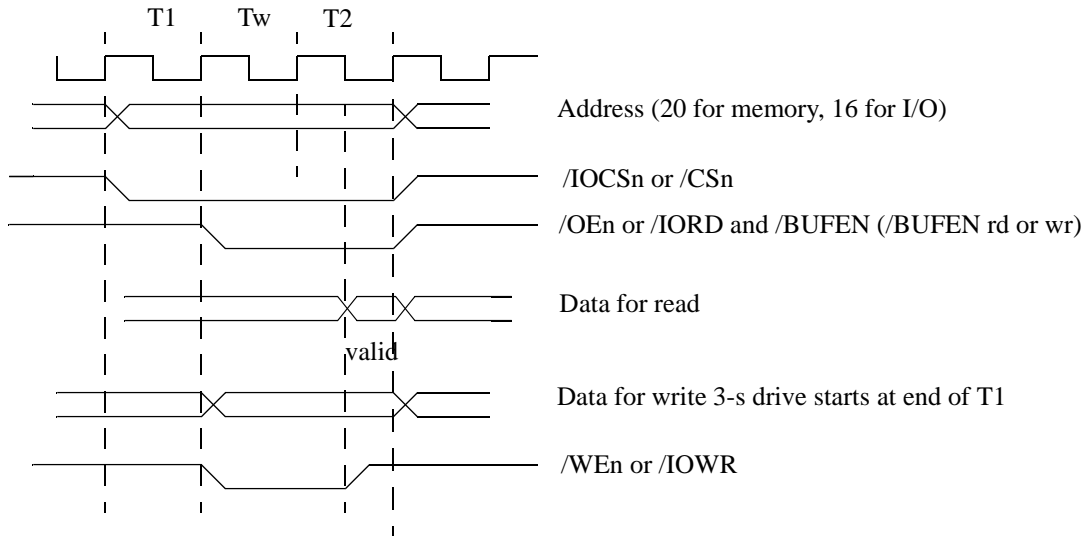


**Table 3. Rabbit Pin Assignments**

External Interrupt 0	INT0A, INT1A	Inputs	These pins are sampled and an interrupt request for external interrupt number 0 is latched on a specified transition (pos, neg, either). There is a separate latch for each pin. May be enabled when this pin is set up as input for parallel port E. The value of the pin may also be read via the parallel port. Uses bits 0, 1 of the parallel port. If parallel port is set up as output, the parallel port output may be used to cause the interrupt.	29, 30
External Interrupt 1	INT0B, INT1B	Inputs	These pins are sampled and an interrupt request for external interrupt number 0 is latched on a specified transition (pos, neg, either). There is a separate latch for each pin. May be enabled when this pin is set up as input for parallel port E. The value of the pin may also be read via the parallel port. Uses bits 4, 5 of the parallel port. If parallel port is set up as output, the parallel port output may be used to cause the interrupt.	23, 24

## 5.4 Bus Timing

The external bus has essentially the same timing for memory cycles or I/O cycles. A memory cycle begins with the chip select and the address lines. One clock later, the output enable is asserted for a read. The output data and the write enable are asserted for a write.



Notes:

Read may have no wait states.

Write cycles and I/O read cycles have at least 1 wait state. Clock may be asymmetric if clock doubler used. I/O chip select available on port E as option.

**Figure 16. Bus Timing Read and Write**

## 5.5 Description of Pins with Alternate Functions

**Table 4. Pins With Alternate Functions**

Pin Name	Output Function	Input Function	Other Function
STATUS (38)	1. Low on first op code fetch. 2. Low on interrupt acknowledge		Programmable output port high/low
SMODE1 (35)		(SMODE1, SMODE2) Startup boot mode control.	1-bit input after boot complete.
SMODE2 (36)		(SMODE1, SMODE2) Startup boot mode control.	1-bit input after boot complete.
CLK (1)	1. Peripheral clock. 2. Peripheral clock/2.		Programmable output port high/low
/WDTOUT (34)	Outputs 30.5 $\mu$ s pulse on watchdog timeout (processor is also reset).		Outputs a pulse between 30.5 and 61 $\mu$ s under program control.
PA7 (88)	SD7	SD7	
PA6 (87)	SD6	SD6	
PA5 (86)	SD5	SD5	
PA4 (85)	SD4	SD4	
PA3 (84)	SD3	SD3	
PA2 (83)	SD2	SD2	
PA1 (82)	SD1	SD1	
PA0 (81)	SD0	SD0	
PB7 (100)	/SLAVEATTN (master needs attention from slave).		
PB5 (98)		SA1 (slave address).	
PB4 (97)		SA0	
PB3 (96)		/SRD (strobe for master to read a slave register).	
PB2 (95)		/SWR (strobe for master to write slave register).	

**Table 4. Pins With Alternate Functions**

PB1 (94)	CLKA (serial port A clocked mode clock, bidirectional).	CLKA	
PB0 (93)	CLKB (bidirectional).	CLKB	
PC7 (51)		RXA	
PC6 (54)	TXA		
PC5 (55)		RXB	
PC4 (56)	TXB		
PC3 (57)		RXC	
PC2 (58)	TXC		
PC1 (59)		RXD	
PC0 (60)	TXD		
PD7 (43)		ARXA	
PD6 (44)	ATXA		
PD5 (45)		ARXB	
PD4 (46)	ATXB		
PD3 (47)			
PD2 (48)			
PD1 (49)			
PD0 (50)			
PE7 (21)	/I7—programmable I/O strobe.	/SCS (slave chip select).	
PE6 (22)	/I6		
PE5 (23)	/I5	INT0 (input).	
PE4 (24)	/I4	INT1 (input).	
PE3 (25)	/I3		
PE2 (27)	/I2		
PE1 (39)	/I1	INT0 (input).	
PE0 (30)	/I0	INT1 (input).	

## 5.6 Register and Interrupt Vector Summary

**Table 5. Register and Interrupt Vector Summary**

On-chip peripheral	I/O address range	ISR starting address
System Management	0000xxxx (0x)	RR00
Memory Management	0001xxxx (1x)	No interrupts
Slave Port	0010xxxx (2x)	RR80
Parallel Port A	0011xxxx (3x)	No interrupts
Parallel Port B	0100xxxx (4x)	No interrupts
Parallel Port C	0101xxxx (5x)	No interrupts
Parallel Port D	0110xxxx (6x)	No interrupts
Parallel Port E	0111xxxx (7x)	No interrupts
External I/O Control	1000xxxx (8x)	No interrupts
External Interrupts	1001xxxx (9x)	II00 (int 0) II10 (int 1)
Timer A	1010xxxx (Ax)	RRA0
Timer B	1011xxxx (Bx)	RRB0
Serial Port A	1100xxxx (Cx)	RRC0
Serial Port B	1101xxxx (Dx)	RRD0
Serial Port C	1110xxxx (Ex)	RRE0
Serial Port D	1111xxxx (Fx)	RRF0
RST 10 instruction	n/a	RR20
RST 18 instruction	n/a	RR30
RST 20 instruction	n/a	RR40
RST 28 instruction	n/a	RR50
RST 38 instruction	n/a	RR70



## 6. Rabbit Internal I/O Registers

**Table 6. Rabbit Internal I/O Registers**

Address	Reset Value	Functionality
GCSR=00h	11000000	Global Control Status Register. Control of clocks, periodic interrupts, and monitoring of watchdog. See Table 7 on page 68.
RTCCR=01h	00000000	Real-Time Clock Control Register. See Section 7.5 on page 71
RTC0R=02h	xxxxxxxx	Real-Time Clock Byte 0 Register.
RTC1R=03h	xxxxxxxx	Real-Time Clock Byte 1 Register.
RTC2R=04h	xxxxxxxx	Real-Time Clock Byte 2 Register.
RTC3R=05h	xxxxxxxx	Real-Time Clock Byte 3 Register.
RTC4R=06h	xxxxxxxx	Real-Time Clock Byte 4 Register.
RTC5R=07h	xxxxxxxx	Real-Time Clock Byte 5 Register.
WDTCR=08h	00000000	Watchdog Timer Control Register. See Section 7.6 on page 73
WDTTR=09h	00000000	Watchdog Timer Test Register.
GOCCR=0Eh	00000x00	Global Output Control Register. See Section 7.4 on page 71.
GCDR=0Fh	xxxxx000	Global Clock Doubler Register.
MMIDR=10h	xxx00000	Memory Management I and D Space Register. Controls I & D space enable and battery switchover support for /CS1.
XPC	00000000	Not an I/O register, but initialized to zero by reset.
STACKSEG=11h (Z180 CBR)	00000000	Stack segment memory pointer. Locates stack segment in physical memory.
DATASEG=12h (Z180 BBR)	00000000	Data segment memory pointer. Locates data segment in physical memory.
SEGSIZE=13h (Z180 CBAR)	11111111	Specifies start of data segment and start of stack segment in 64K memory space.
MB0CR=14h	00000000	Memory Bank 0 Control Register. Controls mapping of first memory quadrant 256K to physical memory chips.
MB1CR=15h	xxxxxxxx	Memory Bank 1 Control Register. Controls mapping of second memory quadrant to physical memory chips.
MB2CR=16h	xxxxxxxx	Memory Bank 2 Control Register. Controls mapping of third memory quadrant to physical memory chips.
MB3CR=17h	xxxxxxxx	Memory Bank 3 Control Register. Controls mapping of fourth memory quadrant to physical memory chips.
SPD0R=20h	xxxxxxxx	Slave Port Register 0. Separate registers for read and write used for slave port communication.
SPD1R=21h	xxxxxxxx	Slave port register 1.

**Table 6. Rabbit Internal I/O Registers**

Address	Reset Value	Functionality
SPD2R=22h	xxxxxxxx	Slave port register 2.
SPSR=023h	00000000	Slave port status register.
SPCR=24h	000x0000	Slave port control register.
PADR=30h	xxxxxxxx	Parallel port A data register. R/W.
PBDR=40h	00xxxxxxxx	Parallel port B data register. R/W.
PCDR=50h	x0x0x0x0	Parallel port C data register.
PCFR=55h	x0x0x0x0	Port C function register.
PDDR=60h	xxxxxxxx	Parallel port D data register. R/W.
PDCR=64h	xx00xx00	Port D control register
PDFR=65h	xxxxxxxx	Port D function register.
PDDCR=66h	xxxxxxxx	Port D drive control register.
PDDDR=67h	00000000	Port D data direction register.
PDB0R=68h	xxxxxxxx	Port D bit 0 register. W
PDB1R=69h	xxxxxxxx	Bit 1.
PDB2R=6Ah	xxxxxxxx	Bit 2.
PDB3R=6Bh	xxxxxxxx	Bit 3.
PDB4R=6Ch	xxxxxxxx	Bit 4.
PDB5R=6Dh	xxxxxxxx	Bit 5.
PDB6R=6Eh	xxxxxxxx	Bit 6.
PDB7R=6Fh	xxxxxxxx	Bit 7.
PEDR=70h	xxxxxxxx	Parallel port E data register. R/W.
PECR=74h	xx00xx00	Port E control register.
PEFR=75h	xxxxxxx	Port E function register.
PEDDR=77h	00000000	Port E data direction register.
PEB0R=78h	xxxxxxx	Port E bit 0 register. W
PEB1R=79h	xxxxxxx	Bit 1.
PEB2R=7Ah	xxxxxxx	Bit 2.
PEB3R=7Bh	xxxxxxx	Bit 3.
PEB4R=7Ch	xxxxxxx	Bit 4.
PEB5R=7Dh	xxxxxxx	Bit 5.
PEB6R=7Eh	xxxxxxx	Bit 6.



**Table 6. Rabbit Internal I/O Registers**

Address	Reset Value	Functionality
PEB7R=7FH	xxxxxxxx	Bit 7
IB0CR=80h	00000xxx	External I/O control bank 0
IB1CR=81h	00000xxx	External I/O control bank 1
IB2CR=82h	00000xxx	External I/O control bank 2
IB3CR=83h	00000xxx	External I/O control bank 3
IB4CR=84h	00000xxx	External I/O control bank 4
IB5CR=85h	00000xxx	External I/O control bank 5
IB6CR=86h	00000xxx	External I/O control bank 6
IB7CR=87h	00000xxx	External I/O control bank 7
I0CR=98h	xx000000	External interrupt 0 control register.
I1CR=99h	xx000000	External interrupt 1 control register.
TACSR=0A0h	0000xx00	Timer A Control/Status Register
TACR=0A2h	xxxxxxxx	Timer A Control Register
TAT1R=0A3h	0000xx00	Timer A1 Time Constant 1 Register
TAT4R=0A9h	xxxxxxxx	Timer A4 Time Constant 4 Register
TAT5R=0ABh	xxxxxxxx	Timer A5 Time Constant 5 Register
TAT6R=0ADh	xxxxxxxx	Timer A6 Time Constant 6 Register
TAT7R=0AFh	xxxxxxxx	Timer A7 Time Constant 7 Register
TBCSR=0B0h	xxxxx000	Timer B Control/Status Register
TBCR=0B1h	xxxxx0000	Timer B Control Register
TBM1R=0B2h	xxxxxxxx	Timer B MSB 1 Reg
TBL1R=0B3h	xxxxxxxx	Timer B LSB 1 Reg
TBM2R=0B4h	xxxxxxxx	Timer B MSB 2 Reg
TBL2R=0B5h	xxxxxxxx	Timer B LSB 2 Reg
TBCMR=0BEh	xxxxxxxx	Timer B Count MSB Reg
TBCLR=0BFh	xxxxxxxx	Timer B Count LSB Reg

**Table 6. Rabbit Internal I/O Registers**

Address	Reset Value	Functionality
SADR=0C0h	xxxxxxxx	Serial port A data register receive/send.
SAAR=0C1h	xxxxxxxx	Serial port A alternate data register (transmit 9th bit)
SASR=0C3h	0xx00000	Serial port A status register.
SACR=0C4h	xx000000	Serial port A control register.
SBDR=0D0h	xxxxxxxx	Serial port B data register receive/send.
SBAR=0D1h	xxxxxxxx	Serial port B alternate data register (transmit 9th bit)
SBSR=0D2h	0xx00000	Serial port B status register.
SBCR=0D3h	xx000000	Serial port B control register.
SCDR=0E0h	xxxxxxxx	Serial port C data register receive/send.
SCAR=0E1h	xxxxxxxx	Serial port C alternate data register (transmit 9th bit)
SCSR=0E2h	0xx00000	Serial port C status register.
SCCR=0E3h	xx00x000	Serial port C control register.
SDDR=0F0h	xxxxxxxx	Serial port D data register receive/send.
SDAR=0F1h	xxxxxxxx	Serial port D alternate data register (transmit 9th bit)
SDSR=0F2h	0xx00000	Serial port D status register.
SDCR=0F3h	xx00x000	Serial port D control register.

## 7. Miscellaneous I/O Functions

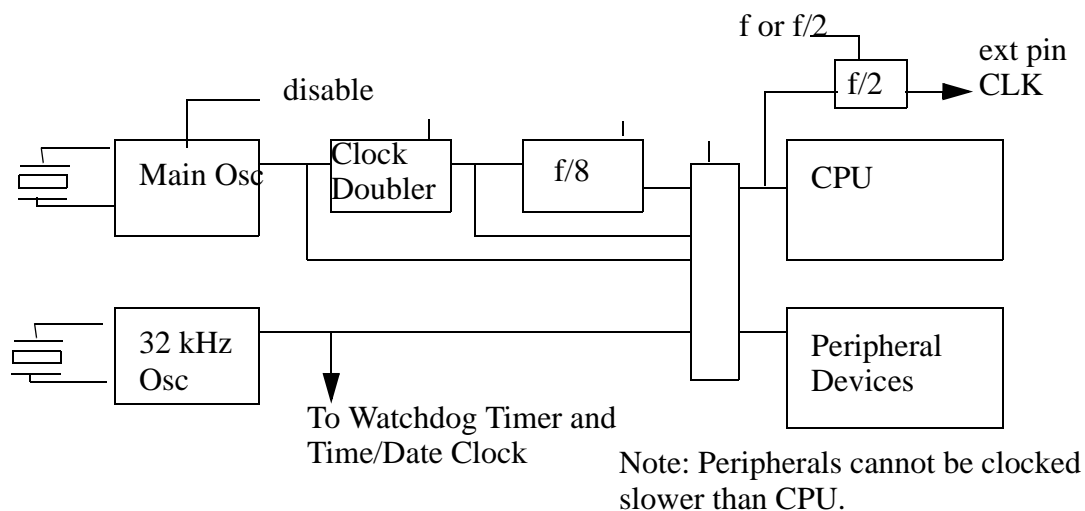
### 7.1 Rabbit Oscillators and Clocks

There are two crystal oscillators built into the Rabbit. The main oscillator accepts crystals up to a frequency of 29.4912 MHz (first overtone crystals only). The clock oscillator requires a 32.768 kHz crystal, and can be battery backed.

An external oscillator or clock can be substituted for either crystal by connecting the external clock to XTALA1 or XTALB1 and leaving the other crystal pin (XTALA2 or XTALB2) unconnected. If an external oscillator is used for the main clock the output pin CLK (pin 1) should be used if the clock is needed externally. This signal is synchronized with the internal clock. In comparison, the internal clock is delayed by approximately 10 nanoseconds compared to the external oscillator input XTALB1.

The main oscillator is normally used to derive the clock for the processor and peripherals. The 32.768 kHz oscillator is normally used to clock the watchdog timer, the battery backable time/date clock, and the periodic interrupt. The main oscillator can be shut down in a special low-power mode of operation, and the 32.768 kHz oscillator is then used to clock all the things normally clocked by the main oscillator. This results in slower execution at low power ( $\sim 200 \mu\text{A}$ ).

The on-chip routing of the clocks is shown in Figure 17. The main oscillator can be doubled in frequency and/or divided by 8. If both doubling and dividing are enabled, then there will be a net division by 4. The CPU clock can optionally be divided by 2 and then optionally drive the external pin CLK. In many cases the clock is not needed externally, and in that case CLK can be used as a general-purpose output pin. The divide by 2 option is available to minimize electromagnetic radiation if the is clock is driven off chip.



**Figure 17. Clock Distribution**

**Table 7. Global Control/Status Register (I/O adr = 00h)**

Bit(s)	Value	Description
7:6  (read only)	00	No reset or watchdog timer timeout since the last read.
	01	The watchdog timer timed out. These bits are cleared by a read of this register.
	10	This bit combination is not possible.
	11	Reset occurred. These bits are cleared by a read of this register.
5 (write only)	0	Read this register to clear periodic interrupt request. This bit always read as zero.
	1	Force a periodic interrupt.
4:2 (write only)	000	Processor clock from the main oscillator, divided by eight. Peripheral clock from the main oscillator, divided by eight.
	001	Processor clock from the main oscillator, divided by eight. Peripheral clock from the main oscillator, without divider.
	01x	Processor clock from the main oscillator, without divider. Peripheral clock from the main oscillator, without divider.
	1x0	Processor clock from the 32 kHz oscillator, without divider. Peripheral clock from the 32KHz oscillator, without divider.
	1x1	Processor clock from the 32 kHz oscillator, without divider. Peripheral clock from the 32 kHz oscillator, without divider. The main oscillator is turned off.
1:0 (write only)	00	Periodic interrupts are disabled.
	01	Periodic interrupts use Interrupt Priority 1.
	10	Periodic interrupts use Interrupt Priority 2.
	11	Periodic interrupts use Interrupt Priority 3.

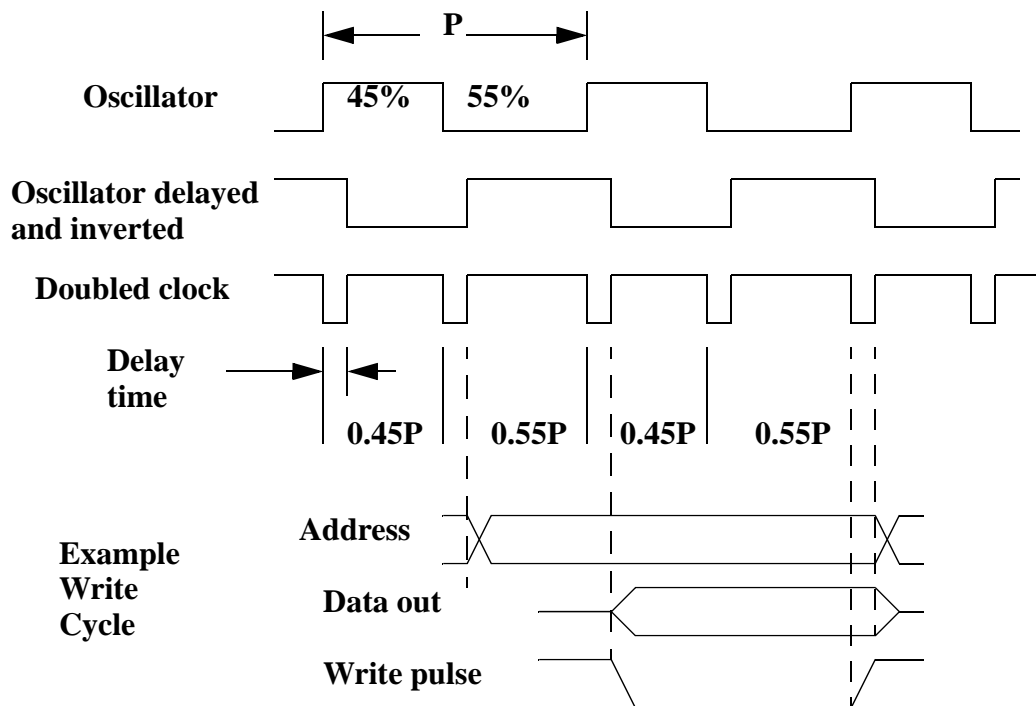
## 7.2 Clock Doubler

The clock doubler is provided to allow a lower frequency crystal to be used for the main oscillator and to provide an added range of clock frequency adjustability. The clock doubler uses an on-chip delay circuit that must be programmed by the user at startup if there is a need to double the clock as shown in Table 8.

**Table 8. Global Clock Double Register (GCDR, adr = 0fh)**

Bit(s)	Value	Description
7:3	xxxxx	These bits are ignored.
2:0	000	The clock double circuit is disabled.
	001	8 ns nominal low time (best for 30 MHz oscillator).
	010	10 ns nominal low time (best for 25 MHz oscillator).
	011	12 ns nominal low time (best for 20 MHz oscillator).
	100	14 ns nominal low time (best for 18 MHz oscillator).
	101	16 ns nominal low time (best for 16 MHz oscillator).
	110	18 ns nominal low time (best for 14 MHz oscillator).
	111	20 ns nominal low time. (best for 12 MHz or slower oscillator).

When the clock doubler is used and there is no subsequent division of the clock, the output clock will be asymmetric, as shown in Figure 18 on page 70. The doubled-clock low time is subject to wide (50%) variation since it depends on process parameters, temperature, and voltage. The times given above are for a supply voltage of 4 V and a temperature of 25°C. These delay values decrease by about 15% when the voltage decreases by 5 V, and increase about 25% when the voltage increases by 3.3 V. The values increase or decrease by 1% for each 5°C increase or decrease in temperature. The doubled clock is created by xor'ing the delayed and inverted clock with itself. If the original clock does not have a 50-50 duty cycle, then alternate clocks will have a slightly different length. Since the duty cycle of the built-in oscillator can be as asymmetric as 55-45, the clock generated by the clock doubler will exhibit up to a 10% variation in period on alternate clocks. This does not affect the no-wait states memory access time since two adjacent clocks are always used. However, the maximum allowed clock speed must be reduced by 10% if the clock is supplied via the clock doubler. The only signals clocked on the falling edge of the clock are the memory and I/O write pulses, and these have noncritical timing. Thus the length of the clock low time is noncritical as long as it is not so long as to shorten the clock high time excessively, which could make the write pulse too short for the memory used. This is unlikely to happen with practical clock speeds and typical static RAM memories.



**Figure 18. Effect of Clock Doubler**

The power consumption is proportional to the clock frequency, and for this reason power can be reduced by slowing the clock when less computing activity is taking place. The clock doubler provides a convenient method of temporarily speeding up or slowing down the clock as part of a power management scheme.

### 7.3 Controlling Power Consumption

The processor power consumption can be traded against speed by slowing the system clock, adding wait states, using low-power-consumption instructions, and for maximum power savings disabling the main system oscillator and using the real-time clock oscillator to provide the clock. The following power saving features can be enabled.

- Add memory wait states for instruction fetching. Total wait states are programmable as 0, 1, 2 or 4. Generally two wait states should use half the power of zero wait states.
- If the clock doubler is not already in use, divide both the processor and the peripheral clock by 4. This is permissible if nothing, particularly timers and serial ports, depends on the peripheral clock.
- If the clock doubler is in use, turn it off, dividing both processor and peripheral by 2.
- Divide the processor and/or peripheral clock by 8.
- Run code in RAM rather than flash memory.
- Switch the processor and peripheral clock to the 32.768 kHz oscillator and, if desired, disable the main oscillator.

- Execute a low-power instruction loop consisting mostly of instructions that don't use much power. The best choice is successive *mul* instructions that multiply 0 x 0. No intervening instructions are needed to load the terms to be multiplied after the first *mul* since all registers involved stay at zero.

It is anticipated that these measures would reduce current consumption to as low as 25  $\mu$ A plus some leakage that would be significant at high operating temperatures.

## 7.4 Output Pins CLK, STATUS, /WDTOUT, /IOBEN

Certain output pins can have alternate assignments as specified in Table 9.

**Table 9. Global Output Control Register (GOCR = 0Eh)**

Bit(s)	Value	Description
7:6	00	CLK pin is driven with processor clock.
	01	CLK pin is driven with processor clock divided by 2.
	10	CLK pin is low.
	11	CLK pin is high.
5:4	00	STATUS pin is active (low) during a first opcode byte fetch.
	01	STATUS pin is active (low) during an interrupt acknowledge.
	10	STATUS pin is low.
	11	STATUS pin is high.
3	1	WDTOUTB pin is low (1 cycle minimum, 2 cycles maximum, of 32 kHz).
	0	WDTOUTB pin follows watchdog function.
2	x	This bit is ignored.
1:0	00	IOBENB pin is active (low) during external I/O cycles.
	01	IOBENB pin is active (low) during data memory accesses.
	10	IOBENB pin is low.
	11	IOBENB pin is high.

## 7.5 Time/Date Clock (Real-Time Clock)

The time/date clock (RTC) is a 48-bit (ripple) counter that is driven by the 32.768 kHz oscillator. The RTC is a modified ripple counter composed of six separate 8-bit counters. The carries are fed into all six 8-bit counters at the same time and then ripple for 8 bits. The time for this ripple to take place is a few nanoseconds per bit and certainly should not should not exceed 200 ns for all 8 bits, even when operating at low voltage.

The 48 bits are enough bits to count up 272 years at the 32 kHz clock frequency. By convention, 12 AM on January 1, 1980, is taken as time zero. Z-World software ignores the highest order bit, giving the counter a capacity of 136 years from January 1, 1980. To read the counter value, the value is first transferred to a 6-byte holding register. Then the individual bytes may be read from the holding registers. To perform the transfer, any data bits are written to RTC0R, the first holding register. The counter may then be read as six 8-bit bytes at RTC0R through RTC5R. The counter and the 32 kHz oscillator are powered from a separate power pin that can be provided with power while the remainder of the chip is powered down. This design makes battery backup possible. Since the processor operates on a different clock than the RTC, there is the possibility of performing a transfer to the holding registers while a carry is taking place, resulting in incorrect information. In order to prevent this, the processor should do the clock read twice and make sure that the value is the same in both reads.

If the processor is itself operating at 32 kHz, the read-clock procedure must be modified since a number of clock counts would take place in the time needed by the slow-clocked processor to read the clock. An appropriate modification would be to ignore the lower bytes and only read the upper 5 bytes, which are counted once every 256 clocks or every 1/128th of a second. If the read cannot be performed in this time, further low-order bits can be ignored.

The RTC registers cannot be set by a write operation, but they can be cleared and counted individually, or by subset. In this manner any register or the entire 48-bit counter can be set to any value with no more than 256 steps. If the 32 kHz crystal is not installed and the input pin is grounded, no counting will take place and the six registers can be used as a small battery-backed memory. Normally this would not be very productive since the circuitry needed to provide the power switchover could also be used to battery-back a regular low-power static RAM.

**Table 10. Real-Time Clock Read Registers**

Real-Time Clock x Holding Register	(RTC0R) R/W	(Address = 00000010)
	(RTC1R)	(Address = 00000011)
	(RTC2R)	(Address = 00000100)
	(RTC3R)	(Address = 00000101)
	(RTC4R)	(Address = 00000110)
	(RTC5R)	(Address = 00000111)

**Table 11. Real-Time Clock RTCxR Data Registers**

Bit(s)	Value	Description
7:0	Read	The current value of the 48-bit RTC holding register is returned.
	Write	Writing to the RTC0R transfers the current count of the RTC to six holding registers while the RTC continues counting.



**Table 12. Real-Time Clock Control Register (RTCCR adr = 01h)**

Bit(s)	Value	Description
7:0	x0xxxxxx	Cancel byte increment mode, disarm reset (except code 80h)
	40h	Arm RTC for a reset with code 80h or reset and byte increment mode with code 0c0h.
	80h	Resets all six byte counters if proceeded by arm command 40h.
	0c0h	Reset all six byte counters and enters byte increment mode—precede this command with 40h arm command.
	010xxxxx	Increment clock(s) register corresponding to bit(s) set to "1". Example: 01001101 increments registers: 0, 2,3. The byte increment mode must be enabled. Storing 00h cancels the byte increment mode.
6	0	Disable the byte increment function by any store with bit 6 set to zero. (Use code 00h.)

## 7.6 Watchdog Timer

The watchdog timer is a 17-bit counter. In normal operation it is driven by the 32 kHz clock. When the watchdog timer reaches any of several values corresponding to a delay of from 0.25 to 2 seconds, it "times out." When it times out, it emits a 1-clock pulse from the watchdog output pin and it resets the processor via an internal circuit. To prevent this timeout, the program must "hit" the watchdog timer before it times out. The hit is accomplished by storing a code in WDTCR.

**Table 13. Watchdog Timer Control Register (WDTCR adr = 08h)**

Bit(s)	Value	Description
7:0	5Ah	Restart (hit) the watchdog timer, with a 2-second timeout period.
	57h	Restart (hit) the watchdog timer, with a 1-second timeout period.
	59h	Restart (hit) the watchdog timer, with a 500 ms timeout period.
	53h	Restart (hit) the watchdog timer, with a 250 ms timeout period.
	other	No effect on watchdog timer.

The watchdog timer may be disabled by storing a special code in the WDTCR register. Normally this should not be done unless an external watchdog device is used. The purpose of the watchdog is to unhang the processor from an endless loop caused by a software crash or a hardware upset.

It is important to use extreme care in writing software to hit the watchdog timer (or to turn off the watchdog timer). The programmer should not sprinkle instructions to hit the watchdog timer throughout his program because such instructions can become part of an endless loop if the program crashes and thus disable the recovery ability given by having a watchdog.

The following is a suggested method for hitting the watchdog. An array of bytes is set up in RAM. Each of these bytes is a virtual watchdog. To hit a virtual watchdog, a number is stored in a byte. Every virtual watchdog is counted down by an interrupt routine driven by a periodic interrupt. This can happen every 10 ms. If none of the virtual watchdogs has counted down to zero, the interrupt routine hits the hardware watchdog. If any have counted down to zero, the interrupt routine disables interrupts, and then enters an endless loop waiting for the reset. Hits of the virtual watchdogs are placed in the user's program at "must exercise" locations.

**Table 14. Watchdog Timer Test Register (WDTTR adr = 09h)**

Bit(s)	Value	Description
7:0	51h	Clock the least significant byte of the WDT timer from the peripheral clock. (Intended for chip test and code 54h below only.)
	52h	Clock the most significant byte of the WDT timer from the peripheral clock. (Intended for chip test and code 54h below only.)
	53h	Clock both bytes of the WDT timer, in parallel, from the peripheral clock. (Intended for chip test and code 54h below only.)
	54h	Disable the WDT timer. This value, by itself, does not disable the WDT timer. Only a sequence of two writes, where the first write is 51h, 52h or 53h, followed by a write of 54h, actually disables the WDT timer. The WDT timer will be re-enabled by any other write to this register.
	other	Normal clocking (32 kHz oscillator) for the WDT timer. This is the condition after reset.

The code to do this may also hit the watchdog with a 0.25-second period to speed up the reset. Such watchdog code must be written so that it is highly unlikely that a crash will incorporate the code and continue to hit the watchdog in an endless loop. The following suggestions will help.

1. Place a jump to self before the entry point of the watchdog hitting routines. This prevents entry other than by a direct call or jump to the routine.
2. Before calling the routine, set a data byte to a special value and then check it in the routine to make sure the call came from the right caller. If not, go into an endless loop with interrupts disabled.
3. Maintain data corruption flags and/or checksums. If these go wrong, go into an endless loop with interrupts off.

## 7.7 System Reset

The Rabbit has a master reset input (/RESET), which initializes everything in the device except for the RTC. This reset is delayed until the completion of any write cycles in progress to prevent any potential corruption of memory. If no write cycles are in progress, the reset takes effect immediately.

The purpose of inhibiting the completion of reset until write cycles in progress are completed is to protect variables in battery-backed memory from corruption when a reset takes place. However, if the power controller responsible for battery switchover blocks the chip select signal to the RAM, the writes in progress will be aborted in any case. This is not necessarily serious as software schemes can be used to protect critical variables in battery-backed memory.

The reset sequence requires a minimum of 128 cycles of the fast oscillator to complete, even if no write cycles were in progress at the start of the reset. Reset forces both the processor clock and the peripheral clock in the divide-by-eight mode. Note that if the processor is being clocked from the 32 kHz oscillator, the 128 cycles of the fast oscillator will probably not be sufficient to allow any writes in progress to be completed before the reset sequence completes and the clocks switch to divide-by-eight mode.

During reset, all of the memory control signals are held inactive. After the /RESET signal is inactive (high), the processor begins fetching instructions and the memory control signals begin normal operation. Note that the default values in the Memory Bank Control registers select four wait states per access, so the initial program fetch memory reads are 48 clock cycles long ( $8 \times (2 + 4)$ ). Software can immediately adjust the processor timing to whatever the system requires.

The default selection for the memory control signals consists of /CS0, /OE0 and /WE0, and writes are enabled. This selection can also be immediately programmed to match the hardware configuration. A typical sequence would be to speed up the clock to full speed, then select the appropriate number of wait states and the chip select signals, output enable signals and write enable signals. At this point software would usually check the system status to determine what type of reset just occurred and begin normal operation.

## 7.8 Rabbit Interrupt Structure

An interrupt causes a call to be executed, pushing the PC on the stack and starting to execute code at the interrupt vector address. The interrupt vector addresses have a fixed lower byte value for all interrupts. The upper byte is adjustable by setting the registers EIR and IIR for external and internal interrupts respectively. There are only two external interrupts generated by transitions on certain pins in parallel port E.

The interrupt vectors are shown in Table 15.

The interrupts differ from most Z80 or Z180 interrupts in that the 256-byte tables pointed to EIR and IIR contain the actual instructions beginning the interrupt routines rather than a 16-bit pointer to the routine. The interrupt vectors are spaced 16 bytes apart so that the entire code will fit in the table for very small interrupt routines.

**Table 15. Peripheral Device Address and Interrupt Vectors**

On-Chip Peripheral	I/O Address Range	ISR Starting Address
System Management (periodic interrupt)	0000xxxx	{IIR, 00000000}
Memory Management	0001xxxx	No interrupts
Slave Port	0010xxxx	{IIR, 10000000}
Parallel Port A	0011xxxx	No interrupts
Parallel Port B	0100xxxx	No interrupts
Parallel Port C	0101xxxx	No interrupts
Parallel Port D	0110xxxx	No interrupts
Parallel Port E	0111xxxx	No interrupts
External I/O Control	1000xxxx	No interrupts
External Interrupts	1001xxxx	{EIR, 0, int, 0000} int0- I,0000000 int1 - I,0001000
Timer A	1010xxxx	{IIR, 10100000}
Timer B	1011xxxx	{IIR, 10110000}
Serial Port A	1100xxxx	{IIR, 11000000}
Serial Port B	1101xxxx	{IIR, 11010000}
Serial Port C	1110xxxx	{IIR, 11100000}
Serial Port D	1111xxxx	{IIR, 11110000}{IIR,0F0h}
RST 10 instruction	n/a	{IIR, 00100000}{IIR,20h}
RST 18 instruction	n/a	{IIR, 00110000}{IIR,30h}
RST 20 instruction	n/a	{IIR, 01000000}{IIR,40h}
RST 28 instruction	n/a	{IIR, 01010000}{IIR,50h}
RST 38 instruction	n/a	{IIR, 01110000}{IIR,70h}

Interrupts have priority 1, 2 or 3. The processor operates at priority 0, 1, 2 or 3. If an interrupt is being requested, and its priority is higher than the priority of the processor, the interrupt will take place after then next instruction. The interrupt automatically raises the processor's priority to its own priority. The old processor priority is pushed into the 4-position stack of priorities contained in the IP register. Multiple devices can be requesting interrupts at the same time. In each case there is a latch set in the device that requests the interrupt. If that latch is cleared before the interrupt is latched by the central interrupt logic, then the interrupt request is lost and no interrupt takes place. This is shown in Table 16. The priorities shown in this table apply only for interrupts of the same priority level and are only meaningful if two interrupts are requested at the same time. Most of the devices can be programmed to interrupt at priority level 1, 2 or 3.

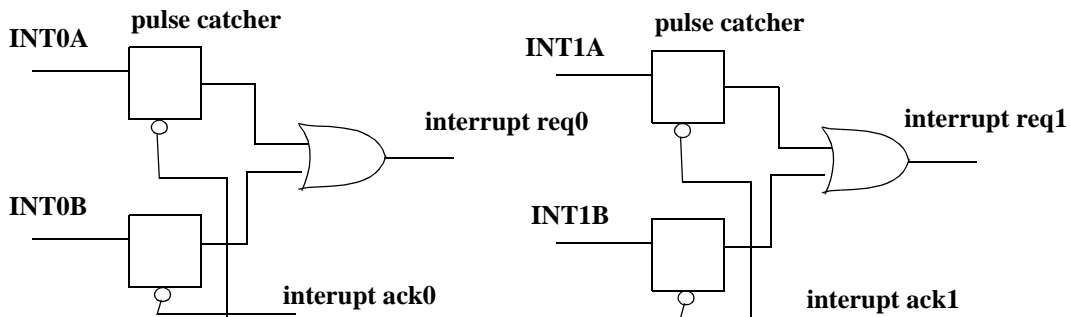
**Table 16. Interrupts—Priority and Action to Clear Requests**

Priority	Interrupt Source	Action Required to Clear the Interrupt
Highest	External 1	Automatically by interrupt acknowledge.
	External 0	Automatically by interrupt acknowledge.
	Periodic (2KHz)	Read the GCSR.
	Timer B	Read the TBSR.
	Timer A	Read the TCSR.
	Slave Port	Write SLSTAT.
	Serial Port A	Rx: Read the SADR or SAAR. Tx: Write the SADR, SAAR or SASR
	Serial Port B	Rx: Read the SBDR or SBAR. Tx: Write the SBDR, SBAR or SBSR
	Serial Port C	Rx: Read the SCDR or SCAR. Tx: Write the SCDR, SCAR or SCSR
Lowest	Serial Port D	Rx: Read the SDDR or SDAR. Tx: Write the SDDR, SDAR or SDSR

In the case of the external interrupts the only action that will clear the interrupt request is for the interrupt to take place, which automatically clears the request. A special action must be taken in the interrupt service routine for the other interrupts.

### 7.8.1 External Interrupts

There are two external interrupts. The external interrupts take place on a transition of the input, programmable for rising, falling or both edges. There are two interrupt pins in port E that can be enabled for each interrupt. Each of the interrupt pins has its own catcher device to catch the edge transition and request the interrupt.



**Figure 19. External Interrupt Line Logic**

When the interrupt takes place, both catcher devices associated with that interrupt are automatically reset. If both edges are detected before the corresponding interrupt takes place, because the triggering edges occur nearly simultaneously or because the interrupts are inhibited by the processor priority, then there will be only one interrupt for the two edges detected. The interrupt service routine can read the interrupt pins via parallel port E and determine which lines experienced a transition, provided that the transitions are not too fast. Interrupts can also be generated by setting up the matching port E bit as an output and toggling the bit.

**Table 17. Control Registers for External Interrupts**

Reg Name	Reg Address	Bits 7,6	Bits 5,4	Bits 3,2	Bits 1,0
I0CR	10011000	xx	INT0 PE4	INT0 PE0	Enb INT0
I1CR	10011001	xx	INT1 PE5	INT1 PE1	Enb INT1
			edge triggered 00-disabled 10-rising 01-falling 11-both	edge triggered 00-disabled 10-rising 01-falling 11-both	interrupt 00-disable 01-pri 1 10-pri 2 11-pri 3

Interrupt vectors: INT0 - EIR,00h / INT1 - EIR,08h

When it is desired to expand the number of interrupts for additional peripheral devices, the user should use the interrupt routine to dispatch interrupts to other virtual interrupt routines. Each additional interrupting device will have to signal the processor that it is requesting an interrupt. A separate signal line is needed for each device so that the processor can determine which devices are requesting an interrupt.

## 7.9 Bootstrap Operation

The device provides the option of bootstrap from any of three sources: from the Slave Port, from Serial Port A in clocked serial mode, or from Serial Port A in asynchronous mode. This is controlled by the state of the SMODE pins after reset. Bootstrap operation is disabled if  $(\text{SMODE1}, \text{SMODE0}) = (0, 0)$ .


Bootstrap operation inhibits the normal fetch of code from memory, and instead substitutes the output of a small internal boot ROM for program fetches. This bootstrap program reads groups of three bytes from the selected peripheral device. The first byte is the most significant byte of a 16-bit address, followed by the least-significant byte of a 16-bit address, followed by a byte of data. The bootstrap program then writes the byte of data to the downloaded address and jumps back to the start of the bootstrap program. The most significant bit of the address is used to determine the destination for the byte of data. If this bit is zero, the byte is written to the memory location addressed by the downloaded address. If this bit is one, the byte is written to the internal peripheral addressed by the downloaded address. Note that all of the memory control signals continue to operate normally during bootstrap.

Execution of the bootstrap program automatically waits for data to become available from the selected peripheral, and each byte transferred automatically resets the watchdog timer. However, the watchdog timer still operates, and bytes must be transferred often enough to prevent the watchdog timer from timing out.

Bootstrap operation is terminated when the SMODE pins are set to zero. The SMODE pins are sampled just prior to fetching the first instruction of the bootstrap program. If the SMODE pins are zero, instructions are fetched from normal memory starting at address 0000h. The Slave Port Control register allows the bootstrap operation to be terminated remotely. Writing a one to bit 7 of this register causes the bootstrap operation to terminate immediately. So the sequence 80h, 24h and 80h will terminate bootstrap operation.

Bootstrap operation is not restricted to the time immediately after reset, because the boot ROM is addressed by only the four least significant bits of the address. So any time that the address ends in four zeros, if the SMODE pins are non-zero and bit 7 of the SPCR is zero, the bootstrap program will begin execution. This allows in-line downloading from the selected bootstrap port. Upon completion of the bootstrap operation, either by returning the SMODE pins to zero or setting the bit in the SPCR, execution will continue from where it was interrupted for the bootstrap operation.

The Slave Port is selected for bootstrap operation when  $(\text{SMODE1}, \text{SMODE0}) = (0, 1)$ . In this case the pins of Parallel Port A are used for a byte-wide data bus, and selected pins of Parallel Ports B and E are used for the Slave Port control signals. Only Slave Port Data Register 0 is used for bootstrap operation, and any writes to the other data registers will be ignored by the processor, and can actually interfere with the bootstrap operation by masking the Write Empty signal.

 See Section 14.8.2 on page 129 for an example of using the bootstrap mode.

Serial Port A is selected for bootstrap operation as a clocked serial port when SMODE = 10. In this case bit 7 of Parallel Port C is used for the serial data and bit 1 of Parallel Port B is used for the serial clock. Note that the serial clock must be externally supplied for bootstrap operation. This precludes the use of a serial EEPROM for bootstrap operation.

Serial Port A is selected for bootstrap operation as an asynchronous serial port when SMODE = 11. In this case bit 7 of Parallel Port C is used for the serial data and the 32 kHz oscillator is used to provide the serial clock. A dedicated divide circuit allows the use of the 32 kHz signal to provide the timing reference for the 2400 bps asynchronous transfer. Only 2400 bps is supported for bootstrap operation, and the serial data must be eight bits for proper operation.

When a bootstrap is performed using Serial Port A, the TXA signal is not needed since the bootstrap is a one-way communication. After the reset ends and the bootstrap mode begins, TXA will be low, reflecting its function as a parallel port output bit that is cleared by the reset. This may be interpreted as a break signal by some serial communication devices. TXA can be forced high by sending the triplet 80h, 50h, 40h, which stores 40h in parallel port C. An alternate approach is to send the triplet 80h, 55h, 40h, which will enable the TXA output from bit 6 of parallel port C by writing to the parallel port C function register (55h).

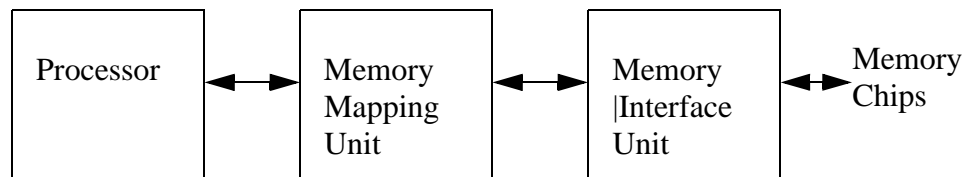
The transfer rate in any bootstrap operation must not be too fast for the processor to execute the instruction stream. The Write Empty signal acts as an interlock when using the Slave Port for bootstrap operation, because the next byte should not be written to the Slave Port until the Write Empty signal is active. No such interlock exists for the clocked serial and asynchronous bootstrap operation. In these cases, remember that the processor clock starts out in divide-by-eight mode with four wait states, and limit the transfer rate accordingly. In asynchronous mode at 2400 bps it takes about 4 ms to send each character, so no problem is likely unless the system clock is extremely slow.



## 8. Rabbit Memory Mapping and Interface

See Section 3.2 on page 20 for a tutorial discussion of the Rabbit memory mapping.

Figure 20 shows an overview of the Rabbit memory mapping. The task of the memory mapping unit is to accept 16-bit addresses and translate them to 20-bit addresses. The memory interface unit accepts the 20-bit addresses and generates control signals applied directly to the memory chips.



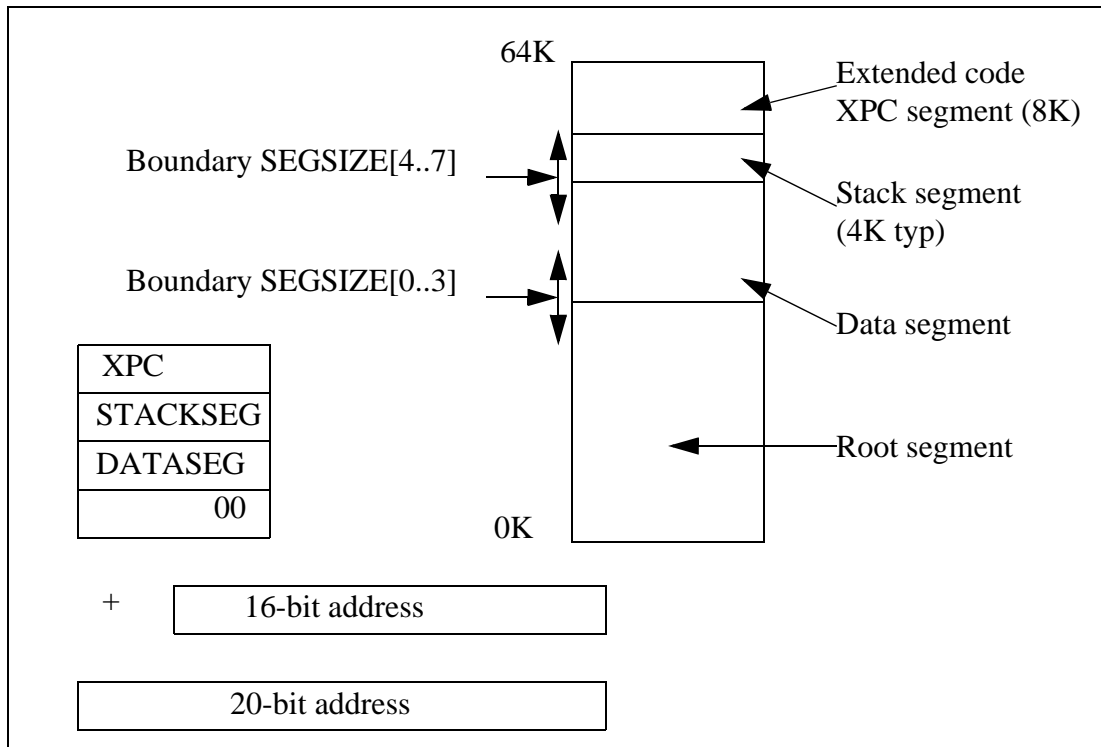
**Figure 20. Overview of Rabbit Memory Mapping**

### 8.1 Memory-Mapping Unit

The 64K 16-bit address space accessed by processor instructions is divided into segments. Each segment has a length that is a multiple of 4K. Except for the extended code segment, the segments have adjustable sizes and some segments can be reduced to zero size and thus vanish from the memory map.

The four segments are shown in the example in Figure 21. The segment size register (SEGSIZE) determines the boundaries marked in the diagram. The extended code segment always occupies the addresses 0E000h–0FFFFh. The stack segment stretches from the address specified by the upper 4 bits of the SEGSIZE register to 0DFFFh. For example, if the upper 4 bits of SEGSIZE are 0Dh, then the stack segment will occupy 0D000h–0DFFFh, or 4K. If the upper 4 bits of SEGSIZE are greater than or equal to 0Eh, the stack segment vanishes. If these bits are set to zero, the two segments below the stack segment will vanish.

The lower 4 bits of SEGSIZE determine the lower boundary shown in the figure. If this boundary is equal to the upper boundary or greater than 0Eh, the data segment will vanish. If this segment is placed at zero the code segment will vanish.



**Figure 21. Memory Segments**

The memory management unit accepts a 16-bit address from the processor and translates it into a 20-bit address. The procedure to do this works as follows.

1. It is determined which segment the 16-bit address belongs to by inspecting the upper 4 bits of the address. Every address must belong to one of the possible 4 segments.
2. Each segment has an 8-bit segment register. The 8-bit segment register is added to the upper 4 bits of the 16-bit address to create a 20-bit address. Wraparound occurs if the addition would result in an address that does not fit in 20 bits.

**Table 18. Segment Registers**

Segment Register	Function
XPC	Locates extended code segment in physical memory. Read and written by processor instructions: ld a,xpc, ld xpc,a, lcall, lret, ljp
STACKSEG = 11h	Locates stack segment in physical memory.
DATASEG = 12h	Locates data segment in physical memory.

**Table 19. Segment Size Register**

	Bits 7..4	Bits 3..0
SEGSIZE = 13h	Boundary address stack segment.	Boundary address data segment.

## 8.2 Memory Interface Unit

The 20-bit memory addresses generated by the memory-mapping unit feed into the memory interface unit. The memory interface unit has a separate control register (see Table 20 on page 83) for each 256K quadrant of the 1M physical memory. This control register specifies how memory access requests to that quadrant are to be dispatched to the memory chips connected to the Rabbit. There are three separate chip select output lines (/CS0, /CS1, and /CS2) that can be used to select one of three different memory chips. A field in the control register determines which chip select (or none) is selected for memory accesses to the quadrant. The same chip select line may be accessed in more than one quadrant. For example, if a 512K RAM is installed and is selected by /CS1, it would be appropriate to use /CS1 for accesses to the 3rd and 4th quadrants, thus mapping the RAM chip to addresses 80000h to 0FFFFFFh.

**Table 20. Memory Bank Control Register  $x$  ( $MBxCR=14h+x$ )**

Bits 7,6	Bit 5	Bit 4	Bit 3	Bit 2	Bits 1,0
00—4 wait states 01—2 wait states 10—1 wait states 11—0 wait states	1—Invert address A19	1—Invert address A18	1—Write-pro- tect memory this quadrant	0—use /OE0, /WE0 1—use /OE1, /WE1	00—use /CS0 01—use /CS1 1x—use /CS2

## 8.3 Memory Bank Control Register Functions

Table 20 describes the operation of the four memory bank control registers. Each register controls one quadrant in the 1M address space.

- Bits 7,6—The number of wait states used in access to this quadrant. Without wait states, read requires 2 clocks and write requires 3 clocks. The wait state adds to these numbers.
- Bits 5, 4—These bits allow the upper address lines to be inverted. This inversion occurs after the logic that selects the bank register, so setting these lines has no effect on which bank register is used. The inversion may be used to install a 1M memory chip in the space normally allocated to a 256K chip. The larger memory can then be accessed as 4 pages of 256K each. There is no effect outside the quadrant that the memory bank control register is controlling.
- Bit 3—Inhibits the write pulse to memory accessed in this quadrant. Useful for protecting flash memory from an inadvertent write pulse, which will not actually write to the flash because it is protected by lock codes, but will temporarily disable the flash memory and crash the system if the memory is used for code.
- Bit 2—Selects which set of the two lines /OEx and /WEx will be driven for memory accesses in this quadrant.
- Bits 1,0—Determines which of the three chip select lines will be driven for memory accesses to this quadrant.
- All bits of the control register are initialized to zero on reset.

### 8.3.1 Optional A16, A19 Inversions by Segment (/CS1 Enable)

The inversion of A19 or A16 controlled by the MMIDR register is used to redirect mapping of the root segment and the data segment by inverting certain bits when these segments are accessed. Currently this functionality has no planned usage.

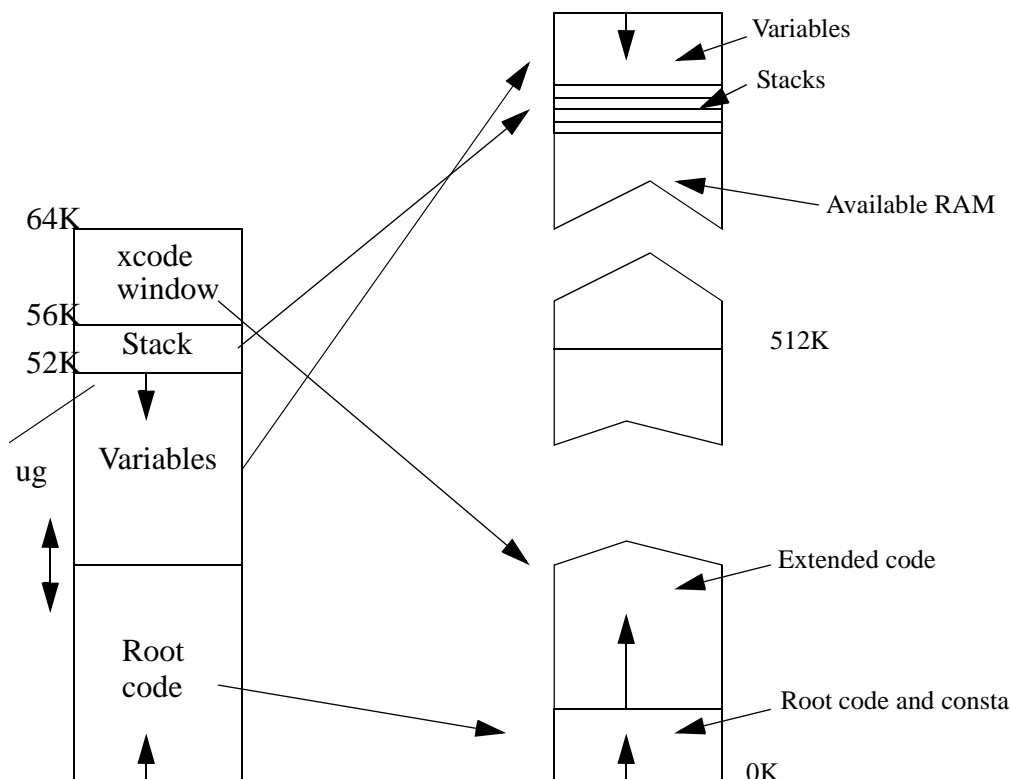
The optional enable of /CS1 is valuable for systems that are pushing the access time of battery-backed RAM. By enabling /CS1, the delay time of the switch that forces /CS1 high when power is off can be bypassed. This feature increases power consumption since the RAM is always enabled and its access is controlled normally by /OE1.

**Table 21. MMU Instruction/Data Register (MMIDR = 010h)**

Bits 7,6,5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
000	1—force /CS1 always enabled	1—invert A19 for accesses in data segment before quadrant selection	1—invert A16 for accesses in data segment	1—invert A19 for accesses in root segment before quadrant selection	1—invert A16 for accesses in root segment

### 8.4 Allocation of Extended Code and Data

The Dynamic C compiler compiles code to root code space or to extended code space. Root code starts in low memory and compiles upward.



**Figure 22. Typical Memory Mapping and Memory Usage**

Allocation of extended code starts above the root code and data. Allocation normally continues to the end of the flash memory.

Data variables are allocated to RAM working backwards in memory. Allocation normally starts at 52K in the 64K D space and continues. The 52K space must be shared with the root code and data, and is allocated upward from zero.

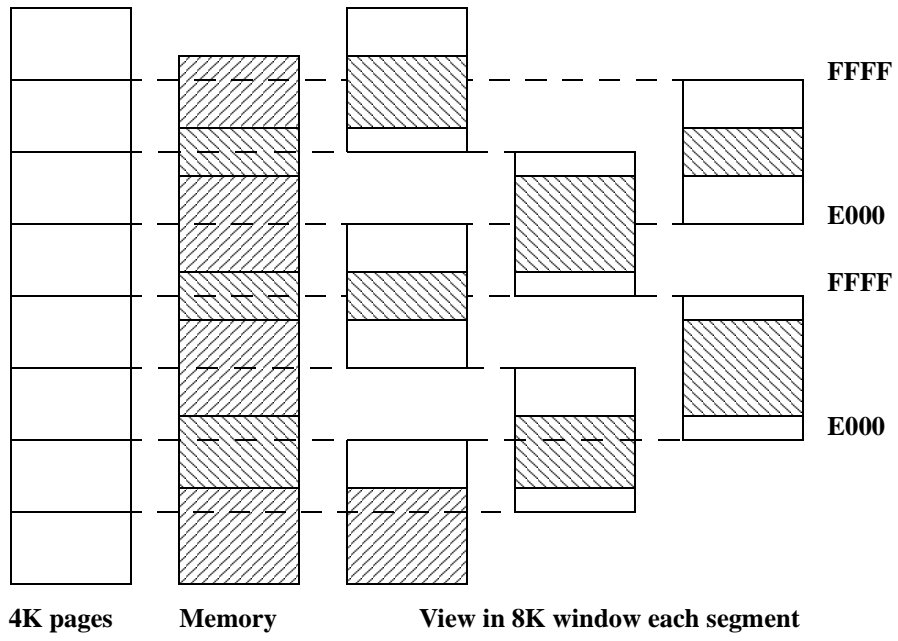
Dynamic C also supports extended data constants. These are mixed in with the extended code in flash.

## 8.5 How the Compiler Compiles to Memory

The compiler actually generates code for root code and constants and extended code and extended constants. It allocates space for data variables, but does not generate data bits to be stored in memory.

In any but the smallest programs, most of the code is compiled to extended memory. This code executes in the 8K window from E000 to FFFF. This 8K window uses paged access. Instructions that use 16-bit addressing can jump within the page and also outside of the page to the remainder of the 64K space. Special instructions, particularly long call, long jump and long return, are used to access code outside of the 8K window. When one of these transfer of control instructions is executed, both the address and the view through the 8K window or page are changed. This allows transfer to any instruction in the 1M memory space. The 8-bit XPC register controls which of the 256 4K pages the 8K window aligns with. The 16-bit PC controls the address of the instruction, usually in the region E000 to FFFF. The advantage of paged access is that most instructions continue to use 16-bit addressing. Only when an out-of-range transfer of control is made does a 20-bit transfer of control need to be made. The beauty of having a 4K minimum step in page alignment while the size of the page is 8K is that code can be compiled continuously without gaps caused by change of page. When the page is moved by 4K, the previous end of code is still visible in the window, provided that the midpoint of the page was crossed before moving the page alignment.

As the compiler compiles code in the extended code window, it checks at opportune times to see if the code has passed the midpoint of the window or F000. When the code passes F000, the compiler slides the window down by 4K so that the code at F000+x becomes resident at E000+x. This results in the code being divided into segments that are typically 4K long, but which can vary very short or as long as 8K. Transfer of control can be accomplished within each segment by 16-bit addressing; 20-bit addressing is required between segments.



**Figure 23. Compilation of Code Segments in Extended Memory**

## 9. Parallel Ports

The Rabbit has five 8-bit parallel ports designated A, B, C, D and E. The pins used for the parallel ports are also shared with numerous other functions as shown in Table 4 on page 59. The important properties of the ports are summarized below.

- Port A—Shared with the slave port data interface.
- Port B—Shared with control lines for slave port and clock I/O for clocked serial mode option for serial ports A and B.
- Port C—Shared with serial port serial data I/O.
- Port D—4 bits shared with alternate I/O pins for serial ports A and B. 4 bits not shared. Port D has the ability to configure its outputs as open drain outputs. Port D has output preload registers that can be clocked into the output registers under timer control for pulse generation. Port D has higher current drive capability.
- Port E—All bits of Port E can be configured as I/O strobes. 4 bits of port E can be used as external interrupt inputs. One bit of port E is shared with the slave port chip select. Port E has output preload registers that can be clocked into the output registers under timer control for pulse generation. Port E has higher current drive capability.

### 9.1 Parallel Port A

Parallel Port A has a single read/write register, shown in Table 22.

**Table 22. Parallel Port A Data Register (adr = 030h)**

R/W 8-bit Data Value
----------------------

If the slave port is enabled, this register should not be used. The slave port control register is used to control whether this port is an output or input. To make the port input, store 080h in the SPCR slave port control register. To make the port output, store 084h in SPCR. Upon reset, port A is set up as an input port.

When the port is read, the value read reflects the voltages on the pins, "1" for high and "0" for low. This could be different than the value stored in the output register if the pin is forced to a different state by an external voltage.

## 9.2 Parallel Port B

Parallel Port B, shown in Table 23, has six inputs and two outputs when used exclusively as a parallel port.

**Table 23. Parallel Port B Data Register PBDR (adr = 040h)**

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Read	Echo drive	Echo drive	PB5 in	PB4 in	PB3 in	PB2 in	PB1 in	PB0 in
Write	PB7	PB6	x	x	x	x	x	x

When the slave port is enabled, parallel port lines PB2–PB7 are assigned to various slave port functions. However, it is still possible to read PB0–PB5 using the Port B data register even when lines PB2–PB7 are used for the slave port. It is also possible to read the signal driving PB6 and PB7 (this signal is on the signaling lines from the slave port logic).

Regardless of whether the slave port is enabled, PB0 reflects the input of the pin unless serial port B has its internal clock enabled, which causes this line to be driven by the serial port clock. PB1 reflects the input of the pin unless serial port A has its internal clock enabled.

On reset the output bits 6 and 7 are reset and the value output on pins PB6 and PB7 (package pins 99, 100) will also be low.

## 9.3 Parallel Port C

Parallel port C, shown in Table 24, has four inputs and four outputs. The even-numbered ports, PC0, PC2, PC4, and PC6, are outputs. The odd-numbered ports, PC1, PC3, PC5, and PC7, are inputs. When the data register is read, bits 1,3,5,7 return the value of the voltage on the pin. Bits 0,2,4,6 return the value of the signal driving the output buffers. The signal driving the output buffers and the value of the output pin are normally the same. Either the Port C data register is driving these pins or one of the serial port transmit lines is driving the pin. The bits set in the PCFR Parallel Port C Function Register identify whether the data register or the serial port transmit lines were driving the pins.

**Table 24. Parallel Port C Data Register and Function Register**

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
PCDR (r) adr = 050h	PC7 in	Echo drive	PC5 in	Echo drive	PC3 in	Echo drive	PC1 in	Echo drive
PCDR (w) adr = 050h	x	PC6	x	PC4	x	PC2	x	PC0
PCFR (w) adr = 055h	x	Drive TXA	x	Drive TXB	x	Drive TXC	x	Drive TXD



Parallel port C shares its pins with the four serial ports. The parallel port input pins may also serve as serial port inputs. (Serial ports A and B can alternately use pins in Port D as inputs, and the source of the serial port inputs for these serial ports depends on the setup of the corresponding serial port control register.) When serving as serial inputs, the data lines can still be read from the parallel port C data register. The parallel port outputs can be selected to be serial port outputs by storing bits in the corresponding positions of the Port C Function register (PCFR). When a parallel port output pin is selected to be a serial port output, the value stored in the data register is ignored. On reset the active (even-numbered) function register bits and data register bits are zeroed. This causes the port to output zeros on the four output bits.

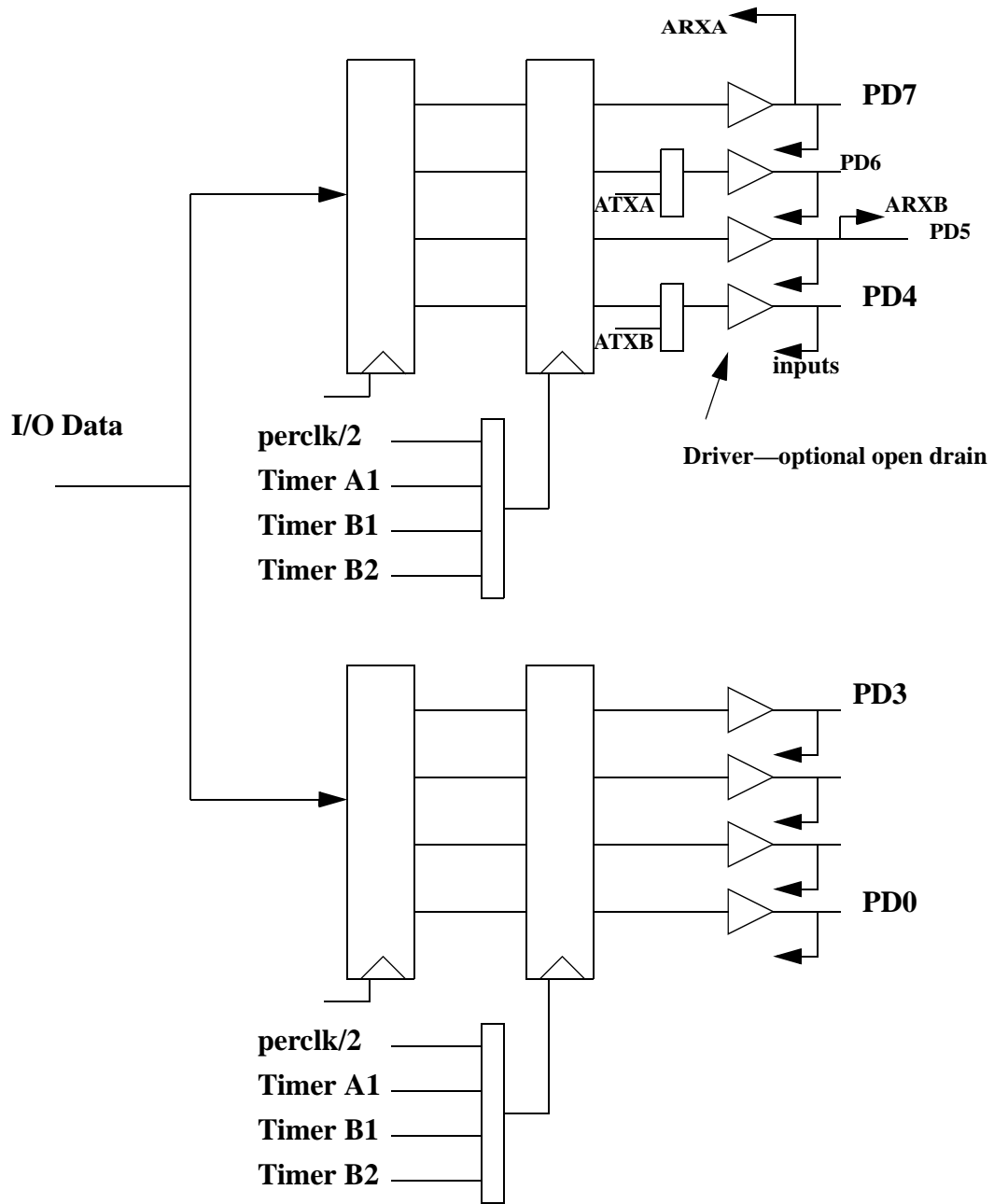
## 9.4 Parallel Port D

Parallel port D, shown in Figure 24 on page 90, has eight pins that can be programmed individually to be inputs and outputs. When programmed as outputs, the pins can be individually selected to be open-drain outputs or standard outputs. Port D pins can be addressed by bit if desired. The output registers are cascaded and timer-controlled, making it possible to generate precise timing pulses. In addition, port D (and E) outputs have a higher drive capability. Port D outputs 4 and 5 can be used as alternate pins for serial port B, and outputs 6 and 7 can be used as alternate pins for serial port A. Alternate serial port outputs make it possible for the same serial port to connect to different communications lines that are not operating at the same time.

On reset, the data direction register is zeroed, making all pins inputs. In addition bits in the control register are zeroed (bits 0,1,4,5) to ensure that data is clocked into the output registers when loaded. All other registers associated with port D are not initialized on reset.

The following registers are described in Table 25 and in Table 26.

- PDDR—Parallel port D data register. Read/Write.
- PDDDR—Parallel port D data direction register. A "1" makes the corresponding pin an output. Write only.
- PDDCR—Parallel port D drive control register. A "1" makes the corresponding pin an open-drain output if that pin is set up for output. This register is zeroed by reset. Write only.
- PCFR—Parallel port D function control register. This port may be used to make port positions 4 and 6 be serial port outputs. Write only.
- PDBxR—These eight registers may be used to set outputs on individual port positions.
- PDCR—Parallel port D control register. This register is used to control the clocking of the upper and lower nibble of the final output register of the port. On reset, bits 0, 1, 4, and 5 are reset to zero.



**Figure 24. Parallel Port D Block Diagram**

**Table 25. Parallel Port D Registers**

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
PDDR (R/W) adr = 060h	PE7	PD6	PD5	PD4	PD3	PD2	PD1	PD0
PDDCR (W) adr = 066h	out = open drain	out = open drain	out = open drain	out = open drain	out = open drain	out = open drain	out = open drain	out = open drain
PDFR (W) adr = 065h	x	alt TXA	x	alt TXB	x	x	x	x
PDDDR (W) adr = 067h	dir = out	dir = out	dir = out	dir = out	dir = out	dir = out	dir = out	dir = out
PDB0R (W) adr = 068h	x	x	x	x	x	x	x	PD0
PDB1R (W) adr = 069h	x	x	x	x	x	x	PD1	x
PDB2R (W) adr = 06Ah	x	x	x	x	x	PD2	x	x
PDB3R (W) adr = 06Bh	x	x	x	x	PD3	x	x	x
PDB4R (W) adr = 06Ch	x	x	x	PD4	x	x	x	x
PDB5R (W) adr = 06Dh	x	x	PD5	x	x	x	x	x
PDB6R (W) adr = 06Eh	x	PD6	x	x	x	x	x	x
PDB7R (W) adr = 06Fh	PD7	x	x	x	x	x	x	x

**Table 26. Parallel Port D Control Register (adr = 064h)**

Bits 7, 6	Bits 5, 4	Bits 3, 2	Bits 1, 0
x	00—clock upper nibble on pclk/2 01—clock on timer A1 10—clock on timer B1 11—clock on timer B2	x	00—clock lower nibble on pclk/2 01—clock on timer A1 10—clock on timer B1 11—clock on timer B2

## 9.5 Parallel Port E

Parallel port E, shown in Figure 25, has eight I/O pins that can be individually programmed as inputs or outputs. Port E has a higher drive than most of the other ports. PE7 is used as the slave port chip select when the slave port is enabled. Each of the port E outputs can be configured as an I/O strobe. In addition, four of the port E lines can be used as interrupt request inputs. The output registers are cascaded and timer-controlled, making it possible to generate precise timing pulses.

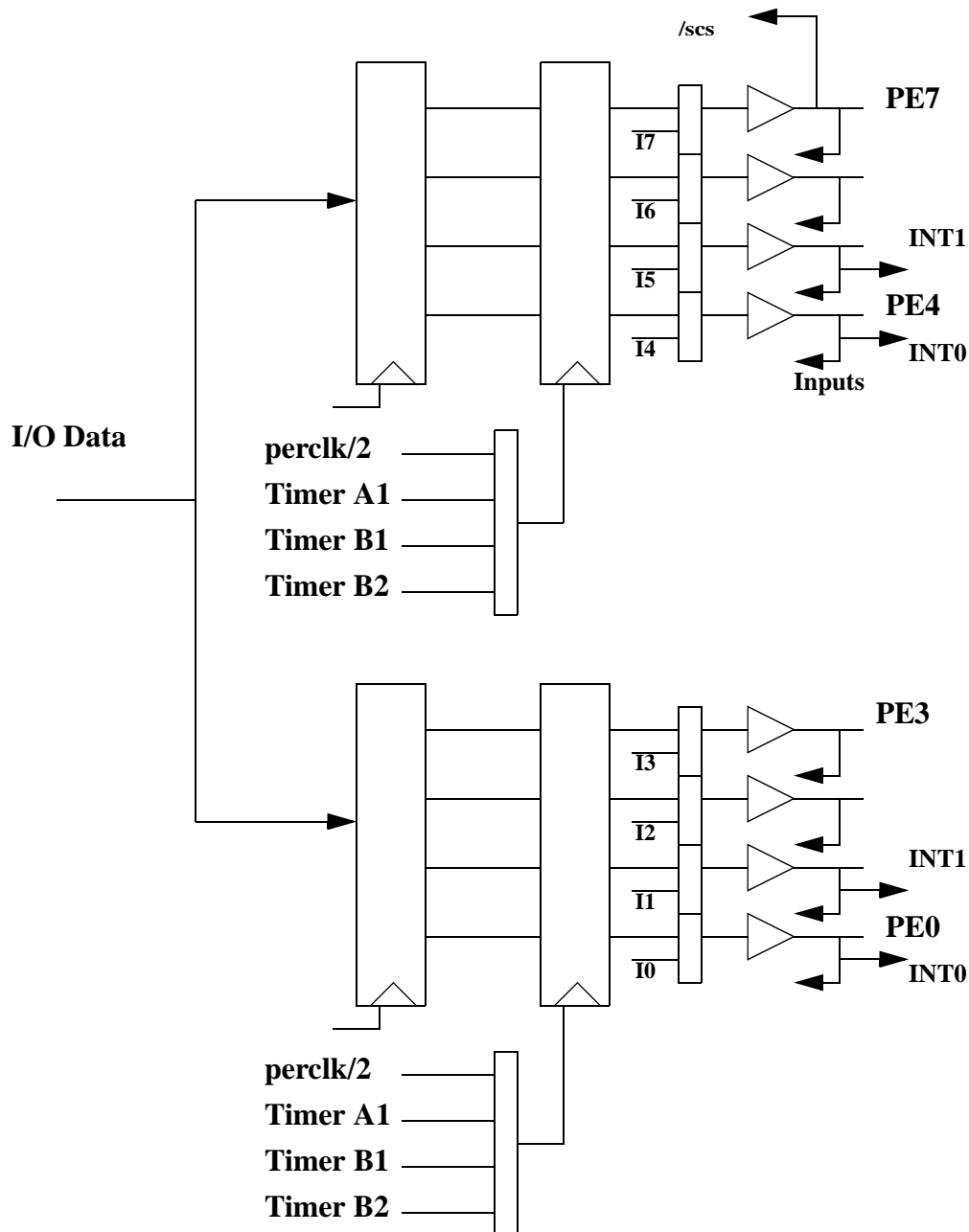


Figure 25. Parallel Port E Block Diagram

The following registers are described in Table 27 and in Table 28.

- PEDR—Port E data register. Reads value at pins. Writes to port E preload register.
- PEDDR—Port E data direction register. Set to "1" to make corresponding pin an output. This register is zeroed on reset.
- PEFR—Port E function register. Set bit to "1" to make corresponding output an I/O strobe. The nature of the I/O strobe is controlled by the I/O bank control registers (IBxCR). The data direction must be set to output for the I/O strobe to work.
- PEBxR—These are individual registers to set individual output bits on or off.
- PECR—Parallel port E control register. This register is used to control the clocking of the upper and lower nibble of the final output register of the port. On reset, bits 0, 1, 4, and 5 are reset to zero.

**Table 27. Parallel Port E Registers**

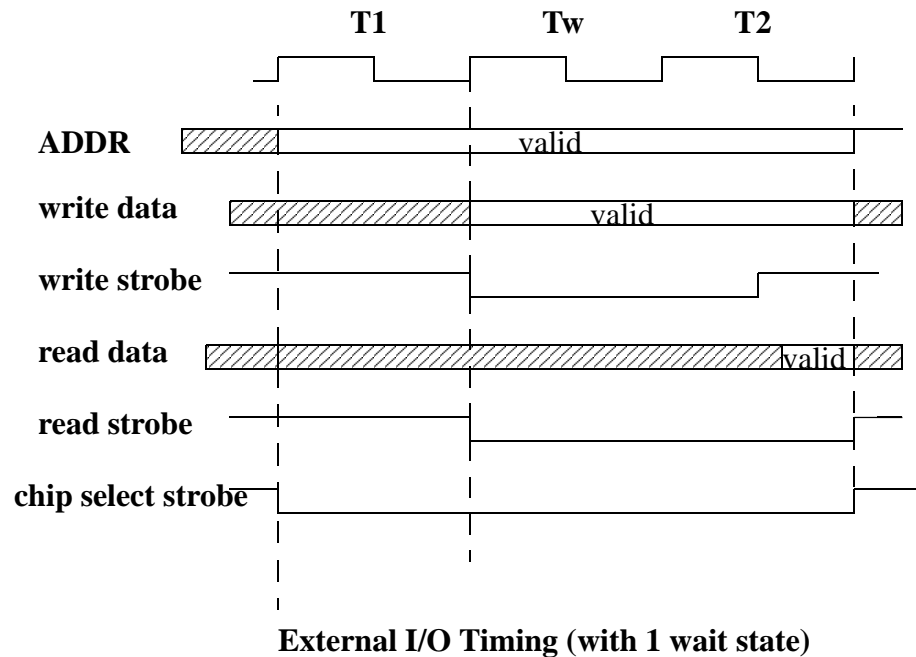
	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
PEDR (R/W) adr = 070h	PE7	PE6	PE5	PE4	PE3	PE2	PE1	PE0
PEFR (W) adr = 075h	alt /I7	alt /I6	alt /I5	alt /I4	alt /I3	alt /I2	alt /I1	alt /I0
PEDDR (W) adr = 077h	dir = out	dir = out	dir = out	dir = out	dir = out	dir = out	dir = out	dir = out
PEB0R (W) adr = 078h	x	x	x	x	x	x	x	PE0
PEB1R (W) adr = 079h	x	x	x	x	x	x	PE1	x
PEB2R (W) adr = 07Ah	x	x	x	x	x	PE2	x	x
PEB3R (W) adr = 07Bh	x	x	x	x	PE3	x	x	x
PEB4R (W) adr = 07Ch	x	x	x	PE4	x	x	x	x
PEB5R (W) adr = 07Dh	x	x	PE5	x	x	x	x	x
PEB6R (W) adr = 07Eh	x	PE6	x	x	x	x	x	x
PEB7R (W) adr = 07Fh	PE7	x	x	x	x	x	x	x

**Table 28. Parallel Port E Control Register (adr = 074h)**

Bits 7, 6	Bits 5, 4	Bits 3, 2	Bits 1, 0
x	00—clock upper nibble on pclk/2 01—clock on timer A1 10—clock on timer B1 11—clock on timer B2	x	00—clock lower nibble on pclk/2 01—clock on timer A1 10—clock on timer B1 11—clock on timer B2

## 10. I/O Bank Control Registers

The pins of port E can be individually set to be I/O strobes. Each of the eight possible I/O strobes has a control register that controls the nature of the strobe and the number of wait states that will be inserted in the I/O bus cycle. Writes can also be suppressed for any of the strobes. The types of strobes are shown in Figure 26. Each of the eight I/O strobes is active for addresses occupying 1/8th of the 64K external I/O address space.



**Figure 26. External I/O Bus Cycles**

Table 29 shows how the eight I/O bank control registers are organized.

**Table 29. I/O Bank Control Reg (adr IBxCR = 08xh)**

Bits 7,6	Bits 5,4	Bit 3	Bits 2-0
Wait state code	/IX strobe type	1—permit write	Ignored
11-1	00—chip select	0—inhibit write	
10-3	01—read strobe		
10-7	10—write strobe		
00-15	11—or of read and write strobe		

The I/O strobes greatly simplify the interfacing of external devices. On reset the upper 5 bits of each register is cleared. Parallel port E will not output these signals unless the data direction register bits are set for the desired output positions. In addition, the parallel port E function register must be set to "1" for each position.

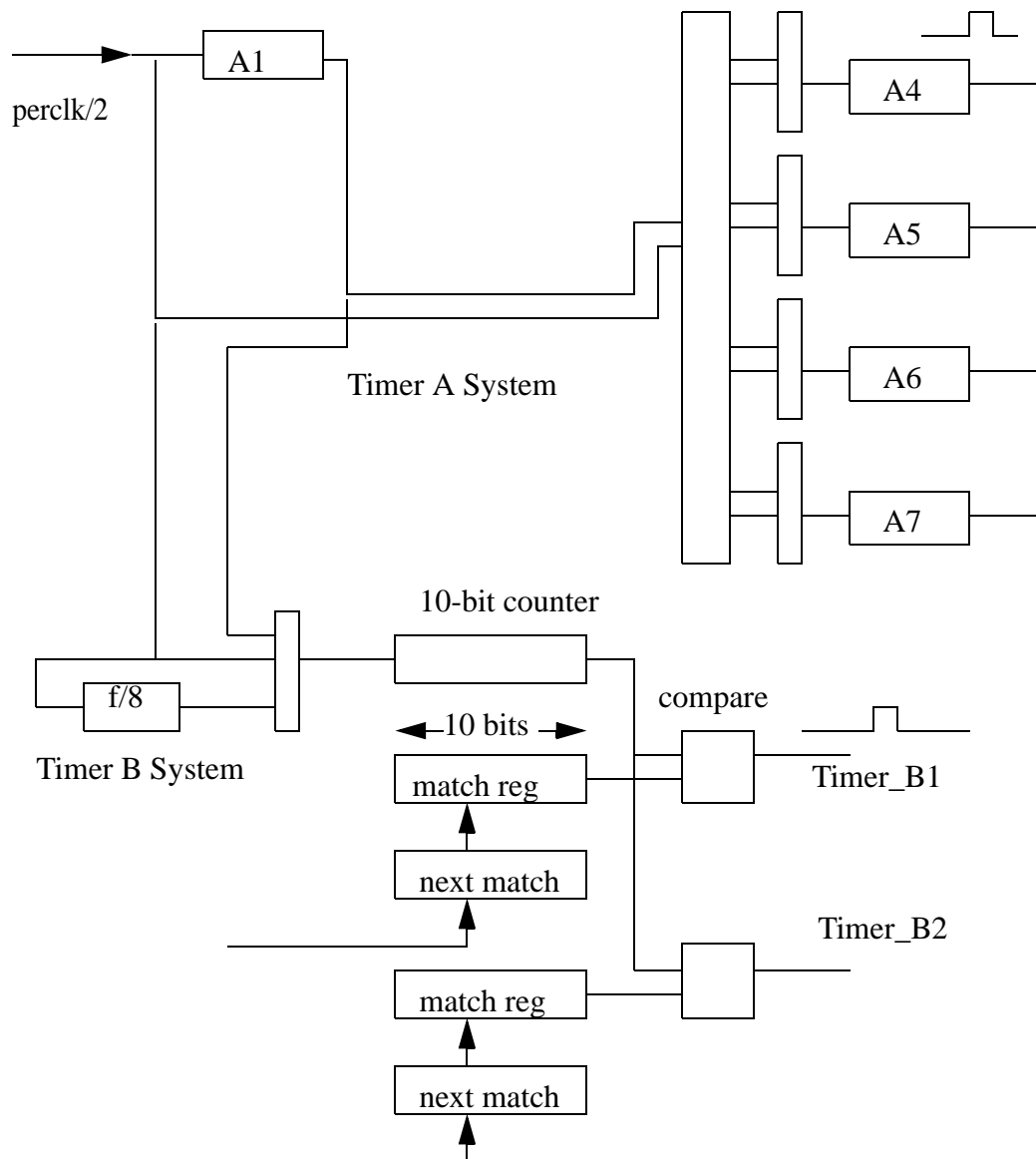




## 11. Timers

There are two timers—Timer A and Timer B. Timer A is intended mainly for generating the baud clock for the serial ports, a periodic clock for clocking parallel ports D and E, or for generating periodic interrupts. Timer B can be used for the same functions, but it is more flexible because the program can read the time from a continuously running counter and events can be programmed to occur at a specified future time.

Figure 27 shows a block diagram of Timers A and B.

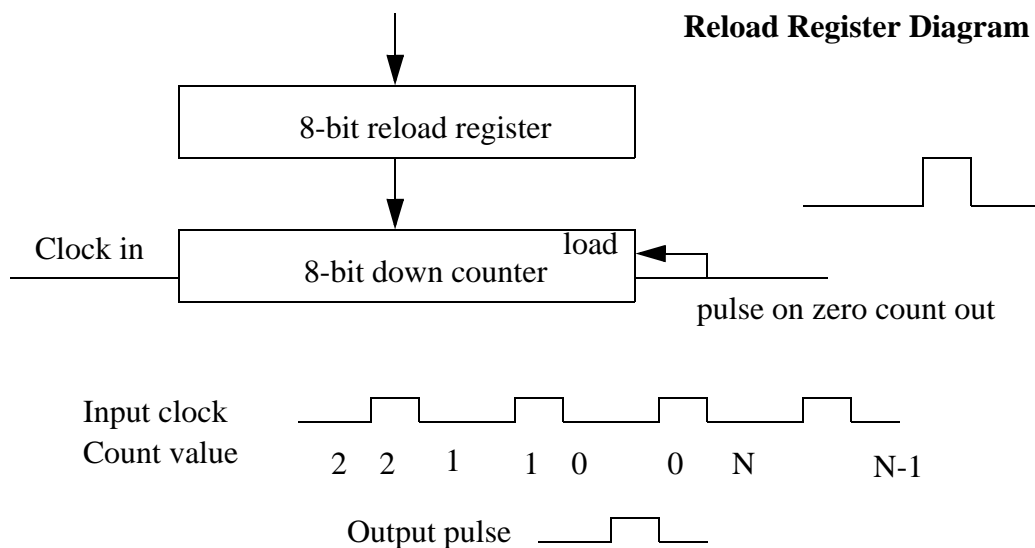


**Figure 27. Block Diagram of Timers A and B**

## 11.1 Timer A

Timer A consists of five separate countdown timers—A1 and A4–A7—as shown in Figure 27.

Timers A1 and A4–A7 are 8-bit countdown registers as shown in Figure 28. The reload register can contain any number in the range from 0 to 255. The counter divides by (n+1). For example, if the reload register contains 127, then 128 pulses enter on the left before a pulse exits on the right. If the reload register contains zero, then each pulse on the left results in a pulse on the right, that is, there is division by one.



**Figure 28. Reload Register Operation**

The timer systems are driven by the peripheral clock divided by two. This clock is always the same as the processor clock, or it is faster than the processor clock by a factor of eight. The output pulses are always one clock long. Clocking of the counters takes place on the negative edge of this pulse. When the counter reaches zero, the reload register is loaded on the next input pulse instead of a count being performed. The reload registers may be reloaded at any time since the peripheral clock is synchronous with the processor clock.

Timers A4, A5, A6 and A7 always provide the baud clock for serial ports A, B, C and D respectively. Except for very low baud rates, clock A1 does not need to be used to prescale the input clock for timers A4–A7. For example, if the system clock is 22.184 MHz, and if the timer A4 divides by 240, a baud rate of 2400 bps can be achieved in one step. The clock input to the serial port must be 16 times the baud rate for asynchronous mode and 8 times the baud rate for synchronous mode. The maximum asynchronous baud rate with a 22.1184 MHz clock would be  $(22,118,400 / (2 * 16)) = 691,200$ .

Each of the five countdown registers in timer A can cause an interrupt. There is one interrupt vector for timer A and a common interrupt priority. A common status register (TACSR) has a bit for each timer that indicates if the output pulse for that timer has taken

place since the last read of the status register. When the status register is read, these bits are cleared. No bit will be lost. Either it will be read by the status register read or it will be set after the status register read is complete. If a bit is on and the corresponding interrupt is enabled, an interrupt will occur when priorities allow. However, a separate interrupt is not guaranteed for each bit with an enabled interrupt. If the bit is read in the status register, it is cleared and no further interrupt corresponding to that bit will be requested. It is possible that one bit will cause an interrupt and then one or more additional bits will be set before the status register is read. After these bits are cleared, they cannot cause an interrupt. If any bits are on, and the corresponding interrupt is enabled, then the interrupt will take place as soon as priorities allow. However, if the bit is cleared before the interrupt is latched, the bit will not cause an interrupt. The proper rule to follow is for the interrupt routine to handle all bits that it sees set.

### 11.1.1 Timer A I/O Registers

The I/O registers for Timer A are listed in Table 30.

**Table 30. Timer A I/O Registers**

Register Name	Register Mnemonic	I/O address (hex)	R/W
Timer A Control/Status Register	TACSR	A0	R/W
Timer A Control Register	TACR	A4	W
Timer A1 Time Constant 1 Register	TAT1R	A3	W
Timer A4 Time Constant 4 Register	TAT4R	A9	W
Timer A5 Time Constant 5 Register	TAT5R	AB	W
Timer A6 Time Constant 6 Register	TAT6R	AD	W
Timer A7 Time Constant 7 Register	TAT7R	AF	W

The control/status register for Timer A (TACSR) is laid out as shown in Table 31.

**Table 31. Timer A Control and Status Register (adr = 0A0h)**

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Read	A7 count done	A6 count done	A5 count done	A4 count done	0	0	A1 count done	???
Write	A7 interrupt enable	A6 interrupt enable	A5 interrupt enable	A4 interrupt enable	x	x	A1 interrupt enable	1—enable Timer A

Bits 1, 4–7—Read only, terminal count reached on timers A1 and A4–A7. Reading this status register clears any bits (bits 1 and 4–7) that are on.

Bit 0—Write, set to a "1" to enable the clock (perclk/2) for Timer A, set to "zero" to disable the clock (perclk/2 in Figure 27 on page 97). Bits 1 and 4–7 are written (write only) to enable the interrupt for the corresponding timer.

The control register (TACR) is laid out as shown in .

**Table 32. Timer A Control Register (adr = 0A4h)**

Bit 7 A7	Bit 6 A6	Bit 5 A5	Bit 4 A4	Bits 3, 2	Bits 1, 0
Source A7 0-pclk/2 1-A1	Source A6 0-pclk/2 1-A1	Source A5 0-pclk/2 1-A1	Source A4 0-pclk/2 1-A1	not used ignored	00—Interrupt disabled 01—Enable priority 1 interrupt 10—Enable priority 2 interrupt 11—Enable priority 3 interrupt

The time constant register for each timer is simply an 8-bit data register holding a number between 0 and 255. The time constant registers are write only.

### 11.1.2 Practical Use of Timer A

Timer A is disabled (bit 0 in control and status register) on power-up. Timer A is normally set up while the clock is disabled, but the timer setup can be changed while the timer is running when there is a need to do so. Timers that are not used should be driven from the output of A1 and the reload register should be set to 0. This will cause counting to be as slow as possible and consume minimum power.

Timer A has five separate subtimer units, A1 and A4–A5, that are also referred to as timers.

Most likely, if a serial port is going to be used and a timer is needed to provide the baud clock, that timer will be set up to be driven directly from the clock, and the interrupt associated with that timer will be disabled. (Serial port interrupts are generated by the serial port logic.) This will isolate the serial port countdown register from the other countdown registers.

The value in the reload register can be changed while the timer is running to change the period of the next timer cycle. When the reload register is initialized, the contents of the countdown counter may be unknown, for example, during power-up initialization. If interrupts are enabled, then the first interrupt may take place at an unknown time. Similarly, if the timer output is being used to drive the clock for a parallel port or serial port, the first clock may come at a random time. If a periodic clock is desired, it is probably not important when the first clock takes place unless a phase relationship is desired relative to a different timers.

A phase relationship between two timers can be obtained in several ways. One way is to set both reload registers to zero and to wait long enough for both timers to reload (maximum 256 clocks). Then both timers' reload registers can be set to new values before or after both are clocked.

## 11.2 Timer B

Figure 27 on page 97 shows a block diagram of Timer B. The Timer B counter can be driven directly by  $\text{perclk}/2$ , by that clock divided by 8, or by the output of timer A1. Timer B has a continuously running 10-bit counter. The counter is compared against two match registers, the B1 match register and the B2 match register. When the counter counts and transitions to a value equal to a match register, a 1-peripheral clock long pulse is output. The match pulses can be used to cause interrupts and/or clock the output registers in parallel ports D and E. The match register is loaded from a preload register that can be written by an I/O instruction. The data bits are advanced from the preload register to the match register when the match pulse is output.

The I/O registers for Timer B are listed in Table 33.

**Table 33. Timer B Registers**

Register Name	Register Mnemonic	I/O Address (hex)	R/W	On Reset To
Timer B Control/Status Register	TBCSR	B0	R/W	xxxxx000
Timer B Control Register	TBCR	B1	W	xxxxxx00
Timer B MSB 1 Reg	TBM1R	B2		x
Timer B LSB 1 Reg	TBL1R	B3	W	x
Timer B MSB 2 Reg	TBM2R	B4	W	x
Timer B LSB 2 Reg	TBL2R	B5	W	x
Timer B Count MSB Reg	TBCMR	BE	R	x
Timer B Count LSB Reg	TBCLR	BF	R	x

The control/status register for Timer B (TBCSR) is laid out as shown in Table 34.

**Table 34. Timer B Control and Status Register (TBCSR) (*adr* = 0B0h)**

Bits 7:3	Bit 2	Bit 1	Bit 0
Not used	1—A match with match register 2 was detected. This bit is cleared by a read of this register; a write to this register enables the interrupt.	1—A match with match register 1 was detected. This bit is cleared by a read of this register; a write to this register enables the interrupt.	1—Enable the main clock for this timer.

The control register for Timer B (TBCR) is laid out as shown in Table 35.

**Table 35. Timer B Control Register (TBCR)**

Bits 7:4	Bits 3:2	Bits 1:0
Not used	00—Counter clocked by perclk/2 01—Counter clocked by output of timer A1 1x—Timer clocked by perclk/2 divided by 8	00—Interrupt disabled xx—Interrupt priority xx enabled.

The MSB *x* registers for Timer B (TBM1R/TBM2R) are laid out as shown in Table 36.

**Table 36. Timer B MSB *x* Register (TBM1R/TBM2R = 0B2h/0B4h)**

Bits 7:6	Bits 5:0
Two most significant bits of timer match preload register.	Not used.

### 11.2.1 Using Timer B

Normally the prescaler is set to divide perclk/2 by a number that provides a counting rate appropriate to the problem. For example, if the clock is 22.1184 MHz, then perclk/2 is 11.0592 MHz. If the prescaler is loaded with 127, it will divide the input clock by 128, and the counting rate will be 86.4 kHz, which will cause a complete cycle of the 10-bit clock in 11.85 ms. Each count will take 11.57  $\mu$ s.

Normally an interrupt will occur when either of the comparators in Timer B generates a pulse. The interrupt routine must detect which comparator is responsible for the interrupt and dispatch the interrupt to a service routine. The service routine sets up the next match value, which will become the match value after the next interrupt. If the clocked parallel ports are being used, then a value will normally be loaded into some bits of the parallel port register. These bits will become the output bits on the next match pulse. (It is necessary to keep a shadow register for the parallel port unless the bit-addressable feature of ports D and E is used.)

If it is desired to read the time from the Timer B counter, either during an interrupt caused by the match pulse or in some other interrupt routine asynchronous to the match pulse, a special procedure needs to be used to read the counter because the upper 2 bits are in a different register than the lower 8 bits. The following method is suggested.

1. Read the lower 8 bits.
2. Read the upper 2 bits
3. Read the lower 8 bits again
4. If bit 7 changed from 1 to 0 between the first and second read of the lower 8 bits there has been a carry to the upper 2 bits. In this case read the upper 2 bits again and decrement those 2 bits to get the correct upper 2 bits. Use the first read of the lower 8 bits.

This procedure assumes that the time between reads can be guaranteed to be less than 256 counts. This can be guaranteed in most systems by disabling the priority 1 interrupts, which will normally be disabled in any case in an interrupt routine.

It is inadvisable to disable the high-priority interrupts (levels 2 and 3) as that defeats their purpose.

If speed is critical, the three reads of the registers can be performed without testing for the carry. The three register values can be saved and the carry test can be performed by a lower priority analysis routine. Since the upper 2 bits are in the register TBCMR at address 0BEh, and the lower 8 bits are in TBCLR at address 0BFh, both registers can be read with a single 16-bit I/O instruction. The following sequence illustrates how the registers could be captured.

```
; enter from external interrupt on pulse input transition
; 19 clocks latency plus 10 clocks interrupt execution
push af ; 7
push hl
ioi ld a,(TBCLR) ; 11 get lower 8 bits of counter
ioi ld hl,(TBCMR) ;13 get l=upper, h=lower
```

Timer B can be used for various purposes. The 10-bit counter can be read to record the time at which an event takes place. If the event creates an interrupt, the timer can be read in the interrupt routine. The known time of execution of the interrupt routine can be subtracted. The variable interrupt latency is then the uncertainty in the event time. This can be as little 19 clocks if the interrupt is the highest priority interrupt. If the system clock is 20 MHz, the counter can count as fast as 10 MHz. The uncertainty in a pulse width measurement can be nearly as low as 38 clocks (2 x 19), or about 2  $\mu$ s for a 20 MHz system clock.

Timer B can be used to change a parallel port output register at a particular specified time in the future. A pulse train with edges at arbitrary times can be generated with the restriction that two adjacent edges cannot be too close to each other since an interrupt must be serviced after each edge to set up the time for the next edge. This restriction limits the minimum pulse width to about 5  $\mu$ s, depending on the clock speed and interrupt priorities.

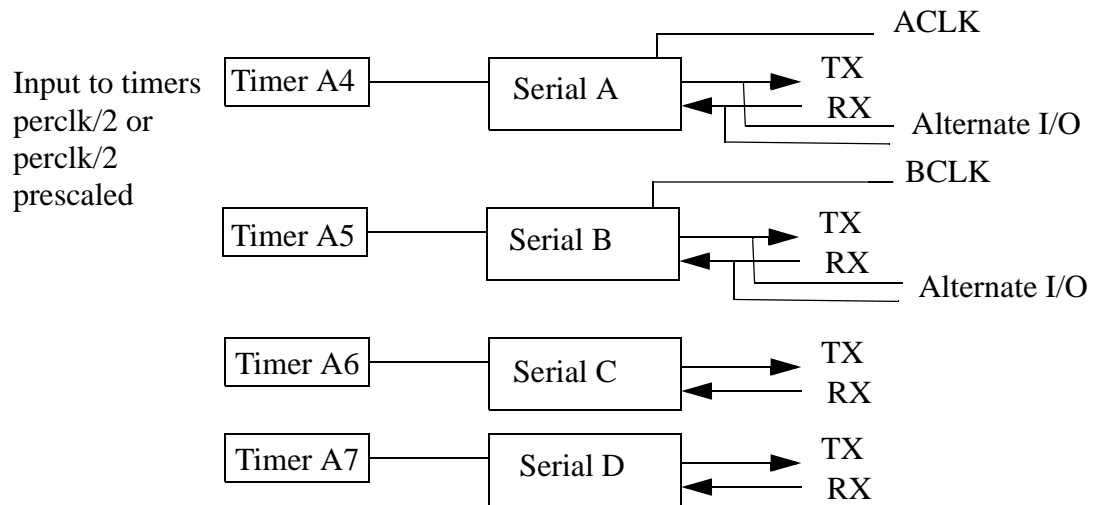




## 12. Rabbit Serial Ports

The Rabbit has four on-chip serial ports designated A, B, C and D. All the ports can perform asynchronous serial communications at high baud rates. Ports A and B have the additional capabilities of being able to operate as clocked ports and of being switchable to alternate I/O pins. Port A has the special capability of being usable to perform a cold boot of the microprocessor system.

Figure 29 shows a block diagram of the serial ports.

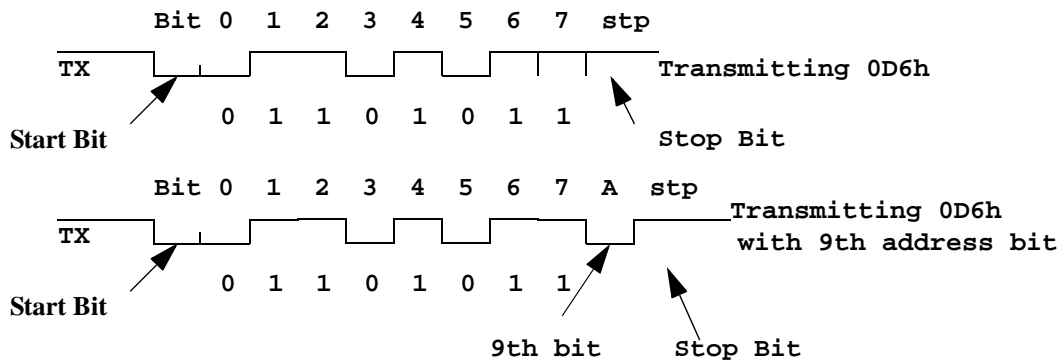
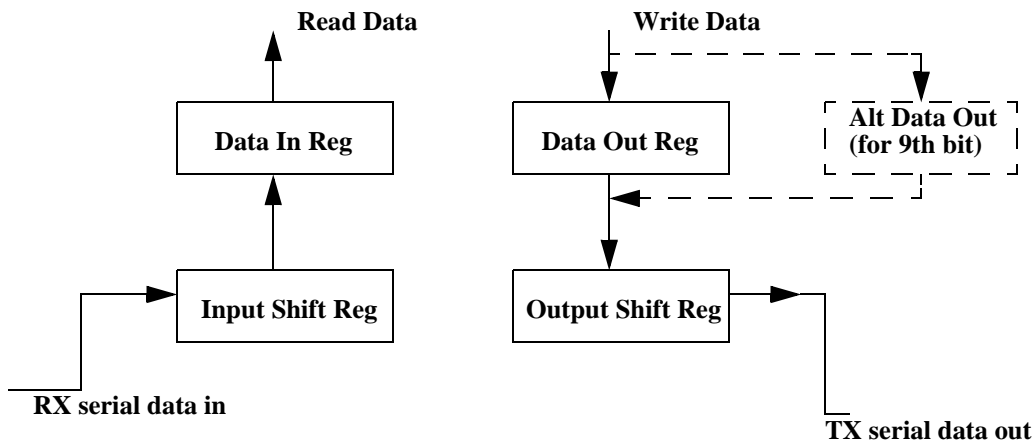


**Figure 29. Block Diagram of Rabbit Serial Ports**

The individual serial ports are capable of operating at baud rates in excess of 500,000 bps for asynchronous mode and 8 times faster than that for synchronous mode. In asynchronous mode either 7 or 8 data bits may be transmitted and received. The so called "9th" bit or address bit mode of operation is also supported. Parity and multiple stop bits are not directly supported by the hardware, but may be accomplished by suitable programming techniques.

### 12.1 Register Layout Serial Port

Figure 30 shows a functional block diagram of a serial port. Each serial port has a data register, a control register and a status register. Writing to the data register starts transmission. If the write is performed to an alternate data register address, the extra address bit or 9th bit is sent. When data bits have been received, they are read from the data register. The control register is used to set the transmit and receive parameters. The status register may be tested to check various information about the operation of the serial port.



Signals Shown At Microprocessor TX pin

**Figure 30. Functional Block Diagram of a Serial Port**

The clock input to the serial port unit must be 16 times the baud rate in asynchronous mode and 2 times the baud rate for the clocked serial mode when the internal clock is used. Timers A4–A7 supply the input clock for serial ports A–F. These timers can divide the frequency by any number from 1 to 256 (see Chapter 11). The input frequency to the timers can be selected in different ways described in the documentation for the timers. One choice is the peripheral clock divided by 2—with that choice and a well-chosen crystal frequency for the main oscillator, most commonly used baud rates can be obtained down to approximately 2400 bps at the highest Rabbit clock frequencies (see Section A.2 in Appendix A).

Table 37 lists the serial port registers.

**Table 37. Serial Port Registers**

Register	Address xx = 00, 01, 10, 11 for A, B, C, D	Mnemonic x = A, B, C, D
Data Register	11xx0000	SxDR
Alternate Data Register to Send 9th (8th) Address Bit.	11xx0001	SxAR
Status Register (read, write to clear transmit IRQ)	11xx0011	SxSR
Control Register (write only)	11xx0100	SxCR

The serial port interrupt vectors are shown Table 15 in Chapter 7.

Table 38 describes the serial port status registers.

**Table 38. Serial Port Status Registers (adr = 11xx0011, xx = A,B,C,D)**

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1,0
Receiver ready	9th (8th) bit received	Overrun	0	Transmitter data buffer empty	Transmitter busy	0,0

Writing to the status register clears the transmit interrupt request FF but has no other effect.

Bit 7—receiver ready. This bit is set when a byte is transferred from the receiver shift register to the receiver data register. The bit is cleared when the receiver data register is read. The transition from "0" to "1" sets the receiver interrupt request flip-flop.

Bit 6—address bit or 9th (8th) bit. This bit is set if the character in the receiver data register has a 9th (8th) bit. This bit is cleared should be checked before reading data register since when the data register is read a new data value may be immediately loaded with a new address bit.

Bit 5—this bit is set if the receiver is overrun. This happens if the shift register and the data register are full and a start bit is detected. This bit is cleared when the receiver data register is read.

Bit 3—transmitter data buffer empty. This bit is set when a bit is transferred from the transmitter data buffer to the transmitter shift register. It is cleared when a byte is stored in the transmitter data buffer or a write operation is performed to the serial port status register. This bit will request an interrupt on the transition from 0 to 1 if interrupts are enabled.

Bit 2—transmitter busy bit. This bit is set if the transmitter shift register is busy sending data. It is set on the falling edge of the start bit, which is also the clock edge that transfers data from the transmitter data register to the transmitter shift register. The transmitter busy bit is cleared at the end of the stop bit of the character sent. This bit will cause an interrupt to be latched when it goes from busy to not busy status after the last character has been sent (there are no more data in the transmitter data register).

Bits 0,1,4—Always read as zero.

Table 39 describes the serial port control registers.

**Table 39. Serial Port Control Registers (*adr = 11xx0100, xx = A,B,C,D*)**

Bit 7,6	Bit 5,4	Bit 3,2	Bit 1,0
00—no op	00—use port C for serial I/O	00—async mode, 8 bits	00—no interrupt
01—receive 1 byte clocked mode (A,B)	01—use port D for serial I/O (A, B)	01—async mode 7 bits	01— priority 1 interrupt
10—send one byte clocked mode (A,B)	1x—disable receiver input	10—clocked mode external clock (A,B)	10—priority 2
11—reserved for future use		11—clocked mode internal clock (A,B)	11—priority 3

Bits 7,6—In asynchronous mode, always store zero in these bits. For ports A and B, if the clocked serial mode is enabled, store the code here to start an operation, either receive or send. If the clock is internal, a burst of 8 clocks will drive the clock line. In external mode the receiver or transmitter waits for an externally supplied burst of 8 clocks.

Bits 5,4—This enables the standard or alternate I/O pins for the ports. Alternate I/O applies only to ports A and B. The parallel port output function for the specified TX pin becomes disabled when the port is enabled.

Bits 3,2—This sets the mode of operation. Modes 10 and 11 apply only to ports A and B.

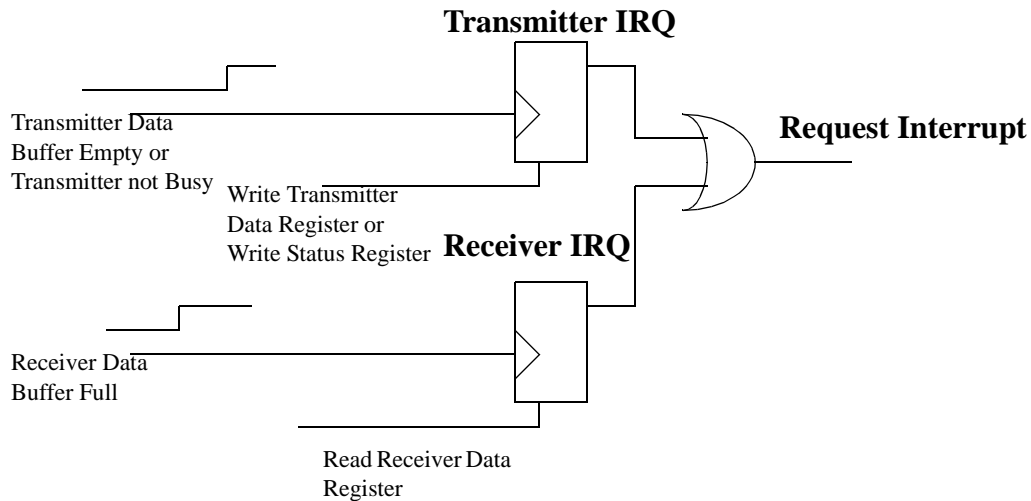
Bits 1,0—These bits enable interrupts and set the interrupt priority.

## 12.2 Serial Port Interrupt

A common interrupt vector is used for the receive and transmit interrupts. There is a separate interrupt request flip-flop for the receiver and transmitter. If either of these flip-flops is set, a serial port interrupt is requested. The flip-flops are set by a rising edge only. The flip-flops are cleared by a pulse generated by an I/O read or write operation as shown in Figure 31. When an interrupt is requested, it will take place immediately when priorities allow and an instruction execution is complete. The interrupt is lost if the request flip-flop is cleared before the interrupt takes place. If the flip-flop is not cleared in the interrupt, another interrupt will take place when priorities are lowered.

The receive interrupt request flip-flop is set after the stop bit is sampled on receive, nominally 1/2 of the way through the stop bit. data bits are transferred on this same clock from the receive shift register to the receive data register.

The transmit interrupt request flip-flop is set on the trailing edge of the stop bit. This edge is the same whether the source of the interrupt is transmitter data register empty or transmitter shift register idle (not busy). Unless the data register is empty on this edge the transmitter does not become idle. The transmitter become idle only if the data register is empty at the trailing edge of the stop bit.



**Figure 31. Generation of Serial Port Interrupts**

The serial port interrupt vectors are shown Table 15 in Chapter 7.

### 12.3 Transmit Serial Data Timing

On transmit, if the interrupts are enabled, an interrupt is requested when the transmit register becomes empty and, in addition, an interrupt occurs when the shift register and transmit register both become empty, that is, when the transmitter becomes idle. When the transmit data register contains data and the shift register finishes sending data, the data bits are clocked from the transmit register to the shift register, and the shift register is never idle. The interrupt request is cleared either by writing to the data register or by writing to the status register (which does not affect the status register). The data register normally is clocked into the shift register each time the shift register finishes sending data, leaving the data register empty. This causes an interrupt request. The interrupt routine normally answers the interrupt before the shift register runs dry (9 to 11 baud clocks, depending on the mode of operation). The interrupt routine stores the next data item in the data register, clearing the interrupt request and supplying the next data bits to be sent. When all the characters have been sent, the interrupt service routine answers the interrupt once the data register becomes empty. Since it has no more data, it clears the interrupt request by storing to the status register. At this point the routine should check if the shift register is empty; normally it won't be. If it is, because the interrupt was answered late, the interrupt routine should do any final cleanup and store to the status register again in case the shift register became empty after the pending interrupt is cleared. Normally, though, the interrupt service routine will return and there will be a final interrupt to give the routine a chance to disable the output buffers, as in the case for RS-485 transmission.

## 12.4 Receive Serial Data Timing

When the receiver is ready to receive data, a falling edge indicates that a start bit must be detected. The falling edge is detected as a different Rx input between two different clocks, the clock being 16x the baud rate. Once the start bit has been detected, data bits are sampled at the middle of each data bit and are shifted into the receive shift register. After 7 or 8 data bits have been received, the next bit will be either a 9th (8th) address bit, or a stop bit will be sampled. If the RX line is low, it is an address bit and the address bit received bit in the status register will be enabled. If an address bit is detected, the receiver will attempt to sample the stop bit. If the line is high when sampled, it is a stop bit a new scan for a new start bit will begin after the sample point. At the same time, the data bits are transferred into the receive data register and an interrupt, if enabled, is requested.

On receive an interrupt is requested when the receiver data register has data. This happens when data bits are transferred from the receive shift register to the data register. This also sets bit 7 of the status register. The interrupt request and bit 7 are cleared when the data register is read.

An interrupt is requested if bit 7 is high. The interrupt is requested on the edge of the transmitter data register becoming empty or the transmitter shift register becoming empty. The transmitter interrupt is cleared by writing to the status register or to the data register.

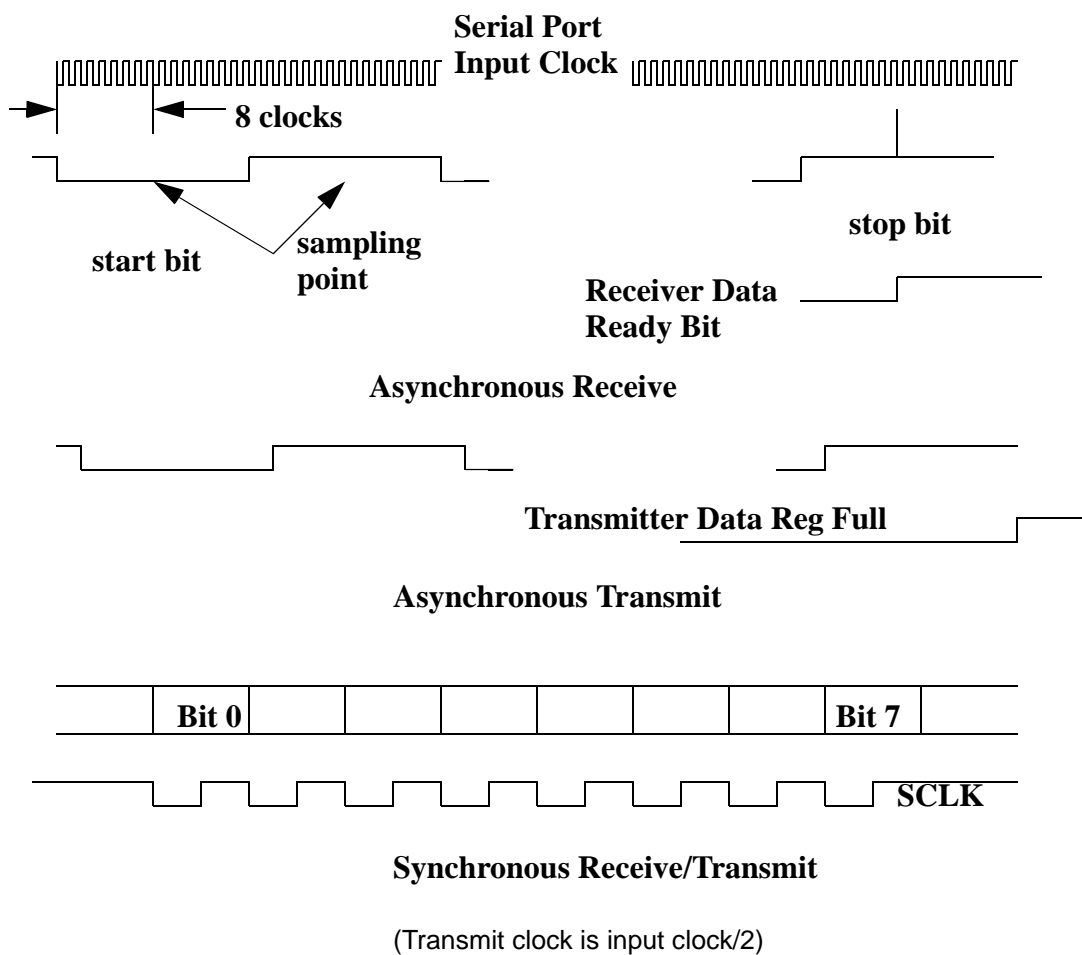
On receive, the scan for the next start bit starts immediately after the stop bit is detected. The stop bit is normally detected at a sample clock that nominally occurs in the center of the stop bit. If there is a 9th (8th) address bit, the stop bit follows that bit.

## 12.5 Clocked Serial Ports

Ports A and B can operate in clocked mode. The data line and clock line are driven as shown in Figure 32. The data and clock are provided as 8-bit bursts. The transmit shift register advances on the falling edge of the clock. The receiver samples the data on the rising edge of the clock. Clocked serial communication is half-duplex. A clocked serial port can transmit or receive, but not both at the same time. The serial port can generate the clock or the clock can be provided externally.

To enable the clocked serial mode, a code must be in bits (3,2) of the control register, enabling the clocked serial mode with either an internal clock or an external clock. The transition between the external and the internal clock should be performed with care. Normally a pullup resistor is needed on the clock line to prevent spurious clocks while neither party is driving the clock.

In clocked serial mode the shift register and the data register work in the same fashion as for asynchronous communications. However, to initiate sending or receiving, a code must be stored in bits (7,6) of the control register for each byte sent or received. One code specifies sending a byte, a different code specifies receiving a byte. The effect of these codes is different, depending on whether the mode is internal clock or external clock.



**Figure 32. Serial Port Synchronization**

To transmit in internal clock mode the user must first load the data register (which must be empty) and then store the send code. When the shift register is finished sending the current character, if any, the data register will be loaded into the shift register and transmitted by an 8-clock burst. One character can be in the process of transmitting while another character is waiting in the data register tagged with the send code. The send code is effectively double-buffered.

To receive a character in internal clock mode, the receive shift register should be idle. The user then stores the receive code in the control register. A burst of 8 clocks will be generated and the sender must detect the clocks and shift output data to the data line on the falling edge of each clock. The receiver will sample the data on the rising edge of each clock. The receive mode cannot double-buffer characters when using the internal clock. The shift register must be idle before another character receive can be initiated. However, the interrupt request and character ready takes place on the rising edge of the last clock pulse. If the next receive code is stored before the natural location of the next falling edge, another receive will be initiated without pausing the clock. To do this, the interrupt has to be serviced within 1/2 clock.

To transmit each byte in external clock mode, the user must load the data register and then store the send code. When the shift register is idle and the receiver provides a clock burst, the data bits are transferred to the shift register and are shifted out. Once the transfer is made to the shift register, a new byte can be loaded into the transmit register and a new send code can be stored.

To receive a byte in external clock mode, the user must set the receive code for the first byte and then store the receive code for the next byte after each byte is removed from the data register. Since the receive code must be stored before the transmitter sends the next byte, the receiver must service the interrupt within 1/2 baud clock to maintain full-speed transmission. This is usually not practical unless a flow control arrangement is made or the transmitter inserts gaps between the clock bursts.

In order to carry on high-speed communication, the best arrangement will usually be for the receiver to provide the clock. When the receiver provides the clock, the transmitter should always be able to keep up because it is double-buffered and has a full character time to answer the transmitter data register empty interrupt. The receiver will answer interrupts that are generated on the last clock rising edge. If the interrupt can be serviced within 1/2 clock, there will be no pause in the data rate. If it takes the receiver longer to answer, then there will be a gap between bytes, the length of which depends on the interrupt latency. For example, if the baud rate is 400,000 bps, then up to 50,000 bytes per second could be transmitted, or a byte every 20  $\mu$ s. No data will be lost if the transmitter can answer its interrupts within 20  $\mu$ s. There will be no slow down if the receiver can answer its interrupt within 1/2 clock or 1.25  $\mu$ s. If it can answer within 1.5 clocks, or 2.75  $\mu$ s, the data rate will slow to 44,444 bytes per second. If it can answer in 2.5 clocks or 6.25  $\mu$ s, the data rate slows to 40,000 bytes per second. If it can answer in 3.5 clocks or 8.75  $\mu$ s, the data rate will slow to 36,363 bytes per second, and so forth.

If two-way half-duplex communication is desired, the clock can be turned around so that the receiver always provides the clock. This is slightly more complicated since the receiver cannot initiate a message. If the receiver attempts to receive a character and the transmitter is not transmitting, the last bit sent will be received for all eight bits.

## 12.6 Serial Port Software Suggestions

The receiver and transmitter share the same interrupt vector, but it is possible to make the receive and transmit interrupt service routines (ISRs) separate by dispatching the interrupt to either of two different routines. This is desirable to make the ISR less complex and to reduce the interrupt off time. No interrupts will be lost since distinct interrupt flip-flops exist for receive and transmit. The dispatcher can test the receiver data register full bit to dispatch. If this bit is on, the interrupt is dispatched for receive, otherwise for transmit. The receiver receives first consideration because it must be serviced attentively or data could be lost.



The dispatcher might look as follows.

```
interrupt:
    push af ; 10
    ioe ld a,(SCSR) ; 7 get status register serial port C
    or a,a ; 2 test sign bit
    jp m,receive ; 7 go service the receive interrupt
    jp transmit ; 7 (41 clocks to here )go service transmit interrupt
```

The individual interrupts would assume that register AF has been saved and the status register loaded into register A.

The interrupt service routines can, as a matter of good practice and obtaining optimum performance, remove the cause of the interrupt and re-enable the interrupts as soon as possible. This keeps the interrupt latency down and allows the fastest transmission speed on all serial ports. Serial ports normally will all generate priority level 1 interrupts. The exception would be if a port must operate at extremely high speed. At 115,200 bps, the highest speed of PC serial ports, the interrupts must be serviced in 10 baud times or 86  $\mu$ s in order to not lose received characters. If all four serial ports were operating at this receive speed, it would be necessary to service the interrupt in less than 21.5  $\mu$ s to assure no lost characters. In addition, the time taken by other interrupts of equal or higher priority would have to be considered. A receiver service routine might appear as follows below. The byte at `bufptr` is used to address the buffer where data bits are stored. It is necessary to save and increment this byte because characters could be handled out of order if two receiver interrupts take place in quick succession.

```
receive:
    push hl ; 10 save hl
    push de ; 10 save de
    ld hl,struct ; 6
    ld a,(hl) ; 5 get in-pointer
    ld e,a ; 2 save in pointer in e
    inc hl ; 2 point to out-pointer
    cmp a,(hl) ; 5 see if in-pointer=out-pointer (buffer full)
    jr z,roverrun ; 5 go fix up receiver over run
    inc a ; 2 increment the in pointer
    and a,mask ; 4 mask such as 11110000 if 16 buffer locs
    dec hl ; 2
    ld (hl),a ; 6 update the in pointer
    ioe ld a,(SCDR) ; 11 get data register port C, clears interrupt
    request
    ipres ; 4 restore the interrupt priority

    ; 68 clocks to here
    ; to level before interrupt took place
    ; more interrupts could now take place,
    ; but receiver data is in registers
    ; now handle the rest of the receiver interrupt routine
    ld hl,bufbase ; 6
    ld d,0 ; 6
    add hl,de ; 2 location to store data
    ld (hl),a ; 6 put away the data byte
    pop de ; 7
    pop hl ; 7
    pop af ; 7
    ret ; 8 from interrupt

    ; 117 clocks to here
```

This routine gets the interrupts turned on in about 68 clocks or 3.5  $\mu$ s at a clock speed of 20 MHz. Although two characters may be handled out of order, this will be invisible to a higher level routine checking the status of the input buffer because all the interrupts will be completed before the higher level routine can perform a check on the buffer status.

A typical way to organize the buffers is to have an in-pointer and an out-pointer that increment through the addresses in the data buffer in a circular manner. The interrupt routine manipulates the in-pointer and the higher level routine manipulates the out-pointer. If the in-pointer equals the out-pointer, the buffer is considered full. If the out-pointer plus 1 equals the in-pointer, the buffer is empty. All increments are done in a circular fashion, most easily accomplished by making the buffer a power of two in length and and'ing a mask after the increment. The actual memory address is the pointer plus a buffer base address.

### **12.6.1 Controlling an RS-485 Driver and Receiver**

RS-485 uses a half-duplex method of communication. One station enables its driver and sends a message. After the message is complete, the station disables the driver and listens to the line for a reply. The driver must be enabled before the start bit is sent and not disabled until the stop bit has been sent. The transmitter idle interrupt is normally used to disable the RS-485 driver and possibly enable the receiver.

### **12.6.2 Transmitting Dummy Characters**

It may be desired to operate the serial transmitter without actually sending any data. The output of the transmitter may be disconnected from the transmitter by manipulating the control registers for parallel port C or D, which are used as output pins. For example, if serial port B is to be temporarily disconnected from its output pin, which is bit 4 of parallel port C, this can be done as follows.

1. Store a "1" in bit 4 of the parallel port data output register to provide the quiescent state of the drive line.
2. Clear bit 4 of the parallel port C function register so that the output no longer comes from the serial port. Of course, this should not be done until the transmitter is idle.

### **12.6.3 Transmitting and Detecting a Break**

A break is created when the output of the transmitter is driven low for an extended period. If a break is received, it will appear as a series of characters filled with zeros and with the 9th bit enabled. A break can be transmitted by transmitting a byte of zeros at a very slow baud rate. Another method is to disconnect the transmitter and use the parallel port bit to set the line low while sending dummy characters to time out the break.

### 12.6.4 Using A Serial Port to Generate a Periodic Interrupt

A serial port may be used to generate a periodic interrupt by continuously transmitting characters. Since the TX output via parallel port C or D can be disabled, the transmitted characters are transmitted to nowhere. Because the character output path is double-buffered, there will be no gaps in the character transmission, and the interrupts will be exactly periodic. The interrupts can happen every 9, 10 or 11 baud times, depending on whether 7 or 8 bits are transmitted and on whether the 9th (8th) bit is sent.

### 12.6.5 Working With Two Stop Bits or a Parity Bit

In the 8-bit data mode, the serial ports do not directly support transmitting more than one stop bit. If it is desired to send more than one stop bit, it is necessary to delay the transmission of the next character for at least 1 baud time. If only 7 data bits are being sent, the problem is easily solved by sending 8 bits and always setting bit 7 of the byte to a "1" if an extra stop bit is desired, or setting it to the parity value if a parity bit is desired. No special precautions are needed if two stop bits are to be received. If parity is received with 7 data bits, receive the data as 8 bits, and the parity will be in the high bit of the byte.

The scenario is more complicated when 8 data bits are used. If a parity bit is to be received, it will appear as an address or 9th bit if a zero is transmitted for the parity bit. Otherwise the parity bit appears as an additional stop bit and the 9th bit is not detected. Transmitting a parity bit is the same as sending a 9th bit if the parity bit is zero; otherwise it is the same as sending an additional stop bit.

The only difficult thing to do is to send an additional stop bit when transmitting 8 data bits. This requires leaving the transmitter idle for one baud time, or longer, before the next character is sent. One way to do this is to use another serial port as a timer. Disable the interrupts on the port being used to transmit and, at the same time the data register is loaded, load a dummy character and a 9th bit in the other serial port. The interrupt in the auxiliary port will occur after 11 baud times rather than 10 baud times, thus guaranteeing the stop bit its full time. Another way is to send a full dummy character to create a very long stop bit. To avoid the long stop bit, the baud timer can be speeded up while the dummy character is sent to reduce the length of the extra stop bit. This method will work as long as the baud rate is low enough that it can be speeded up by a factor of 10 to send the dummy character. The synchronous nature of timers A4–A7 allows the divide ratio to be increased or decreased at will without generating irregular clock pulses. Yet another method is to use a timer to generate the extra 1-baud delay between characters.

### 12.6.6 Data Framing/Modbus

Some protocols, for example, Modbus, depend on a gap in the data frame to detect the beginning of the next frame. The 9th bit protocol is another way to detect the start of a data frame.

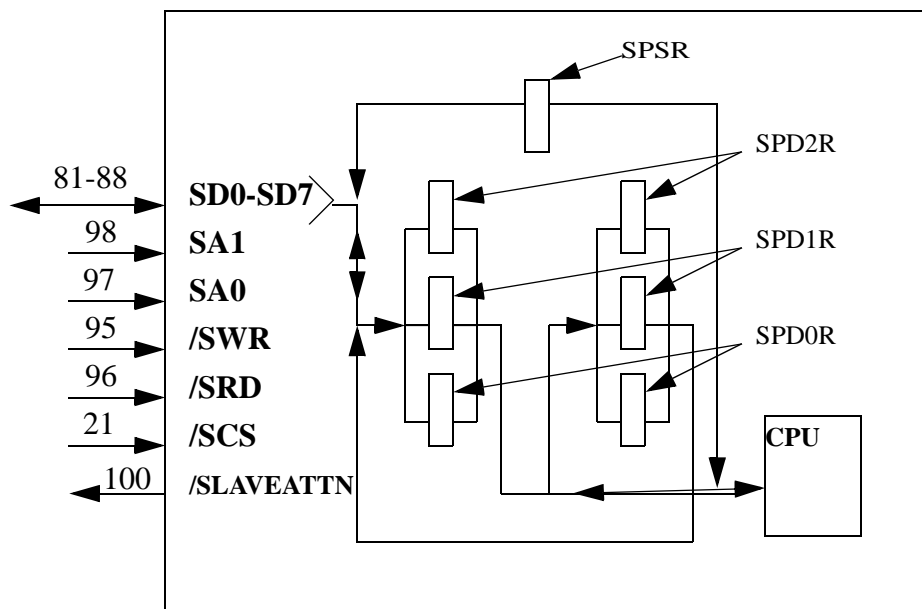
The Modbus protocol requires that data frames begin with a minimum 3.5-character quiet time. The receiver uses this 3.5-character gap to detect the start of a frame. In order for

the receiving ISR to detect this gap, it is suggested that dummy characters be transmitted to aid in detecting the gap. This can be done in the following manner. The transmitter starts transmitting dummy characters when the first character interrupt is received. Each time there is an interrupt, either receiver data register full or transmitter data register empty, a dummy character is transmitted if the transmitter data register is empty. Although the transmitter and receiver operate at approximately the same baud rate, there can be a difference of up to about 5% between their baud rates. Thus the receiver full and transmitter empty interrupts will become out of phase with each other, assuming that the remote station transmits without gaps between characters. A counter is zeroed each time a character is received, and the counter is incremented each time a character is transmitted. If this counter holds (n), this indicates that a gap has been detected in the frame; the length of the gap is (n-1) to (n) characters. The start of frame could be marked by (n) reaching 3, indicating that the existence of a gap at least two characters long.

### 13. Rabbit Slave Port

When a Rabbit microprocessor is configured as a slave, parallel port A and certain other data lines are used as communication lines between the *slave* and the *master*. The slave unit is a Rabbit configured as a slave. The master can be another Rabbit or any other type of processor. Rabbits configured as slaves can themselves have slaves.

The master and slave communicate with each other via the slave port. The slave port is a physical device that includes data registers, a data bus and various handshaking lines. The slave port is a part of the slave Rabbit, but logically it is an independent device that is used to communicate between the two processors. A diagram of the slave port is shown in Figure 33.

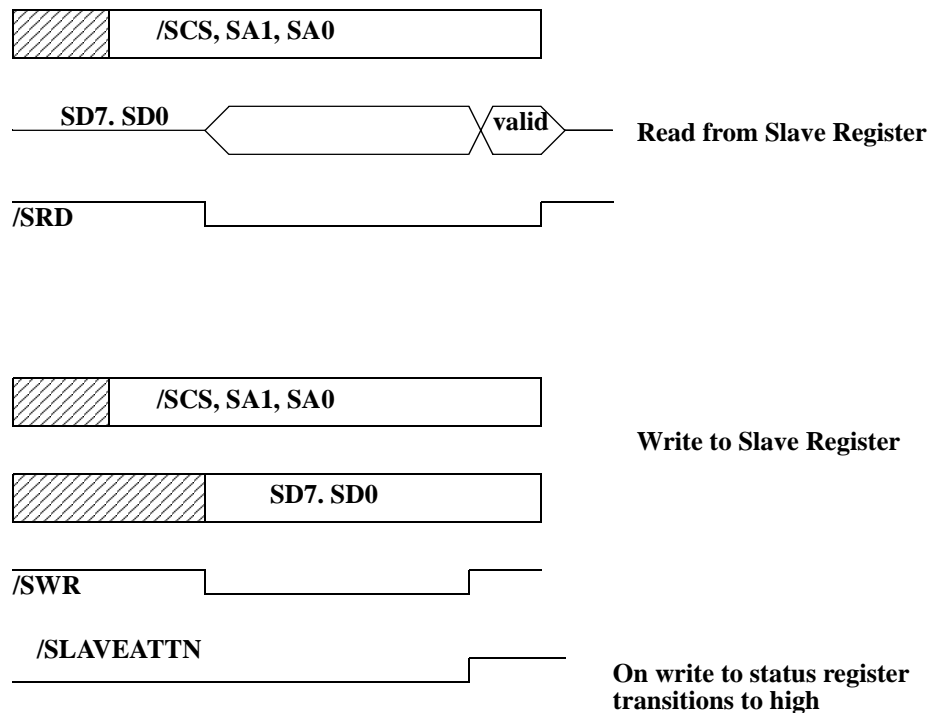


**Figure 33. Rabbit Slave Port**

The slave port has three data registers for each direction of communication. Three registers, named SPD0R, SPD1R, and SPD2R, can be written by the master and read by the slave. Three different registers, also named SPD0R, SPD1R, and SPD2R, can be written by the master and read by the slave. The same names are used for different registers since it is usually clear from the context which register is meant. If it is necessary to distinguish between registers, we will refer to the registers as SPD0R writable by the slave or SPD0R readable by the master, different descriptions for the same register.

A status register can be read by either the slave or the master. The status register has full/empty bits for each of the six registers. A data register is considered *full* when it is written to by whichever side is capable of writing to it. If the same register is then read by either side it is considered to be *empty*. The flag for that register is thus set to a "1" when the register is written to, and the flag is set to a "0" when the register is read.

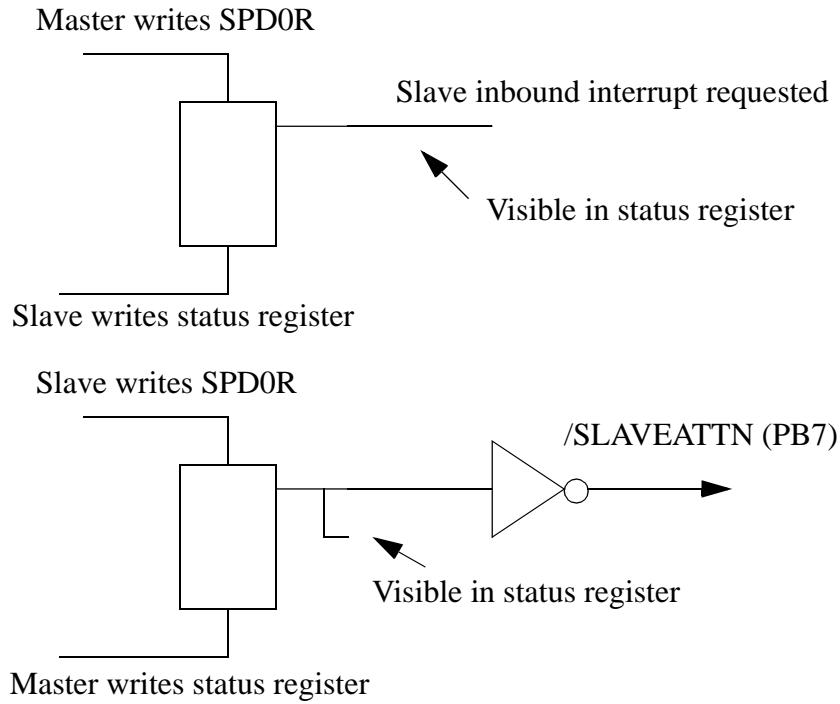
The registers appear to be internal I/O registers to the slave. To the master, at least for a Rabbit master, the registers appear to be external I/O registers. Figure 34 shows the sequence of events when the master reads/writes the slave port registers.



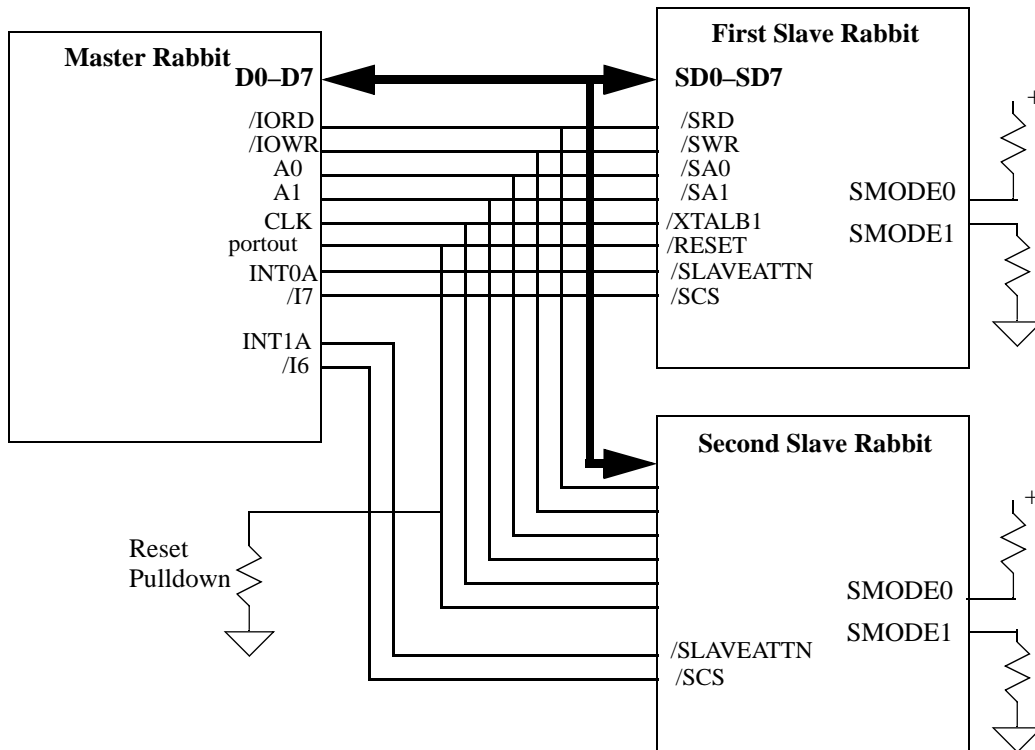
**Figure 34. Slave Port R/W Sequencing**

The two SPDOR registers have special functionality not shared by the other data registers. If the master writes to SPDOR, an inbound interrupt flip-flop is set. If slave port interrupts are enabled, the slave processor will take a slave port interrupt. If the slave writes to the other SPDOR register, the slave attention line (/SLAVEATTN, pin 100) is asserted (driven low) by the slave processor. This line can be used to create an interrupt in the master. Either side that is interrupted can clear the signal that is causing an interrupt request by writing to the slave port status register. The data bits are ignored, but the flip-flop that is the source of the interrupt request is cleared. Figure 35 shows a logical schematic of this functionality.

Figure 36 shows a sample connection of two slave Rabbits to a master Rabbit. The master drives the slave reset line for both slaves and provides the main processor clock from its own clock. There is no requirement that the master and slave share a clock, but doing so makes it unnecessary to connect a crystal to the slaves. Each Rabbit in Figure 36 has to have RAM memory. The master must also have flash memory. However, the slaves do not need nonvolatile memory since the master can cold boot them over the slave port and download their program. In order for this to happen, the SMODE0 and SMODE1 pins must be properly configured as shown in Figure 36 to begin a cold boot process at the end of the slave reset.



**Figure 35. Slave Port Handshaking and Interrupts**



**Figure 36. Typical Connection Slave Rabbit to Master Rabbit**

The slave port lines are shown in Figure 33. The function of these lines is described below.

- SD0–SD7—These are bidirectional data lines, and are generally connected to the data bus of the master processor. Multiple slaves can be connected to the data bus. The slave drives the data lines only when /SCS and /SRD are both pulled low.
- SA1, SA0—These are address lines used to select one of the four data registers of the slave interface. Normally these lines are connected to the low-order address lines of the master. The master always drives these lines which are always inputs to the slave.
- /SCS—Input. Slave chip select. The slave ignores read or write requests unless the chip select is low. If a Rabbit is used as a master, this line can be connected to one of the master's programmable chip select lines /I0–/I7.
- /SRD—Input. If /SCS is also low, this line pulled low causes the contents of the register selected by the address lines to be driven on the data bus. If a Rabbit is used as a master, this line is normally connected to the global I/O read strobe /IORD.
- /SWR—Input. If /SCS is also low, this line causes the data bits on the data bus to be clocked into the register selected by the address lines on the rising edge of /SWR or /SCS, whichever rises first. If a Rabbit is used as a master, this line is normally connected to the global I/O write strobe /IOWR.
- /SLAVEATTN—This line is set low (asserted) if the slave writes to the SPD0R register. This line is set high if the master writes anything to the slave status register. This line is usually connected to cause the master to be interrupted when it goes low.

The data lines of the slave port are shared with parallel port A that uses the same package pins. The slave port can be enabled, and parallel port A be disabled, by storing an appropriate code in the slave port control register (SCR). After the processor is reset, all the pins belonging to the slave interface are configured as parallel-port inputs unless (SMODE1, SMODE0) are set to (0,1), in which case the slave port is enabled after reset and the slave starts the cold-boot sequence using the slave port.

### 13.1 Hardware Design of Slave Port Interconnection

Figure 36 shows a typical circuit diagram for connecting two slave Rabbits to a master Rabbit. The designer has the option of cold-booting the slave and downloading the program to RAM on each cold start. Another option is to configure the slave with both RAM and flash memory. In this case, the slave will only have the program downloaded for maintenance or upgrades. Usually, the flash would not be written to on every startup because of the limited number of lifetime writes to flash memory. The slaves' reset in Figure 36 is under the program control of the master. If the master is reset, the slave will also be reset because the master's drive of the reset line will be lost on reset and the pull-down resistor will pull the slaves' resets low. This may be undesirable because it forces the slave to crash if the master crashes and has a watchdog timeout.



## 13.2 Slave Port Registers

The slave port registers are listed in Table 40. These registers, each of which is actually two separate registers, one for read and one for write, are accessible to the slave at the I/O addresses shown in the table and they are accessible to the master at the external address shown which specifies the value of the slave address (SA0, SA1) input to the slave when the master reads or writes the registers. The register that can be written by the slave can only be read by the master and vice versa. If one side were to attempt to read a register at the same time that the other side attempted to write the register the result of the read could be scrambled. However, the protocols and handshaking bits used in communication are normally such that this never happens.

**Table 40. Slave Port Registers**

Register	Mnemonic	Internal Address	External Address
Slave Port Data x Register	SPD0R	20h	0
	SPD1R	21h	1
	SPD2R	22h	2
Slave Port Status Register	SPSR	23h	3
Slave Port Control Register	SPCR	24h	N.A.

If the user for some reason wants to depart from the suggested protocols and poll a register while waiting for the other side to write something to the register, the user should be aware that all the bits might not change at the exact same time when the result changes, and a transitional value could be read from the register where some bits have changed to the new value and others have not. To avoid being confused by a transitional value, the user can read the register twice and make sure both values are the same before accepting the value, or the user can test only one bit for a change. The transitional value can only exist for one read of the register, and each bit will have its old value change to the new value at some point without wavering back and forth. The existence of a transitional value could be very rare and has the potential to create a bug that happens often enough to be serious, but so infrequently as to be difficult to diagnose. Thus, the user is cautioned to avoid this situation.

Table 41 describes the slave port control register.

**Table 41. Slave Port Control Register (SPCR) (adr = 024h)**

Bit 7 w/o	Bits 6,5 R/O	Bit 4	Bit 3,2 w/o	Bits 1,0 w/o
0—obey SMODE pins 1—ignore SMODE pins	Reads SMODE pins smode1,smode0	x	00—disable slave port, port A is a byte wide input port 01—disable slave port, port A is a byte wide output port 1x—enable the slave port	00—no slave interrupt pp—enable slave port interrupt priority 1–3.

The functionality of the bits is as follows.

Bit 7—If set to "0," the cold-boot feature will be enabled. Normally this bit is set to a "1" after the cold boot is complete. The cold boot for the slave port is enabled automatically if (SMODE1, SMODE0) lines are set to (0,1) after the reset ends. This feature disables the normal operation of the processor and causes commands to be accepted via the slave port register SPD0R. These commands cause data to be stored in memory or I/O space. When the master that is managing the cold boot has finished setting up memory and I/O space, the (SMODE1, SMODE0) pins are changed to code (0,0), which causes execution to start at address zero. Typically this will start execution of a secondary boot program. At some point, bit 7 will be set to a "1" so that the SMODEx pins can be used as normal input pins.

Bits 6,5—May be used to read the input pins SMODE , SMODE0.

Bits 3,2—Bit 3 if set to a "1" enables the slave port, disabling parallel port A and various other port lines. Bit 3 is automatically set to a "1" (?????) if a cold boot is done via the slave port. If bit 3 is "0," then bit 2 controls whether parallel port A is input (bit 2=0) or output (bit 2=1).

Bits 1,0—This 2-bit field sets the priority of the slave port interrupt. The interrupt is disabled by (0,0).

Table 42 describes the slave port status register. The status register has 6 bits that are set if the particular register is full. That means that the register has been written by the processor that can write to it but it has not been read by the processor that can read it. The bits for SPD0R are used to control the slave interrupt and the handshaking lines as shown in Figure 35.

**Table 42. Slave Port Status Register (SPSR) (adr = 023h)**

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
1—set by master write to SPD0R. Cleared by slave write to SPSR.	1—set by master write to SPD2R. Cleared when slave reads register.	1—set by master write to SPD1R. Cleared when slave reads register.	1—set by master write to SPD0R. Cleared when slave reads register.	1—set by slave write to SPD0R. Cleared by master write to SPSR.	1—set by slave write to SPD2R. Cleared when master reads register.	1—set by slave write to SPD1R. Cleared when master reads register.	1—set by master write to SPD0R. Cleared when slave reads register.

## 13.3 Applications and Communications Protocols for Slaves

The communications protocol used with the slave port depends on the application. A slave processor may be used for various reasons. Some possible applications are listed below.

Keep in mind that the Rabbit can also be operated as a slave processor via a serial port and some of the protocols will work well via a serial communications connection. If a serial connection is used, the protocol becomes more complicated if errors in transmission need to be taken into account. If the physical link can be controlled so that transmission errors do not occur, a realistic possibility if the interconnection environment is controlled, the serial protocol is simpler and faster than if error correction needs to be taken into account.

### 13.3.1 Slave Applications

- **Motion Controller**—Many types of motion control require fast action, may be compute-intensive or both. Traditional servo system solutions may be overly expensive or not work very well because of system nonlinearities. The basic communications model for a motion controller is for the master to send short messages—positioning commands—to the slave. The slave acknowledges execution of the commands and reports exception conditions.
- **Communications Protocol Processor**—Communications protocols may be very complex, may require fast responses, or may be compute-intensive.
- **Graphics Controller**—The Rabbit can be used to perform operations such as drawing geometric figures and generating characters.
- **Digital Signal Processing**—Although the Rabbit is not a speciality digital signal processor, it has enough compute speed to handle some types of jobs that might otherwise require a speciality processor. The slave processor can process data to perform pattern recognition or to extract a specific parameter from a data stream.

### 13.3.2 Master-Slave Messaging Protocol

In this protocol the master sends messages to the slave and receives an acknowledgement message. The protocol can be polled or interrupt driven. Generally, the master sends a message that has a message type code, perhaps a byte count, and the text of the message. The slave responds with a similar message as an acknowledgement. Nothing happens unless the master sends a message. The slave is not allowed to initiate a message, but the slave could signal the master by using a parallel port line other than /SLAVEATN or by placing data in one of the registers the master can read without interfering with the message protocol.

The master sends a message byte by storing it in SPDOR. The slave notices that SPDOR is full and reads the byte. When the master notices that SPDOR is empty because the slave read it, the master stores the next byte in SPDOR. Either side can tell if any register is empty or full by reading the status register. When the slave acknowledges the message with a reply message, the process is reversed. To perform the protocol with interrupts, a

slave interrupt can be generated each time the slave receives a character. The slave can acknowledge the master by reading SPDOR if the master is polling for the slave response to each character. If the master is to be interrupted to acknowledge each character, the slave can create an interrupt in the master by storing a dummy character in SPDOR to create a master interrupt, assuming that the /SLAVEATTN line is wired to interrupt the master. The acknowledgement message works in a similar manner, except that the master writes a dummy character to interrupt the slave to say that it has the character.

Several problems can arise if there are dual interrupts for each character transmitted. One problem is that the message transmission rate will free run at a speed limited by the interrupt latency and compute speed of each processor. This could consume a high percentage of the compute resources of one or both processors, starving other processes and especially interrupt routines, for compute time. If this is a problem, then a timed interrupt can be used to drive the process on one side, thus limiting the data transmission rate.

Another solution, which may be better than limiting the transmission rate, is to use interrupts only for the first byte of the message on the slave side, and then lower the interrupt priority and conduct the rest of the transaction as a polled transaction. On the master side the entire transaction can be a polled transaction. In this case, the entire transaction takes place in the interrupt routine on the slave, but other interrupts are not inhibited since the priority has been lowered.

A typical slave system consists of a Rabbit microprocessor and a RAM memory connected to it. The clock can be provided either by connecting a crystal, or crystals to the slave or by providing an external clock, which could be the master's clock. The reset line of the slave would normally be driven by the master. At system startup time the master resets the slave and cold boots it via the slave port. (The SMODE pins must be configured for this.) Once the software is loaded into the slave, the slave can begin to perform its function.

As a simple example, suppose that the slave is to be used as a four-port UART. It has the capability to send or receive characters on any of its four serial ports. Leaving aside the question of setup for parameters such as the baud rate, we could define a protocol as follows.

SPDOR readable by master is a status register with bits indicating which of the four receivers and four transmitters is ready, that is, has a character received or is ready to send a character.

SPDOR writable by the master is a control register used to send commands to the slave.

SPD1R is used to send or receive data characters or control bytes.

The line /SDATAVL is wired to the external interrupt request of the master so that the master is interrupted when the slave writes to SPDOR. Typically the slave will write to SPDOR when there is a change of status on one of the serial ports.

The line /SDATRDY is wired to a port input line on the master so that the master can tell if the slave has accepted the command written to SPDOR.

The slave can interrupt the master at any time by storing to SPD0R. It will do this every time an enabled transmitter is ready to accept a character or every time an enabled receiver receives a character. When it stores to SPD0R, it will store a code indicating the reason for the interrupt, that is, receive or transmit and channel number. If the cause is receive, the received character will also be placed in SPD1R writable by the slave. When the master is interrupted for any reason, the master will sneak a peek at SPD0R by reading SPSR. If the interrupt is caused by a receive character, it will remove the character from SPD1R and read SPD0R to handshake with the slave.

If the master is interrupted for transmitter ready, as determined by the sneak peek, it will place the outgoing character in SPD1R and write a code to SPD0R indicating transmit and channel number. This will cause the slave to be interrupted, and the slave will take the character and handshake by reading SPD0R. This handshake does not interrupt the master.



## 14. Rabbit Hardware Design and Development

Core designs and printed-circuit board layouts are available on the Rabbit web site. A core design includes memory, the microprocessor, oscillator crystals, the Rabbit standard programming port, and in some cases a power controller and power supply. Although modern designs usually use at least four-layer printed circuit boards, two-sided boards are a viable option with the Rabbit, especially if the clock speed is not high and the design is intended to operate at 2.5 V or 3.3 V—actors which reduce edge speed and electromagnetic radiation.

### 14.1 RS-485 Communication Interface

To be added in the future.

### 14.2 RS-232 Communication Interface

To be added in the future.

### 14.3 Analog-to-Digital Converters

To be added in the future.

### 14.4 Digital-to-Analog Converters

To be added in the future.

### 14.5 High-Voltage Drivers

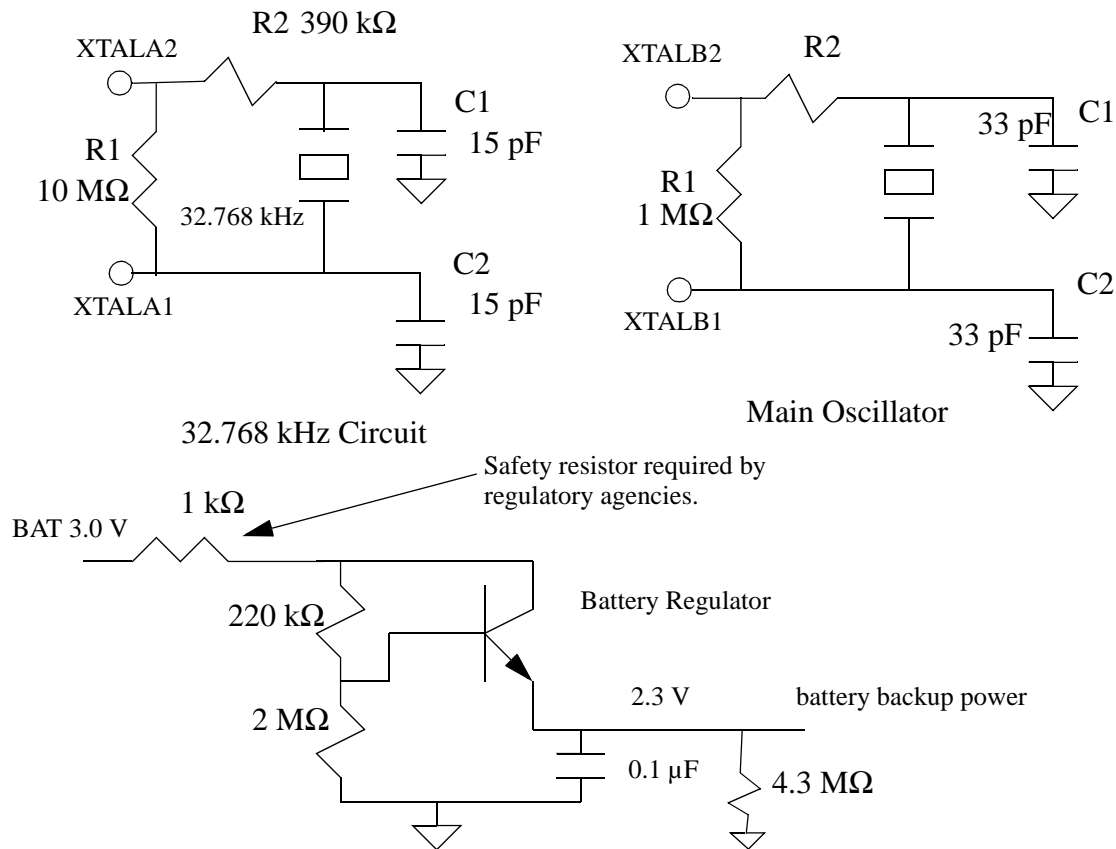
To be added in the future.

### 14.6 Clocks

The Rabbit has two built in oscillators. The 32.768 kHz clock oscillator is needed for the battery-backable clock, the watchdog timer, and the cold boot function. The main oscillator provides the run time clock for the microprocessor.

The 32.768 kHz oscillator is slow to start oscillating after power-on. For this reason a wait loop in the BIOS waits until this oscillator is oscillating regularly before continuing the startup procedure. If the clock is battery-backed, there will be no startup delay since the oscillator is already oscillating. The startup delay may be as much as 5 seconds. Crystals with low series resistance ( $R < 35 \text{ k}\Omega$ ) will start faster. The required oscillator circuit is shown in Figure 37. If current consumption by the real-time clock is important, the regulator circuit shown in the figure below will reduce the current consumption by a substantial amount when a 3 V lithium battery is used. Using this circuit the battery backed clock

will require less than 25  $\mu\text{A}$ . If the full 3 V is used the current consumption will be approximately 70  $\mu\text{A}$ .



**Figure 37. Oscillator Circuits**

## 14.7 Low-Power Design

The power consumption is proportional to the clock frequency and to the square of the operating voltage. Thus, operating at 3.3 V instead of 5 V will reduce the power consumption by a factor of  $10.9/25$  or 43% of the power required at 5 V. The clock speed is reduced proportionally to the voltage at the lower operating voltage. Thus the clock speed at 3.3 V will be about  $2/3$  of the clock speed at 5 V. The operating current is reduced in proportion to operating voltage.

The Rabbit does not have a "standby" mode that some microprocessors have. Instead, the Rabbit has the ability to switch its clock to the 32.768 kHz oscillator. This is called the *sleepy* mode. When this is done, the power consumption is dramatically decreased. The current consumption is often reduced to the region of 100  $\mu\text{A}$  at this clock speed. The Rabbit executes about 6 instructions per millisecond at this low clock speed. Generally, when the speed is reduced to this extent, the Rabbit will be in a tight polling loop looking for an event that will wake it up. The clock speed is increased to wake up the Rabbit.



## 14.8 Basic Memory Design

Normally /CS0 and /OE0 and /WE0 should be connected to a flash memory that holds the startup code that executes at address zero. When the processor exits reset with (SMODE1, SMODE0) set to (0,0), it will attempt to start executing instructions at the start of the memory connected to /CS0, /OE0, and /WE0.

By convention, the basic RAM memory should be connected to /CS1, /OE1, and /WE1. /CS1 has a special property that makes it the preferred chip select for battery-backed RAM. A bit may be set in the MMIDR register to force /CS1 to stay enabled (low) (see Table 21 in Section 8.3.1). This capability can be used to counter a problem encountered when the chip select line is passed through a device that is used to place the chip in standby by raising /CS1 when the power is switched over to battery backup. The battery switchover device typically has a propagation delay that may be 20 ns or more. This is enough to require the insertion of wait states for RAM access in some cases. By forcing /CS1 low, the propagation delay is not a factor because the RAM will be always selected and will be controlled by /OE1 and /WE1. If this is done, the RAM will consume more power while not battery-backed than it would if it were run with dynamic chip select and a wait state. If this special feature is used to speed up access time for battery backed RAM then no other memory chips should be connected to OE1 and WE1.

### 14.8.1 Memory Access Time

The memory access time required depends on the clock speed and the capacitive loading of the address and data lines. Wait states can be specified by programming to accommodate slow memories for a given clock speed. Wait states should be avoided with memory that holds programs because there is a significant slowing of the execution speed. Wait states are far more important in the instruction memory than in the data memory since the great majority of accesses are instruction fetches. Going from 0 to 1 wait states is about the same as reducing the clock speed by 30%. Going from 0 to 2 wait states is worth approximately a 45% reduction in clock speed. A table of memory access times required for various clock speeds is given in Table 44 in Chapter 15.

### 14.8.2 Precautions for Unprogrammed Flash Memory

If a Rabbit-based system is powered up and released from reset when not in one of the cold-boot modes, the processor attempts to begin execution by reading from address zero of the memory attached to /CS0, /OE0, and /WE0. If this memory is an unprogrammed or improperly programmed flash memory, there is a danger that the memory could be destroyed if the write security feature of the flash memory is disabled. Flash memories have a write security feature that inhibits starting write cycles unless a special code is first stored to the memory. For example, Atmel flash memories use the bytes AAh, 55h, and A0h stored to addresses AAAAh or 5555h in a particular sequence. Any write executed that is not prefixed by this sequence will be ignored. If the memory has write protection disabled, and execution starts, it is possible that an endless loop that includes a write to memory will establish itself. Since the flash memory wears out after a few hundred thousand writes, the memory could be damaged in a short period of time by such a loop. Un-

fortunately, flash memory is shipped from the factory with the protection feature disabled to accomodate obsolete memory programmers.

The solution to this problem is to order the memory with the write protection enabled, or to enable it with a flash programming system. Then the memory will be safe if it is soldered into the Rabbit system. If an unsafe memory is soldered into a system, then the memory can be powered up with the programming cable connected, and a sequence can be sent using the cold-boot procedure to enable the write protection. Compiling any Dynamic C program to the flash will make the memory safe. If this is not convenient, tester software can make the memory safe by sending a byte sequence over the programming connection serial link.

The following example shows a program that can be downloaded via the cold-boot protocol to make a Atmel AT29C010A 128K x 8 flash memory safe. In this case, the RAM connected to /CS1 is used to hold a program starting at address zero. The flash memory is mapped into the data segment starting at address 1000h for access to the start of the flash memory.

```
; before storing this program the RAM is mapped to the
; first quadrant
; the program that resides at address zero in RAM
; note: this program has not been tested
ld a,0e1h ; 3e e1 segsize reg
ioi ld (13h),a ; d3 32 13 00 data seg starts at 1000h
ld a,3fh ; 3e 3f dataseg reg
ioi ld(12h),a ; d3 32 12 00 set data seg base of flash to 1000h
ld a,0 ; 3e 00 for MB1CR memory bank reg for flash on cs0
ld (15h),a ; 32 15 00 bank 1 reads flash starting at 256k
ld a,0aah ; 3e aa
ld (5555h+1000h),a ; 32 55 65 first byte of unlock code
ld a,55h ; 3e 55
ld (2AAAh+1000h),a ; 32 aa 3a 2nd byte of unlock code
ld a,0a0h ; 3e a0
ld (5555h+1000h),a ; 32 55 65 3rd byte of unlock code
ld hl,1000h ;21 00 10 point to start of flash memory
ld (hl),0c3h ; 36 c3 jump op code
inc hl ; 23
ld (hl),00h ; 36 00 zero
inc hl ; 23
ld (hl),00h ; 36 00 zero
jr * ; 18 fe end with endless loop
```

This code can be sent by means of a sequence of triplets via the serial port.

```
80 14 01 ; I/O write 01 to 0000 MB0CR select cs1- map RAM to Q1
00 00 3e ; write to memory address 0
00 01 e1
00 02 d3
00 03 32
00 04 12
00 05 00
; continue code above here
00 2b 18 ; last instruction
```

```
00 2c ef ; last byte
80 24 80 ; start execution of program at zero
```

The program will execute within about 10 ms.

## 14.9 PC Board Layout and Memory Line Permutation

In order to use the PC board real estate efficiently, it is recommended that the address and data lines to memory be permuted to minimize the use of PC board resources. By permuting the lines, the need to have lines cross over each other on the PC board is reduced, saving feed-through's and space.

For static RAM, address and data lines can be permuted freely, meaning that the address lines from the processor can be connected in any order to the address lines of the RAM, and the same applies for the data lines. For example, if the RAM has 15 address lines and 8 data lines, it makes no difference if A15 from the processor connects to A8 on the RAM and vice versa. Similarly D8 on the processor could connect to D3 on the RAM. The only restriction is that all 8 processor data lines must connect to the 8 RAM data lines. If several different types of RAM can be accommodated in the same PC board footprint, then the upper address lines that are unused if a smaller RAM is installed must be kept in order. For example, if the same footprint can accept either a 128K x 8 RAM with 17 address lines or a 512K x 8 RAM with 19 address lines, then address lines A18 and A19 can be interchanged with each other, but not exchanged with A0–A17.

Permuting lines does make a difference with flash memory. If the memory is socketed and it is intended to program the memory off the board, then it is probably best to keep the address and data lines in their natural order. However, since the flash can be programmed in the circuit using the Rabbit programming port, it is expected that most designers will solder the flash memory directly to the board in an unprogrammed state. In this case, the permutation of data and address lines must still be taken into account because flash memory requires the use of a special unlock code that removes write protection. The unlock operation involves a special sequence of reads and writes accessing special addresses and writing the unlock codes.

Another consideration is that the flash memory may be divided into sectors. In order to modify the memory, an entire sector must be written. In the small-sector memories the memory is divided into 1024 sectors. If the largest flash memory that is usable in a particular design is 512K, the largest sector size is 512 bytes. If the smallest memory used is 128K, then the smallest sector is 128 bytes. In order that the sector can be contiguous for all possible types of memory, the lower 7 address lines (A0...A6) should be permuted as a group. Address lines A7 and A8 should not be permuted at all if it is desirable to keep the larger sectors contiguous. The upper 10 address lines can be permuted as a separate group. The special memory chip addresses 05555h and 0AAAAh must be accessed as part of the unlock sequence. These addresses use only the first 16 address lines and have the odd and even numbered bits the same. The unlock codes use the numbers 55h, AAh or A0h.

Permuting data or address lines with flash memory should probably be avoided in practical systems.

## 15. AC Timing Specifications

The Rabbit processor may be operated at voltages between 2.5 V and 6.0 V, and at temperatures from  $-40^{\circ}\text{C}$  to  $+85^{\circ}\text{C}$ . Most users will operate the Rabbit at either 5.0 V or 3.3 V. The most computation per watt is obtained at approximately 3.3 V. The highest practical speed is usually obtained at 5 V (or more).

The Rabbit is available in two versions: the R-25, which has a maximum clock speed of 26.0 MHz, and the R-30, which has a maximum clock speed of 29.5 MHz. The R-30 has a maximum clock speed at  $3.3\text{ V} \pm 10\%$  of 18.9 MHz, and the R-25 has a maximum clock speed of 16.25 MHz at 3.3 V. The maximum clock speed at 2.5 V is 8.25 MHz for either the R-25 or R-30.

If a half-speed crystal is used with the clock doubler to achieve the desired clock speed, the maximum clock speed must be reduced by 10% to allow for an up to 10% asymmetry (55/45) in the waveform generated by the oscillator. This is because the clock doubler uses the intermediate edge to generate the double frequency.

A voltage between 3 V and 3.5 V should be used to minimize power consumption, and the clock speed should be adjusted downward as far as feasible. This will give the maximum computation per watt.

**Table 43. Rabbit Basic Worst-Case Timings (Preliminary 6/24/99)**

	<b>2.50 V min. -40°C—+85°C</b>	<b>3.3 V <math>\pm 10\%</math> -40°C—+85°C R-25/R-30</b>	<b>5.0 V <math>\pm 10\%</math> -40°C—+85°C R-25/R-30</b>
Maximum clock speed	8.25 MHz	16.25 MHz/ 18.9 MHz	26 MHz/ 29.5 MHz
Maximum clock speed generated using clock doubler	7.42 MHz	14.8 MHz/ 17.0 MHz	22.5 MHz/ 26.5 MHz
Tadr clock to address out delay with 20 pf address line load	18 ns	10 ns	6 ns
Minimum clock period	121 ns	61.5 ns/53 ns	39 ns/34 ns
Tadr clock to address out delay with 20 pf address bus load	24 ns	13 ns	8 ns
Tadr clock to address out delay with 70 pf address line load	39 ns	20 ns	13 ns
Tsetup data setup prior to clock	12 ns	7 ns	4 ns

The values in Table 43 may be improved by 6% for commercial ratings. The industrial rating is the specified voltage plus or minus 10% and a maximum temperature  $85^{\circ}\text{C}$ . The commercial rating is plus or minus 5% voltage and a maximum temperature of  $70^{\circ}\text{C}$ . The effect of temperature alone is approximately 1% worse speed for each  $5^{\circ}\text{C}$  temperature increase.

The memory access time required for a directly interfaced memory is given by:

$$\text{access time} = (\text{clock period}) * (2 + \text{waitstates}) - T_{\text{setup}} - T_{\text{addr}} \quad (1)$$

This formula remains true if the clock doubler is used, except that, if there are an odd number of wait states, the access time must be reduced by 10% of one clock period. See the diagrams below for the definition of  $T_{\text{setup}}$  and  $T_{\text{addr}}$ .

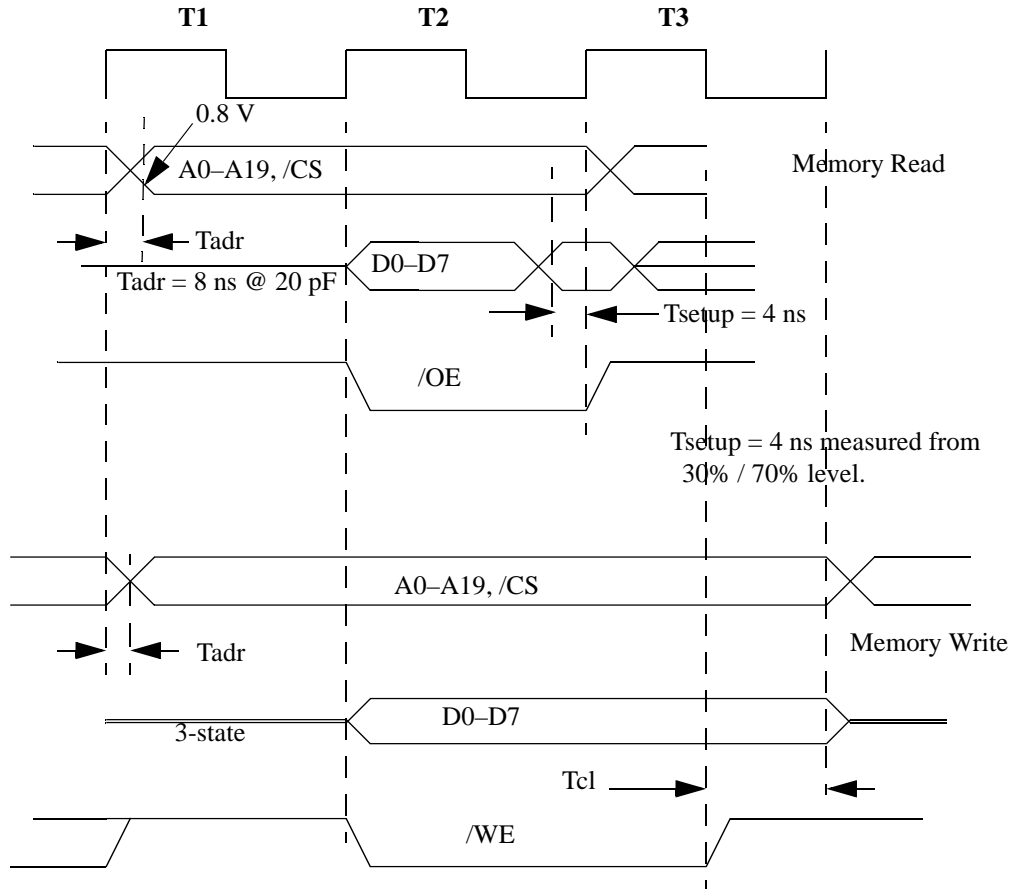
If serial communication is to be used at standard baud rates, then certain clock speeds must be used. These clock speeds are usually multiples of 1.8432 MHz to ensure that baud rates of 57,600 bps, 19,200 bps, and less will be available. Multiples of 3.6862 MHz ensure that baud rates of 115,200 bps, 38,400 bps, and less will be available. Multiples of 1.2288 MHz ensure that baud rates of 38,400 bps and less will be available. The standard Rabbit BIOS will accept any clock speed that is a multiple of .6144 MHz.

**Table 44. Memory Access Time Requirements ( $V \pm 10\%$ ,  $T -40^{\circ}\text{C}$  to  $+85^{\circ}\text{C}$ )**

Clock Speed (MHz)	Period (ns)	Wait States	Memory Access Time @ 5 V 20 pF Load (ns)	Memory Access Time @ 5 V 70 pF Load (ns)	Maximum PC-Compatible Baud Rate (bps)
29.4912	34	0	56	51	921,600
27.6480	36.2	0	60	55	57,600
25.8048	38.7	0	65	59	115,200
25.8048	38.7	1	104	99	115,200
25.8048	38.7	2	142	137	115,200
24.576	40.7	0	70	65	38,400
23.9616	41.7	0	71	66	57,600
22.1184	45.2	0	78	73	230,400
22.1184	45.2	1	124	119	230,400
22.1184	45.2	2	169	164	230,400
20.2752	49.3	0	88.6	82.6	57,600
18.432	54.2	0	96.5	91.5	115,200
14.7456	67.8	0	124 @ 5 V/ 119 @ 3.3 V	119 @ 5 V/ 109 @ 3.3 V	460,800
14.7456	67.8	1	192 @ 5 V/ 187 @ 3.3 V	187 @ 5 V/ 177 @ 3.3 V	460,800
11.0592	90.5	0	169 @ 5 V/ 164 @ 3.3 V	164 @ 5 V/ 154 @ 3.3 V	115,200
7.3728	135.6	0	259 @ 5 V/ 254 @ 3.3 V 235 @ 2.5 V	254 @ 5 V/ 244 @ 3.3 V/ 220 @ 2.5 V	230,400

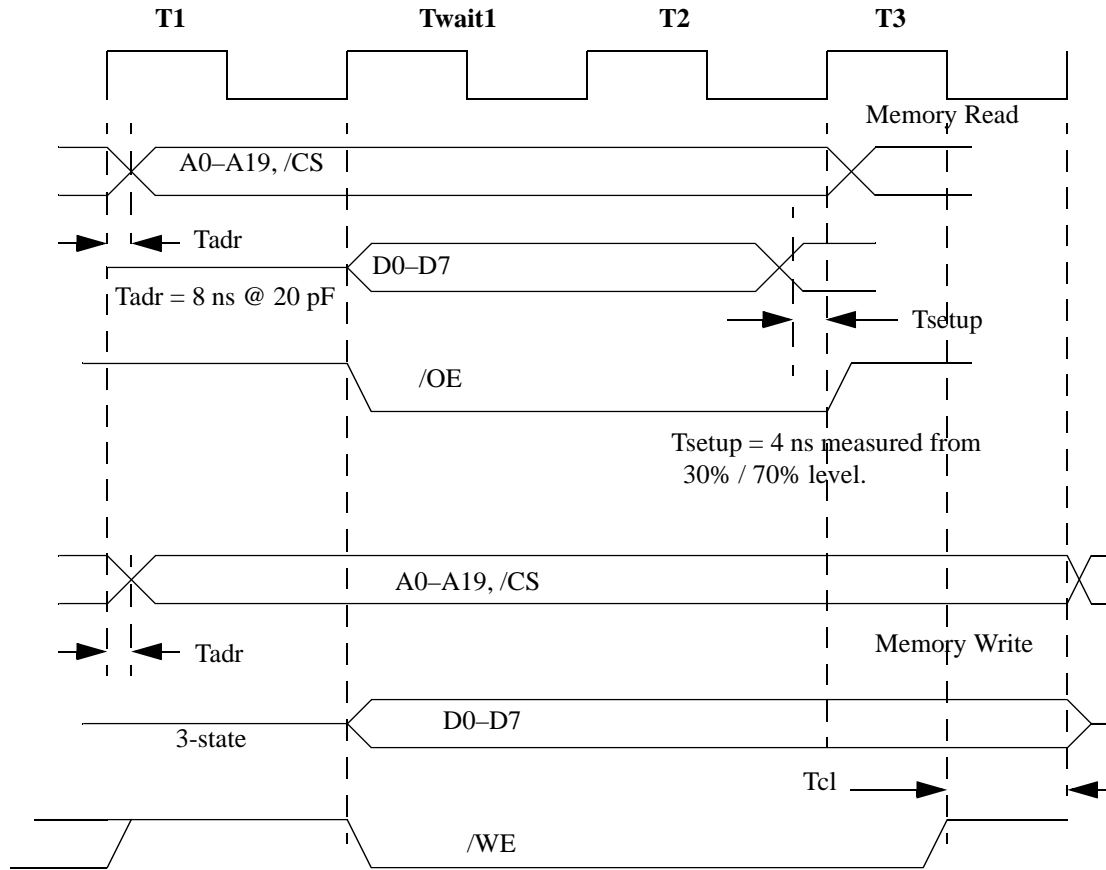
The die suffers significant self-heating at higher clock speeds. The thermal impedance, die to ambient, with zero air flow is 44°C/W. At 5 V with 65 mA current consumption this would result in about 15°C of self-heating, and would reduce the maximum clock speed by approximately 3%. This reduction is included in the tables above.

Figure 38, Figure 39, and Figure 40 illustrate the memory read and write cycles.



**Figure 38. Memory Read and Write Cycles**

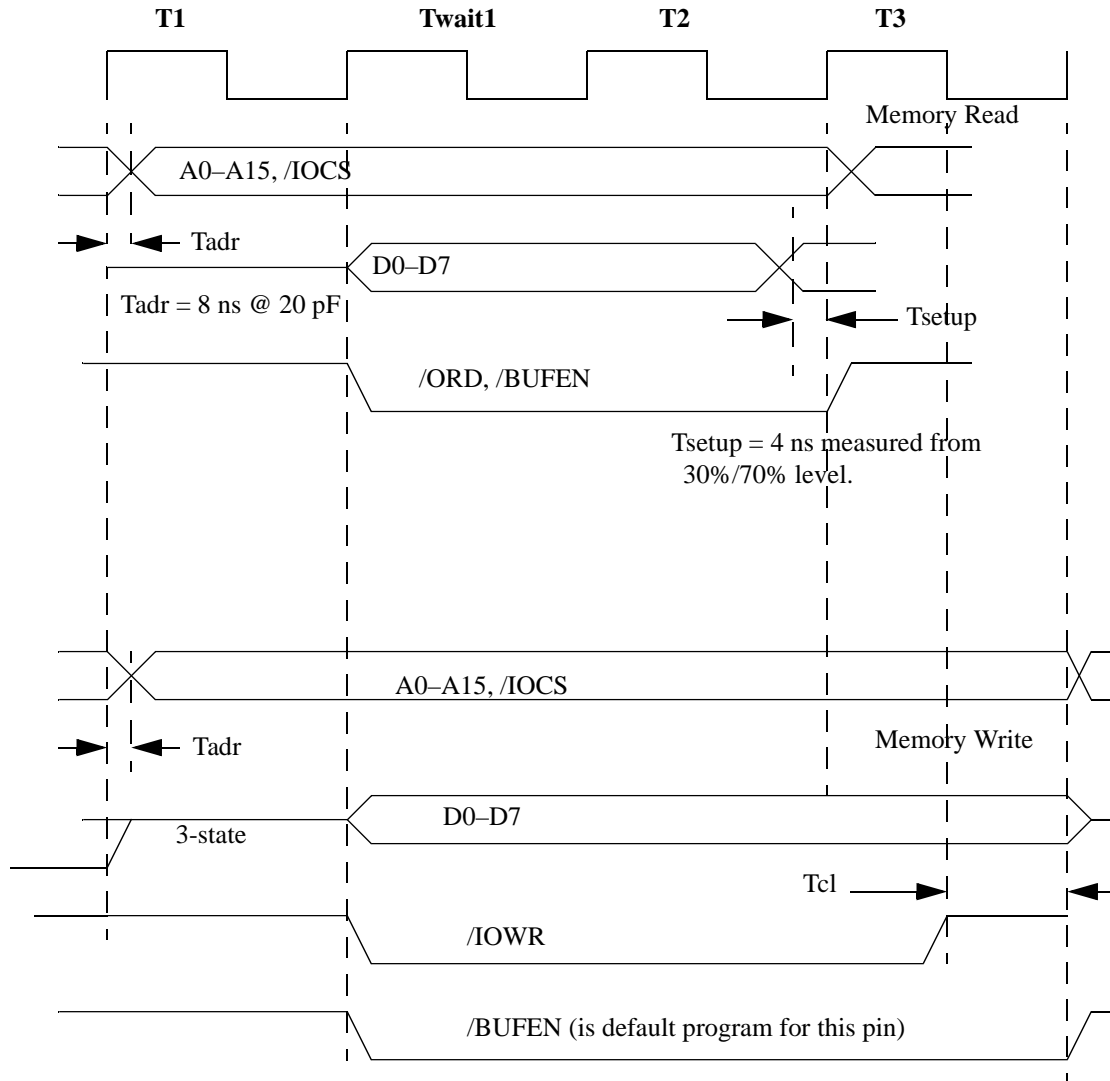
$T_{adr}$  is the time required for the address output to reach 0.8 V. This time depends on the bus loading. A0 has a stronger driver and can handle larger capacitive loads than the other address lines.  $T_{setup}$  is the data setup time relative to the clock.  $T_{setup}$  is specified from the 30%/70% of the VDD voltage level. Most 5 V memories have TTL compatible inputs that switch at 0.8 V and 2.0 V. Approximately 2 ns must be added for a 20 pF bus loading in order to pull a low level of 0.8 V instead of 2.5 V.



**Figure 39. Memory Read and Write with Wait States**

Generally, the maximum operating speed is proportional to the power supply voltage. The operating current is proportional to the voltage, and so the operating power is proportional to the square of the voltage. The operating power is also proportional to the clock speed. Higher temperatures reduce the maximum operating speed by approximately 1% for each 5°C. In addition, higher operating speeds increase the die temperature because of the heat generated and therefore slightly compound the adverse effects of higher temperature. The Rabbit operates at 2 clocks per bus cycle plus any wait states that might be specified. I/O bus cycles have an automatic wait state and thus require 3 clocks plus any wait states specified as shown in Figure 40.





**Figure 40. I/O Read and Write Cycles No Extra Wait States**

## 15.1 Current Consumption

Typical current is proportional to both clock frequency and voltage. The main oscillator requires approximately 6 mA at 5V and 2 mA at 3 volts independent of frequency. The basic current consumption for the processor exclusive of the oscillator at 5 V and 15 MHz is approximately 42 mA. The following formula can be used to compute the current consumption:

$$I = (0.7) * (\text{freq MHz}) * (\text{voltage}) + (1.27) * (\text{voltage} - 2.7) + 1.3$$

The first term represents the power consumed by the processor. The second term is the current consumed by the main oscillator. If the main oscillator is disabled then this term is zero. Some checkpoints for current consumption are below.

**Figure 41. Current Versus Frequency and Voltage**

Frequency MHz	Voltage	Current mA
29.4912	5	107
22.0592	5	82
14.7456	5	56
14.7456	3.3	36
7.3728	3.3	19
3.6864	3.3	11
1.8432	3.3	6
.9216	3.3	4

To these figures must be added current consumed by memory and other devices included in the system. The 32.768 clock oscillator consumes approximately 70 uA at 3 volts. (At 2.4 volts, when backed by a battery the consumption is approximately 25 uA.) Current consumed by RAM or flash memory will be substantial and very significant at lower frequencies if auto power down flash is not used. At this time accurate estimates for current consumption when the main clock is provided by the 32.768 kHz oscillator are not available. However, in principle at 3.3 volts the current should consist of 90 uA for the oscillator and 75 uA for the processor, or a total of 165 uA.

## 16. Rabbit Software

This chapter outlines basic low-level software constructs that are recommended for use with the Rabbit microprocessor. By following these suggestions, the user will find it much easier to use the software and to avoid various pitfalls. More details can be found by consulting the Dynamic C manual or the source code in the Dynamic C libraries.

### 16.1 Reading and Writing I/O Registers and Shadow Registers

The Rabbit has two I/O spaces: internal I/O registers and external I/O registers. Access is the same as for accessing data memory except that the instruction is preceded by a prefix (ioi or ioe) to indicate the internal or external I/O space.

The fastest way to read and write I/O registers in Dynamic C is to use a short segment of assembly language inserted in the C program. For example.

```
// compute value and write to port A data register
value=x+y
#asm
ld a,(value) ; value to write
ioi ld (PADR),a ; write value to PADR
#endasm
```

In the example above the ioi prefix changes a store to memory to a store to an internal I/O port. The prefix ioe is used for writes to external I/O ports.

A series of C callable functions are available to read and write I/O registers.

```
// Internal I/O Register Calls
int RdPortI(int PORT); // returns port, high byte zero
int BitRdPortI( int PORT, int bitcode) ; // bit code 0-7
// writes 8 bits to port and shadow
// no shadow write if a null pointer is used for shadow
void WrPortI( int PORT, char *PORTShadow, int value);
// write to a port bit (bits are numbered 7, 6, ... 1, 0.
void BitWrPortI(
    int PORT, char *PORTShadow, int value, int bitcode);

// Same external I/O registers
int RdPortE(int adr); // returns contents of port, high byte zero
int BitRdPortE( int PORT, int bitcode) ; // bit code 0-7
int WrPortE( int PORT, char *PORTShadow, int value);
int BitWrPortE(
    int PORT, char *PORTShadow, int value, int bitcode);
```

In order to read a port the following code could be used:

```
k=RdPortI(PDDR); // returns port D data register
```

If the port is a write only port then the shadow register can be used to find out what the contents of the port are. For example the global control status register has a number of

write-only bits. These can be read by consulting the shadow, provided that the shadow register is always updated when writing to the register.

```
k=GCSRShadow;
```

In order to write a write only register and update the shadow register the following routines can be used:

```
WrPortI(GCSR,&GCSRShadow,value); // update register and shadow
BitWrPortI(GCSR,&GCSRShadow,1,5); // set 1 to bit 5 of GCSR
```

In the WrPortI routine a zero can be substituted for the pointer to the shadow register if no shadow register is to be used. The pointer to the shadow register is mandatory for BitWrPortI.

## 16.2 Shadow Registers

Many of the registers of the Rabbit's internal I/O devices are write-only. Write-only registers save gates on the chip, making possible greater capability at lower cost. Typical designs that implement external I/O registers also have write-only registers due to the cost saving of not having the additional hardware to make it possible to read back the register's contents. Write-only registers are easier to use if a memory location, called a shadow register, is associated with each write-only register. In order to make the names of the shadows easy to remember they are compounded from the internal register names used in this manual. For example the register PADR (port A data register) has the shadow PADRShadow. Some shadow registers are defined in the bios files as shown below.

```
// the internal I/O registers -the shadows
// parallel ports
char PADRShadow,PBDRShadow, PCDRShadow, PCFRShadow;
char PDDRShadow, PDCRShadow, PDFRShadow, PDDCRShadow, PDDDRShadow;
char PEDRShadow, PECRShadow, PEFRShadow, PEDDRShadow;

char GCSRShadow; // global control status register
char GOCRShadow; // global control
char GCDRShadow; // clock doubler
```

When manipulating I/O registers and shadow registers, the programmer must keep in mind that an interrupt can take place in the middle of the sequence of operations, and then the interrupt routine may manipulate the same registers. If this possibility exists, then a solution must be crafted for the particular situation. Usually it is not necessary to disable the interrupts while manipulating registers and their associated shadow registers.

As an example, consider the parallel port D data direction register (PDDDR). This register is write only, and it contains 8 bits corresponding to the 8 I/O pins of parallel port D. If a bit in this register is a "1," the corresponding port pin is an output, otherwise it is an input. It is easy to imagine a situation where different parts of the application, such as an interrupt routine and a background routine, need to be in charge of different bits in the PDDDR register. The following code sets a bit in the shadow and then sets the I/O regis-

ter. If an interrupt takes place between the `set` and the `ldd`, and changes the shadow register and PDDDR, the correct value will still be set in PDDDR.

```
ld hl,PDDDRShadow    ; point to shadow register
ld de,PDDDR          ; set de to point to I/O reg
set 5,(hl)           ; set bit 5 of shadow register
; use ldd instruction for atomic transfer
ioi ldd    ; (io de)<-(hl)  side effect: hl--, de--
```

In this case, the `ldd` instruction when used with an I/O prefix provides a convenient data move from a memory location to an I/O location. Importantly, the `ldd` instruction is an atomic operation so there is no danger that an interrupt routine could change the shadow register during the move to the PDDDR register. If two instructions such as the following were used instead of the `ldd` instruction,

```
ld a,(hl)
ld (PDDDR),a ; output to PDDDR
```

then there is the possibility that an interrupt would take place after the first instruction, change the shadow register and the PDDDR register, and then after a return from the interrupt, the second instruction would execute and store an obsolete copy of the shadow register in the PDDDR, setting it to a wrong value.

There is no reason to have a shadow register for many of the registers that can be written to. In some cases, writing to registers is used as a handy way of changing a peripheral's state, and the data bits written are ignored. For example, a write to the status register in the Rabbit serial ports is used to clear the transmitter interrupt request, but the data bits are ignored, and the status register is actually a read-only register except for the special functionality attached to the act of writing the register. An illustration of a write-only register for which a shadow is unnecessary is the transmitter data register in the Rabbit serial port. The transmitter data register is a write-only register, but there is little reason to have a shadow register since any data bits stored are transmitted promptly on the serial port.

### 16.3 Timer and Clock Usage

The real time clock or battery backable clock is a 48 bit counter that counts at 32768 counts per second. The counting frequency comes from the 32.768 kHz oscillator which is separate from the main oscillator. Two other important devices are also powered from the 32.768 kHz oscillator: the periodic interrupt and the watchdog timer. It is assumed that all measurements of time will derive from this clock and not the main processor clock which operates at a much higher frequency (e.g. 22.1184 MHz). This allows the main processor oscillator to use less expensive ceramic resonators rather than quartz crystals. Ceramic resonators typically have an error of 5 parts in 1000, while crystals are much more accurate, to a few seconds per day.

It is not intended that the real time clock be read and written frequently. The procedure to read it is lengthy and has an uncertain execution time. The procedure for writing the clock is even more complicated. Rather, Dynamic C software maintains a long variable

**SEC\_TIMER** in memory. **SEC\_TIMER** is updated every second by the periodic interrupt and may be read or written directly by the user's programs. Since **SEC\_TIMER** is driven by the same oscillator as the real time clock there is no relative gain or loss of time between the two. As part of the standard startup code **SEC\_TIMER** has bits 15-46 of the real time clock copied to it. **SEC\_TIMER** holds the number of seconds since 12 AM of 1-January-1980. **SEC\_TIMER** can accomodate 136 years from 1980 or to the year 2116. Another long timer **MS\_TIMER** counts milliseconds and can be used in a similar manner. **MS\_TIMER** wraps around from max count to zero approximately every 6 weeks. The software that uses the counters measures intervals correctly even if the counter used wraps around.

```
unsigned long int read_rtc(void); // read bits 15-46 rtc
void write_rtc(unsigned long int time); // write bits 15-46
// note: bits 0-14 and bit 47 are zeroed
```

Two utility routines are provided that can be used to convert times between the traditional format (10-Jan-2000 17:34:12) and the seconds since 1-Jan-1980 format.

```
// converts time structure to seconds
unsigned long mktime(struct tm *timeptr);

// seconds to structure
unsigned int mktm(struct tm *timeptr, unsigned long time);
```

The format of the structure used is the following

```
struct tm {
char tm_sec;           // seconds 0-59
char tm_min;          // 0-59
char tm_hour;         // 0-59
char tm_mday;         // 1-31
char tm_mon;          // 1-12
char tm_year;         // 00-150 (1900-2050)
char tm_wday;         // 0-6 0==sunday
};
```

The day of the week is not used to compute the long seconds, but it is generated when computing from long seconds to the structure. A utility routine setclock.c is available to set the date and time in the real time clock from the Dynamic C console.

## 16.4 WatchDog Support Software

A microprocessor system can crash for a variety of reasons. A software bug or an electrical upset are common reasons. When the system crashes the program will typically settle into an endless loop because parameters that govern looping behavior have been corrupted. Typically the stack becomes corrupted and returns are made to random addresses.

The usual corrective action taken in response to a crash is to reset the microprocessor and reboot the system. The crash can be detected either because an anomaly is detected by program consistency checking or because a part of the program that should be executing periodically is not executing and the watchdog times out.

Direct detection of crashes is supported in Dynamic C by certain checksumming operations and other causes of error that are classed as fatal errors.

The virtual watchdog system allows establishing multiple virtual watchdogs for different parts of the program which must be executed periodically. If any of the virtual watchdogs times out, then hits are withheld from the hardware watchdog and it times out, resulting in a hardware reset. The virtual watchdogs are implemented by an array of memory counters. The counters are counted down 16 times per second by the periodic interrupt routine. Hitting a virtual watchdog is performed by calling a routine that stores a count between 1 and 255 in the memory counter. Virtual watchdogs may be allocated, deallocated, enabled and disabled. One virtual watchdog is implemented by default and it is hit in the periodic interrupt routine. If the periodic interrupt stops working, then the watchdog will time out. The advantage of the virtual watchdogs is that if any of them fail and error is detected. Directly hitting the hardware watchdog will cause an error only if every place where the watchdog is hit fails to be included in the crash loop.

### 16.4.1 The Watchdog Hardware

The Rabbit microprocessor has a hardware watchdog timer. The watchdog is hit by calling a bios routine `hitwd()` which hits it for 2 seconds or by storing a special code in the `WDTCR` register, the code determining the time delay. The watchdog is a 17 bit counter that counts toward zero at 32768 Hz provided by the 32.768 kHz oscillator. A hit stores a number in the counter, postponing the time when it will reach zero. If hits are not frequent enough the counter will reach zero and perform a microprocessor reset.

Best practice requires that extreme care be taken before the program hits the watchdog. If hits of the watchdog are scattered in a reckless manner throughout the user's program then there is a good chance that the watchdog will become effectively useless, because when a crash takes place the program will enter an endless loop that includes a hit of the watchdog.

### 16.4.2 The Virtual Watchdog System

By default 10 virtual watchdogs are available. This can be changed by a `#define`:

```
#define N_WATCHDOG 15 // default is 10 watchdogs
```

To allocate a watchdog make the call:

```
N= VdGetFreeWd(char count); // establish watdog with
                             // timeout count/16 seconds
```

To hit that watchdog use the call:

```
VdHitWd(int N); // hit that watchdog
```

To remove that watchdog from the table use the call:

```
VdReleaseWd(int N); // release (deestablish) watchdog N
```





## 17. Rabbit Standard BIOS

(Note: The information in this chapter is provided for conceptual purposes only. An update in the form of a software manual will clarify these issues.)

The Rabbit standard BIOS is a package of software that handles startup, shutdown and various basic features of the Rabbit. By providing standard software to perform basic functions, the user is relieved of the necessity of re-inventing this software for his own needs. Further, by using Z-World's tested software, the user greatly reduces the possibility of errors and bugs. Z-World provides the full source code for the BIOS so the user has the possibility of modifying it and so that the user has a ready reference to examine details of the operation of the BIOS that are not apparent from the documentation.

In general, the BIOS is customized for each different controller board. The BIOS has declarations at the start of the code that define the hardware configuration and use options.

A general-purpose BIOS is available that will work with most systems based on the Rabbit. The general-purpose BIOS is useful for bringing up new designs.

### 17.1 The BIOS—More Details

The BIOS is compiled separately from the user's application. It occupies space at the bottom of the root code segment. When execution of the user's program starts at address zero, it starts in the BIOS. There is no limit to the amount of code that can be included in the BIOS. If the user compiles libraries as a part of the BIOS, time can be saved since the BIOS is not recompiled except for specific reasons.

Normally routines that are frequently called on or that are needed for essential functions are in the BIOS. When Dynamic C cold-boots the target and downloads the binary image, the symbol table is retained to make it possible for the user program to call entry points in the BIOS.

The BIOS supports the following services.

- System startup, including setup of memory, wait states and clock speed.
- Reading and programming the real-time clock.
- Operation and management of the periodic interrupt.
- Maintenance of memory counters that count ticks, milliseconds and seconds since January 1, 1980.
- Operation of the watchdog timer and maintenance of a system of virtual watchdog timers.
- Routines to speed up and slow down the system clock for power management. The execution speed can be controlled over a wide range.
- Routines to set up the mode of operation of the parallel ports and to input and output data to them.

- Routines to initialize the timers and manipulate them.
- Routines to write flash memory with built-in protection of the system identification area.
- Routines to maintain an error log or operation log and to handle fatal errors and watchdog timeouts
- Basic services for multitasking
- Routines that support a general-purpose parameter setup via the programming port. This system can be used in the field for such items as setting a network address or calibration constants and setting the real-time clock.
- A download manager (to be available with future releases of the BIOS).
- Modbus slave.

## 17.2 BIOS Assumptions

The BIOS makes certain assumptions concerning the physical configuration of the processor. Processors are expected to have RAM connected to /CS1, /WE1, and /OE1. Flash memory, if present, is expected to be connected to /CS0, /WE0, and /OE0. The crystal frequency is expected to be  $n \cdot 6144 \cdot 3$  MHz, or else  $4 \cdot 6144$  MHz.

## 17.3 Periodic Interrupt and Real-Time Clock BIOS Services

The real-time clock is driven by the 32.768 kHz oscillator, which may be battery backed. The periodic interrupt, when enabled, occurs every 16 clocks or every 488  $\mu$ s. If the 32.768 kHz oscillator is absent, it is possible to substitute a different periodic interrupt, but this alternative is not supported by Z-World since it the cost of connecting a crystal is very small.

The periodic interrupt is used to count several memory counters that are used for general software use. These counters count ticks, milliseconds and seconds. These counters have an exact relationship with the real-time clock and the 32.768 kHz oscillator. A seconds counter is generated by adding  $65536/2048=32$  to a 16-bit word on every tick. The carry out counts the seconds counter. A millisecond counter is obtained by adding 32000 to a 16-bit word on every tick. The carry out occurs on an average of once per millisecond and drives the millisecond counter. The tick counter is counted 2048 times per second.

In addition, the periodic interrupt provides support for function slicing and a real-time kernel by providing periodic calls to clock routines. The periodic interrupt keeps the interrupts turned off (that is, the processor priority is raised to 1 from zero) for a minimum time of about 35 clocks. In this way, the periodic interrupt makes little contribution to interrupt latency.

### **17.3.1 Real-Time Clock Support**

If the real-time clock is battery-backed, which is an option in the BIOS, the BIOS reads the real-time clock on startup and sets up the seconds since 1980 counter synchronized with the clock. If the clock is not battery-backed, the seconds since 1980 counter is set to zero, thus measuring time since startup. A log of startup times and conditions is kept as a debugging aid. The time of exiting reset is recorded as well as the reason for the reset is kept in what may or may not be battery-backed memory.

### **17.3.2 Watchdog Timer Support**

In a well-organized system, the periodic interrupt should be the only place where instructions to hit the watchdog timer are located. This is accomplished by setting up a number of virtual watchdog timers. Each virtual watchdog is an 8-bit counter that is counted down 16 times per second, or every 128 ticks of the real-time clock. If any counter reaches zero, the program turns off interrupts and freezes until the hardware watchdog times out and resets the processor. The user program must hit each virtual watchdog periodically by calling a routine with the number of the watchdog and count value to be stored. The number of virtual watchdogs is itself a parameter that can be increased by a call to get the virtual watchdog routine, which adds another watchdog to the list (not a linked list) up to the maximum number in the array. Initially there is one virtual watchdog that is hit by the periodic interrupt itself. The virtual watchdogs can be distributed across interrupts to reduce the interrupt execution time if all possible virtual watchdogs must be counted in one interrupt. Precautions are taken to make sure that a crash will not result in accidentally hitting the watchdog.

### **17.3.3 Power Management Support**

The power consumption and speed of operation can be throttled up and down with rough synchronism. This is done by changing the clock speed, clock doubler, and memory wait states. The range of control is quite wide, 16-1 or more. In addition, the main clock can be switched to the 32.768 kHz clock. In this case, the slowdown is very dramatic, perhaps 500-1. Each clock takes about 30  $\mu$ s, and a typical instruction takes 150  $\mu$ s to execute. At this slow speed, the periodic interrupt cannot still operate because the interrupt routine would execute too slowly to keep up with an interrupt every 16 clocks. Only about 3 instructions could be executed between ticks.

A different set of rules applies in the ultra slow mode. The user will set up an endless loop to determine when to exit the ultra slow mode. The user should include a call in this loop to a polling routine that is a part of the BIOS. The polling routine will update the memory counters and the watchdog each time it is called. It will do this by directly reading the real-time clock and by catching up the memory counters. If the user's routine cannot get around the loop in the maximum watchdog timer timeout time, the user should put several calls to the polling routine in the loop. The user should avoid indiscriminate direct access to the watchdog timer and real-time clock. The least bits of the real-time clock cannot be read in ultra slow mode because they count fast compared to the instruction execution time.

### **17.3.4 Flash Memory Write Support**

A flash memory write routine is provided. This routine works differently or there are different routines for the cases when the memory to be written is in the primary code memory or is in a separate special memory. Writes to the primary code memory require freezing the system for 10 ms or so. Other writes just require that the user wait for the write to be done before doing the next write.

To protect the system identification block, the program tests the absolute memory address relative to the start of the flash memory connected to /CS0, /WE0, and /OE0, which is used to store the system identification block. The program has to check the contents of the memory bank control registers to make this check. A routine that can actually write this block is not included in the BIOS to make it hard to accidentally write this block.

## 18. Rabbit Instructions

Summary:

“Load Immediate Data” on page 149  
“8-bit Indexed Load and Store” on page 150  
“16-bit Indexed Loads and Stores” on page 150  
“16-bit Load and Store 20-bit Address” on page 150  
“Register to Register Moves” on page 151  
“Exchange Instructions” on page 151  
“Stack Manipulation Instructions” on page 152  
“16-bit Arithmetic and Logical Operations” on page 152  
“8-bit Arithmetic and Logical Operations” on page 153  
“8-bit Bit Set, Reset and Test Instructions” on page 154  
“8-bit Increment and Decrement” on page 154  
“8-bit Fast A register Operations” on page 154  
“8-bit Shifts and Rotates” on page 155  
“Instruction Prefixes” on page 156  
“Block Move Instructions” on page 156  
“Control Instructions - Jumps and Calls” on page 156  
“Miscellaneous Instructions” on page 157  
“Privileged Instructions” on page 157

Key

n-8-bit data item

d-8 bit offset

f - alternate destination prefix permitted, and the flags stored in F'

r - in "A" column - alternate destination prefix stores to alternate

s - in "I" column - I/O prefixes are allowed (IOE, IOI) I/O is source

d - in "I" column - I/O prefixes are allowed (IOE, IOI) I/O is dest

V - overflow flag set on overflow

L - overflow flag set if upper 4 bits of data word are zero

- - flag not affected

\* - flag is set by operation

Flag labels: S- sign, Z- zero, V- overflow, C- carry

### 18.1 Load Immediate Data

Instruction	clk	A	I	S	Z	V	C	Operation
LD IX,mn	8			-	-	-	-	IX = mn
LD IY,mn	8			-	-	-	-	IY = mn
LD dd,mn	6	r		-	-	-	-	dd = mn
LD r,n	4	r		-	-	-	-	r = n

## 18.2 Load and Store to an Immediate Address

Instruction	clk	A	I	S	Z	V	C	Operation
LD (mn),A	10		d	-	-	-	-	(mn) = A
LD A,(mn)	9	r	s	-	-	-	-	A = (mn)
LD (mn),HL	13		d	-	-	-	-	(mn) = L; (mn+1) = H
LD (mn),IX	15		d	-	-	-	-	(mn) = IXL; (mn+1) = IXH
LD (mn),IY	15		d	-	-	-	-	(mn) = IYL; (mn+1) = IYH
LD (mn),ss	15		d	-	-	-	-	(mn) = ssl; (mn+1) = ssh
LD HL,(mn)	11	r	s	-	-	-	-	L = (mn); H = (mn+1)
LD IX,(mn)	13		s	-	-	-	-	IXL = (mn); IXH = (mn+1)
LD IY,(mn)	13		s	-	-	-	-	IYL = (mn); IYH = (mn+1)
LD dd,(mn)	13	r	s	-	-	-	-	ddl = (mn); ddh = (mn+1)

## 18.3 8-bit Indexed Load and Store

Instruction	clk	A	I	S	Z	V	C	Operation
LD A,(BC)	6	r	s	-	-	-	-	A = (BC)
LD A,(DE)	6	r	s	-	-	-	-	A = (DE)
LD (BC),A	7		d	-	-	-	-	(BC) = A
LD (DE),A	7		d	-	-	-	-	(DE) = A
LD (HL),n	7		d	-	-	-	-	(HL) = n
LD (HL),r	6		d	-	-	-	-	(HL) = r = B, C, D, E, H, L, A
LD r,(HL)	5	r	s	-	-	-	-	r = (HL)
LD (IX+d),n	11		d	-	-	-	-	(IX+d) = n
LD (IX+d),r	10		d	-	-	-	-	(IX+d) = r
LD r,(IX+d)	9	r	s	-	-	-	-	r = (IX+d)
LD (IY+d),n	11		d	-	-	-	-	(IY+d) = n
LD (IY+d),r	10		d	-	-	-	-	(IY+d) = r
LD r,(IY+d)	9	r	s	-	-	-	-	r = (IY+d)

## 18.4 16-bit Indexed Loads and Stores

Instruction	clk	A	I	S	Z	V	C	Operation
LD (HL+d),HL	13		d	-	-	-	-	(HL+d) = L; (HL+d+1) = H
LD HL,(HL+d)	11	r	s	-	-	-	-	L = (HL+d); H = (HL+d+1)
LD (SP+n),HL	11		-	-	-	-	-	(SP+n) = L; (SP+n+1) = H
LD (SP+n),IX	13		-	-	-	-	-	(SP+n) = IXL; (SP+n+1) = IXH
LD (SP+n),IY	13		-	-	-	-	-	(SP+n) = IYL; (SP+n+1) = IYH
LD HL,(SP+n)	9	r	-	-	-	-	-	L = (SP+n); H = (SP+n+1)
LD IX,(SP+n)	11		-	-	-	-	-	IXL = (SP+n); IXH = (SP+n+1)
LD IY,(SP+n)	11		-	-	-	-	-	IYL = (SP+n); IYH = (SP+n+1)
LD (IX+d),HL	11		d	-	-	-	-	(IX+d) = L; (IX+d+1) = H
LD HL,(IX+d)	9	r	s	-	-	-	-	L = (IX+d); H = (IX+d+1)
LD (IY+d),HL	13		d	-	-	-	-	(IY+d) = L; (IY+d+1) = H
LD HL,(IY+d)	11	r	s	-	-	-	-	L = (IY+d); H = (IY+d+1)

## 18.5 16-bit Load and Store 20-bit Address

Instruction	clk	A	I	S	Z	V	C	Operation
LDP (HL),HL	12		-	-	-	-	-	(HL) = L; (HL+1) = H. (Adr[19:16] = A[3:0])

LDP (IX),HL	12	- - - -	(IX) = L; (IX+1) = H. (Adr[19:16] = A[3:0])
LDP (IY),HL	12	- - - -	(IY) = L; (IY+1) = H. (Adr[19:16] = A[3:0])
LDP HL,(HL)	10	- - - -	L = (HL); H = (HL+1). (Adr[19:16] = A[3:0])
LDP HL,(IX)	10	- - - -	L = (IX); H = (IX+1). (Adr[19:16] = A[3:0])
LDP HL,(IY)	10	- - - -	L = (IY); H = (IY+1). (Adr[19:16] = A[3:0])
LDP (mn),HL	15	- - - -	(mn) = L; (mn+1) = H. (Adr[19:16] = A[3:0])
LDP (mn),IX	15	- - - -	(mn) = IXL; (mn+1) = IXH. (Adr[19:16] = A[3:0])
LDP (mn),IY	15	- - - -	(mn) = IYL; (mn+1) = IYH. (Adr[19:16] = A[3:0])
LDP HL,(mn)	13	- - - -	L = (mn); H = (mn+1). (Adr[19:16] = A[3:0])
LDP IX,(mn)	13	- - - -	IXL = (mn); IXH = (mn+1). (Adr[19:16] = A[3:0])
LDP IY,(mn)	13	- - - -	IYL = (mn); IYH = (mn+1). (Adr[19:16] = A[3:0])

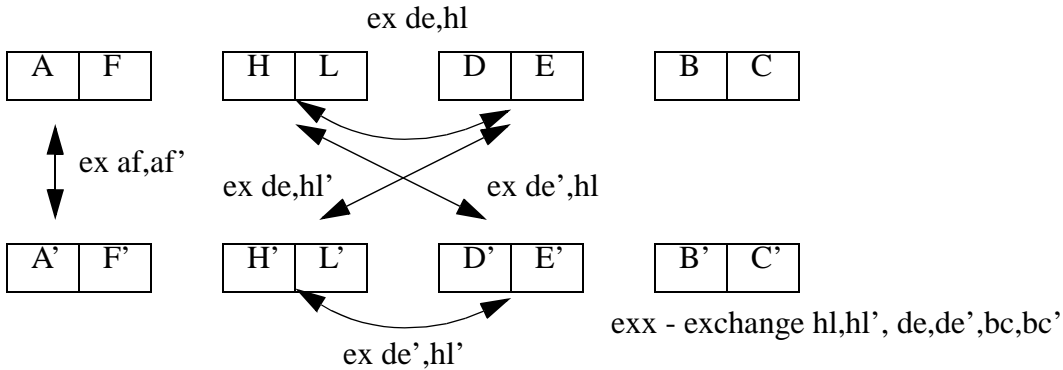
## 18.6 Register to Register Moves

Instruction	clk	A	I	S	Z	V	C	Operation
LD r,g	2	r	-	-	-	-	-	r = g- r,g any of B, C, D, E, H, L, A
LD A,EIR	4	fr	*	*	-	-	-	A = EIR
LD A,IIR	4	fr	*	*	-	-	-	A = IIR
LD A,XPC	4	r	-	-	-	-	-	A = MMU
LD EIR,A	4		-	-	-	-	-	EIR = A
LD IIR,A	4		-	-	-	-	-	IIR = A
LD XPC,A	4		-	-	-	-	-	XPC = A
LD HL,IX	4	r	-	-	-	-	-	HL = IX
LD HL,IY	4	r	-	-	-	-	-	HL = IY
LD IX,HL	4		-	-	-	-	-	IX = HL
LD IY,HL	4		-	-	-	-	-	IY = HL
LD SP,HL	2		-	-	-	-	-	SP = HL
LD SP,IX	4		-	-	-	-	-	SP = IX
LD SP,IY	4		-	-	-	-	-	SP = IY
LD dd',BC	4		-	-	-	-	-	dd' = BC (dd': 00-BC', 01-DE', 10-HL')
LD dd',DE	4		-	-	-	-	-	dd' = DE (dd': 00-BC', 01-DE', 10-HL')

## 18.7 Exchange Instructions

Instruction	clk	A	I	S	Z	V	C	Operation
EX (SP),HL	15	r	-	-	-	-	-	H <-> (SP+1); L <-> (SP)
EX (SP),IX	15		-	-	-	-	-	IXH <-> (SP+1); IXL <-> (SP)
EX (SP),IY	15		-	-	-	-	-	IYH <-> (SP+1); IYL <-> (SP)

EX AF,AF'	2		- - - -	AF <-> AF'
EX DE',HL	2	s	- - - -	if (!ALTD) then DE' <-> HL else DE' <-> HL'
EX DE',HL'	4	s	- - - -	DE' <-> HL'
EX DE,HL	2	s	- - - -	if (!ALTD) then DE <-> HL else DE <-> HL'
EX DE,HL'	4	s	- - - -	DE <-> HL'
EXX	2		- - - -	BC <-> BC'; DE <-> DE'; HL <-> HL'



## 18.8 Stack Manipulation Instructions

Instruction	clk	A	I	S	Z	V	C	Operation
ADD SP,d	4	f		-	-	-	*	SP = SP + d -- d=0 to 255
POP IP	7			-	-	-	-	IP = (SP); SP = SP+1
POP IX	9			-	-	-	-	IXL = (SP); IXH = (SP+1); SP = SP+2
POP IY	9			-	-	-	-	IYL = (SP); IYH = (SP+1); SP = SP+2
POP zz	7	r		-	-	-	-	zzl = (SP); zzh = (SP+1); SP=SP+2 -- zz= BC,DE,HL,AF
PUSH IP	9			-	-	-	-	(SP-1) = IP; SP = SP-1
PUSH IX	12			-	-	-	-	(SP-1) = IXH; (SP-2) = IXL; SP = SP-2
PUSH IY	12			-	-	-	-	(SP-1) = IYH; (SP-2) = IYL; SP = SP-2
PUSH zz	10			-	-	-	-	(SP-1) = zzh; (SP-2) = zzl; SP=SP-2 --zz= BC,DE,HL,AF

## 18.9 16-bit Arithmetic and Logical Operations

Instruction	clk	A	I	S	Z	V	C	Operation
ADC HL,ss	4	fr		*	*	V	*	HL = HL + ss + CF -- ss=BC, DE, HL, SP
ADD HL,ss	2	fr		-	-	-	*	HL = HL + ss
ADD IX,xx	4	f		-	-	-	*	IX = IX + xx -- xx=BC, DE, IY, SP
ADD IY,yy	4	f		-	-	-	*	IY = IY + yy -- yy=BC, DE, IX, SP
ADD SP,d	4	f		-	-	-	*	SP = SP + d -- d=0 to 255



AND HL,DE	2	fr	**L0	HL = HL & DE
AND IX,DE	4	f	**L0	IX = IX & DE
AND IY,DE	4	f	**L0	IY = IY & DE
BOOL HL	2	fr	**00	if (HL != 0) HL = 1, set flags to match HL
BOOL IX	4	f	**00	if (IX != 0) IX = 1
BOOL IY	4	f	**00	if (IY != 0) IY = 1
DEC IX	4		---	IX = IX - 1
DEC IY	4		---	IY = IY - 1
DEC ss	2	r	---	ss = ss - 1 -- ss= BC, DE, HL, SP
INC IX	4		---	IX = IX + 1
INC IY	4		---	IY = IY + 1
INC ss	2	r	---	ss = ss + 1 -- ss= BC, DE, HL, SP
MUL	12		---	HL:BC = BC * DE, signed 32 bit result. DE unchanged
OR HL,DE	2	fr	**L0	HL = HL   DE -- bitwise or
OR IX,DE	4	f	**L0	IX = IX   DE
OR IY,DE	4	f	**L0	IY = IY   DE
RL DE	2	fr	**L*	{CY,DE} = {DE,CY} -- left shift with CF
RR DE	2	fr	**L*	{DE,CY} = {CY,DE}
RR HL	2	fr	**L*	{HL,CY} = {CY,HL}
RR IX	4	f	**L*	{IX,CY} = {CY,IX}
RR IY	4	f	**L*	{IY,CY} = {CY,IY}
SBC HL,ss	4	fr	**V*	HL=HL-ss-CY (cout if (ss-CY)>hl)

## 18.10 8-bit Arithmetic and Logical Operations

Instruction	clk	A	I	S	Z	V	C	Operation
ADC A,(HL)	5	fr	s	**	V	*		A = A + (HL) + CF
ADC A,(IX+d)	9	fr	s	**	V	*		A = A + (IX+d) + CF
ADC A,(IY+d)	9	fr	s	**	V	*		A = A + (IY+d) + CF
ADC A,n	4	fr	**	V	*			A = A + n + CF
ADC A,r	2	fr	**	V	*			A = A + r + CF
ADD A,(HL)	5	fr	s	**	V	*		A = A + (HL)
ADD A,(IX+d)	9	fr	s	**	V	*		A = A + (IX+d)
ADD A,(IY+d)	9	fr	s	**	V	*		A = A + (IY+d)
ADD A,n	4	fr	**	V	*			A = A + n
ADD A,r	2	fr	**	V	*			A = A + r
AND (HL)	5	fr	s	**	L	0		A = A & (HL)
AND (IX+d)	9	fr	s	**	L	0		A = A & (IX+d)
AND (IY+d)	9	fr	s	**	L	0		A = A & (IY+d)
AND n	4	fr	**	L	0			A = A & n
AND r	2	fr	**	L	0			A = A & r
CP (HL)	5	f	s	**	V	*		A - (HL)
CP (IX+d)	9	f	s	**	V	*		A - (IX+d)
CP (IY+d)	9	f	s	**	V	*		A - (IY+d)
CP n	4	f	**	V	*			A - n
CP r	2	f	**	V	*			A - r
OR (HL)	5	fr	s	**	L	0		A = A   (HL)
OR (IX+d)	9	fr	s	**	L	0		A = A   (IX+d)

OR (IY+d)	9	fr s * * L 0	A = A   (IY+d)
OR n	4	fr * * L 0	A = A   n
OR r	2	fr * * L 0	A = A   r
SBC (IX+d)	9	fr s * * V *	A = A - (IX+d) - CY
SBC (IY+d)	9	fr s * * V *	A = A - (IY+d) - CY
SBC A,(HL)	5	fr s * * V *	A = A - (HL) - CY
SBC A,n	4	fr * * V *	A = A-n-CY (cout if (r-CY)>A)
SBC A,r	2	fr * * V *	A = A-r-CY (cout if (r-CY)>A)
SUB (HL)	5	fr s * * V *	A = A - (HL)
SUB (IX+d)	9	fr s * * V *	A = A - (IX+d)
SUB (IY+d)	9	fr s * * V *	A = A - (IY+d)
SUB n	4	fr * * V *	A = A - n
SUB r	2	fr * * V *	A = A - r
XOR (HL)	5	fr s * * L 0	A = [A & ~(HL)]   [~A & (HL)]
XOR (IX+d)	9	fr s * * L 0	A = [A & ~(IX+d)]   [~A & (IX+d)]
XOR (IY+d)	9	fr s * * L 0	A = [A & ~(IY+d)]   [~A & (IY+d)]
XOR n	4	fr * * L 0	A = [A & ~n]   [~A & n]
XOR r	2	fr * * L 0	A = [A & ~r]   [~A & r]

## 18.11 8-bit Bit Set, Reset and Test Instructions

Instruction	clk	A	I	S	Z	V	C	Operation
BIT b,(HL)	7	f	s	-	*	-	-	(HL) & bit
BIT b,(IX+d)	10	f	s	-	*	-	-	(IX+d) & bit
BIT b,(IY+d)	10	f	s	-	*	-	-	(IY+d) & bit
BIT b,r	4	f	-	*	-	-	-	r & bit
RES b,(HL)	10		d	-	-	-	-	(HL) = (HL) & ~bit
RES b,(IX+d)	13		d	-	-	-	-	(IX+d) = (IX+d) & ~bit
RES b,(IY+d)	13		d	-	-	-	-	(IY+d) = (IY+d) & ~bit
RES b,r	4	r	-	-	-	-	-	r = r & ~bit
SET b,(HL)	10		b	-	-	-	-	(HL) = (HL)   bit
SET b,(IX+d)	13		b	-	-	-	-	(IX+d) = (IX+d)   bit
SET b,(IY+d)	13		b	-	-	-	-	(IY+d) = (IY+d)   bit
SET b,r	4	r	-	-	-	-	-	r = r   bit

## 18.12 8-bit Increment and Decrement

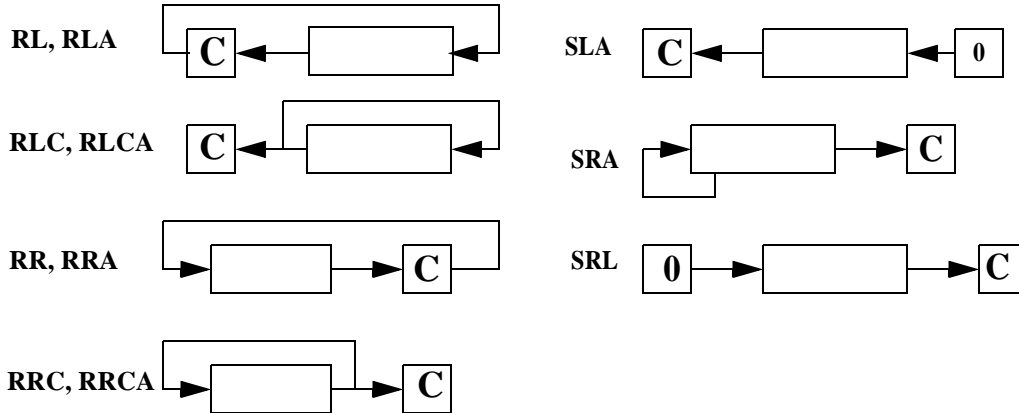
Instruction	clk	A	I	S	Z	V	C	Operation
DEC (HL)	8	f	b	*	*	v	-	(HL) = (HL) - 1
DEC (IX+d)	12	f	b	*	*	v	-	(IX+d) = (IX+d) -1
DEC (IY+d)	12	f	b	*	*	v	-	(IY+d) = (IY+d) -1
DEC r	2	fr	*	*	v	-	-	r = r - 1
INC (HL)	8	f	b	*	*	v	-	(HL) = (HL) + 1
INC (IX+d)	12	f	b	*	*	v	-	(IX+d) = (IX+d) + 1
INC (IY+d)	12	f	b	*	*	v	-	(IY+d) = (IY+d) + 1
INC r	2	fr	*	*	v	-	-	r = r + 1

## 18.13 8-bit Fast A register Operations

Instruction	clk	A	I	S	Z	V	C	Operation
CPL	2	r	-	-	-	-	-	A = ~A
NEG	4	fr	*	*	v	*	-	A = 0 - A

RLA	2	fr	-	-	-	*	{CY,A} = {A,CY}
RLCA	2	fr	-	-	-	*	A = {A[6,0],A[7]}; CY = A[7]
RRA	2	fr	-	-	-	*	{A,CY} = {CY,A}
RRCA	2	fr	-	-	-	*	A = {A[0],A[7,1]}; CY = A[0]

## 18.14 8-bit Shifts and Rotates



Instruction	clk	A	I	S	Z	V	C	Operation
RL (HL)	10	f	b	*	*	L	*	{CY,(HL)} = {(HL),CY}
RL (IX+d)	13	f	b	*	*	L	*	{CY,(IX+d)} = {(IX+d),CY}
RL (IY+d)	13	f	b	*	*	L	*	{CY,(IY+d)} = {(IY+d),CY}
RL r	4	fr	*	*	*	L	*	{CY,r} = {r,CY}
RLC (HL)	10	f	b	*	*	L	*	(HL) = {(HL)[6,0],(HL)[7]}; CY = (HL)[7]
RLC (IX+d)	13	f	b	*	*	L	*	(IX+d) = {(IX+d)[6,0], (IX+d)[7]}; CY = (IX+d)[7]
RLC (IY+d)	13	f	b	*	*	L	*	(IY+d) = {(IY+d)[6,0], (IY+d)[7]}; CY = (IY+d)[7]
RLC r	4	fr	*	*	*	L	*	r = {r[6,0],r[7]}; CY = r[7]
RR (HL)	10	f	b	*	*	L	*	{(HL),CY} = {CY,(HL)}
RR (IX+d)	13	f	b	*	*	L	*	{(IX+d),CY} = {CY,(IX+d)}
RR (IY+d)	13	f	b	*	*	L	*	{(IY+d),CY} = {CY,(IY+d)}
RR r	4	fr	*	*	*	L	*	{r,CY} = {CY,r}
RRC (HL)	10	f	b	*	*	L	*	(HL) = {(HL)[0],(HL)[7,1]}; CY = (HL)[0]
RRC (IX+d)	13	f	b	*	*	L	*	(IX+d) = {(IX+d)[0], (IX+d)[7,1]}; CY = (IX+d)[0]
RRC (IY+d)	13	f	b	*	*	L	*	(IY+d) = {(IY+d)[0],( IY+d)[7,1]}; CY = (IY+d)[0]
RRC r	4	fr	*	*	*	L	*	r = {r[0],r[7,1]}; CY = r[0]
SLA (HL)	10	f	b	*	*	L	*	(HL) = {(HL)[6,0],0}; CY = (HL)[7]
SLA (IX+d)	13	f	b	*	*	L	*	(IX+d) = {(IX+d)[6,0],0}; CY = (IX+d)[7]
SLA (IY+d)	13	f	b	*	*	L	*	(IY+d) = {(IY+d)[6,0],0}; CY = (IY+d)[7]
SLA r	4	fr	*	*	*	L	*	r = {r[6,0],0}; CY = r[7]
SRA (HL)	10	f	b	*	*	L	*	(HL) = {(HL)[7],(HL)[7,1]}; CY = (HL)[0]

SRA (IX+d)	13	f	b	*	*	L	*	(IX+d) = {(IX+d)[7], (IX+d)[7,1]}; CY = (IX+d)[0]
SRA (IY+d)	13	f	b	*	*	L	*	(IY+d) = {(IY+d)[7], (IY+d)[7,1]}; CY = (IY+d)[0]
SRA r	4	fr	*	*	L	*	*	r = {r[7],r[7,1]}; CY = r[0]
SRL (HL)	10	f	b	*	*	L	*	(HL) = {0,(HL)[7,1]}; CY = (HL)[0]
SRL (IX+d)	13	f	b	*	*	L	*	(IX+d) = {0,(IX+d)[7,1]}; CY = (IX+d)[0]
SRL (IY+d)	13	f	b	*	*	L	*	(IY+d) = {0,(IY+d)[7,1]}; CY = (IY+d)[0]
SRL r	4	fr	*	*	L	*	*	r = {0,r[7,1]}; CY = r[0]

## 18.15 Instruction Prefixes

Instruction	clk	A	I	S	Z	V	C	Operation
ALTD	2		-	-	-	-	-	alternate register destination for next Instruction
IOE	2		-	-	-	-	-	I/O external prefix
IOI	2		-	-	-	-	-	I/O internal prefix

## 18.16 Block Move Instructions

Instruction	clk	A	I	S	Z	V	C	Operation
LDD	10		d	-	-	*	-	(DE) = (HL); BC = BC-1; DE = DE-1; HL = HL-1
LDDR	6+7i		d	-	-	*	-	if {BC != 0} repeat:
LDI	10		d	-	-	*	-	(DE) = (HL); BC = BC-1; DE = DE+1; HL = HL+1
LDIR	6+7i		d	-	-	*	-	if {BC != 0} repeat:

If any of the block move instructions are prefixed by an I/O prefix, the destination will be in the specified I/O space. Add 1 clock for each iteration for the prefix if the prefix is IOI (internal I/O). If the prefix is IOE, add 2 clocks plus the number of I/O wait states enabled. The V flag is set when BC transitions from 1 to 0. If the V flag is not set another step is performed for the repeating versions of the instructions. Interrupts can occur between different repeats, but not within an iteration equivalent to LDD or LDI. Return from the interrupt is to the first byte of the instruction which is the I/O prefix byte if there is one.

## 18.17 Control Instructions - Jumps and Calls

Instruction	clk	A	I	S	Z	V	C	Operation
CALL mn	12		-	-	-	-	-	(SP-1) = PCH; (SP-2) = PCL; PC = mn; SP = SP-2
DJNZ j	5	r	-	-	-	-	-	B = B-1; if {B != 0} PC = PC + j
JP (HL)	4		-	-	-	-	-	PC = HL
JP (IX)	6		-	-	-	-	-	PC = IX
JP (IY)	6		-	-	-	-	-	PC = IY
JP f,mn	7		-	-	-	-	-	if {f} PC = mn
JP mn	7		-	-	-	-	-	PC = mn

JR cc,e	5	- - - -	if {cc} PC = PC + e
JR e	5	- - - -	PC = PC + e (if e==0 next seq inst is executed)
LCALL xpc,mn	19	- - - -	(SP-1) = XPC; (SP-2) = PCH; (SP-3) = PCL; XPC=xpc;
LJP xpc,mn	10	- - - -	XPC=xpc; PC = mn
LRET	13	- - - -	PCL = (SP); PCH = (SP+1); XPC = (SP+2); SP = SP+3
RET	8	- - - -	PCL = (SP); PCH = (SP+1); SP = SP+2
RET f	8/2	- - - -	if {f} PCL = (SP); PCH = (SP+1); SP = SP+2
RETI	12	- - - -	IP = (SP); PCL = (SP+1); PCH = (SP+2); SP = SP+3
RST v	8	- - - -	(SP-1) = PCH; (SP-2) = PCL; SP = SP - 2; PC = {R,v} v=10,18,20,28,38 only

## 18.18 Miscellaneous Instructions

Instruction	clk	A	I	S	Z	V	C	Operation
CCF	2	f	-	-	-	-	*	CF = ~CF
IP 0	4		-	-	-	-		IP = {IP[5:0], 00}
IP 1	4		-	-	-	-		IP = {IP[5:0], 01}
IP 2	4		-	-	-	-		IP = {IP[5:0], 10}
IP 3	4		-	-	-	-		IP = {IP[5:0], 11}
IPRES	4		-	-	-	-		IP = {IP[1:0], IP[7:2]}
LD A,EIR	4	fr	*	*	-	-		A = EIR
LD A,IIR	4	fr	*	*	-	-		A = IIR
LD A,XPC	4	r	-	-	-	-		A = MMU
LD EIR,A	4		-	-	-	-		EIR = A
LD IIR,A	4		-	-	-	-		IIR = A
LD XPC,A	4		-	-	-	-		XPC = A
NOP	2		-	-	-	-		No Operation
POP IP	7		-	-	-	-		IP = (SP); SP = SP+1
PUSH IP	9		-	-	-	-		(SP-1) = IP; SP = SP-1
SCF	2	f	-	-	-	1		CF = 1
ZINTACK	10		-	-	-	-		(SP-1) = PCH; (SP-2) = PCL; SP = SP-2; IP = {IP[6:

## 18.19 Privileged Instructions

The privileged instructions are described in this section. Privilege means that an interrupt cannot take place between the privileged instruction and the following instruction.

The three instructions below are privileged.

```
ld sp,h1 ; load the stack pointer
ld sp,iy
ld sp,ix
```

The instructions to load the stack are privileged so that they can be followed by an instruction to load the stack segment (SSEG) register without the danger of an interrupt taking place with an incorrect association between the stack pointer and the stack segment register. For example,

```
ld sp,hl
ioi ld (STACKSEG),a
```

The following instructions are privileged.

```
ip 0      ; shift ip left and set priority 00 in bits 1,0
ip 1
ip 2
ip 3
ipres     ; rotate ip right 2 bits, restoring previous priority
pop ip    ; pop IP register from stack
```

The instructions to modify the IP register are privileged so that they can be followed by a return instruction that is guaranteed to execute before another interrupt takes place. This avoids the possibility of an ever-growing stack.

```
reti     ; pops IP from stack and then pops return address
```

The instruction `reti` can be used to set both the return address and the ip in a single instruction. If preceded by a `ld xpc`, a complete jump or call to a computed address can be done with no possible interrupt.

```
ld a,xpc ; get and set the xpc
ld xpc,a
```

The instruction `ld xpc,a` is privileged so that it can be followed by other code setting interrupt priority or program counter without an intervening interrupt.

```
bit b,(hl) ; test a bit in memory
```

The instruction `bit b,(hl)` is privileged to make it possible to implement a semaphore without disabling interrupts. The following sequence is used. A bit is a semaphore, and the first task to set the bit owns the semaphore and has a right to manipulate the resources associated with the semaphore.

```
bit b,(hl)
set b,(hl)
jp z,ihaveit
; here I don't have it
```

The set instruction has no effect on the flags. Since no interrupt takes place after the bit instruction, if the flag is zero that means that the semaphore was not set when tested by the bit instruction and that the set instruction has set the semaphore. If an interrupt was allowed between the bit and set instructions, another routine could set the semaphore and two routines could think that they both owned the semaphore.

## 19. Differences Rabbit vs. Z80/Z180 Instructions

The Rabbit is highly code compatible with the Z80 and Z180, and it is easy to port non I/O dependent code. The main areas of incompatibility are instructions that are concerned with I/O or particular hardware implementations. The more important instructions that were dropped from the Z80/Z180 are automatically simulated by an instruction sequence in the Dynamic C assembler. A few fairly useless instructions have been dropped and cannot be easily simulated. Code using these instructions should be rewritten.

The following Z80/Z180 instructions have been dropped and there is no exact substitute.

DAA, HALT, DI, EI, IM 0, IM 1, IM 2, OUT, IN, OUT0, IN0, SLP, OUTI, IND, OUTD, INIR, OTIR, INDR, OTDR, TESTIO, MLT SP, RRD, RLD, CPI, CPIR, CPD, CPDR

Most of these op codes deal with I/O devices and thus do not represent transportable code. The only opcodes that are not processor I/O related are MLT SP, DAA, RRD, RLD, CPI, CPIR, CPD, and CPDR. MLT SP is not a practical op code. The codes that are concerned with decimal arithmetic, DAA, RRD, and RLD, could be simulated, but the simulation is very inefficient. (The bit in the status register used for half carry is available and can be set and cleared using the push and pop af instructions to gain access.) Usually code that uses these instructions should be rewritten. The instructions CPI, CPIR, CPD, and CPDR are repeating compare instructions. These instructions are not very useful because the scan stops when equal compare is detected. Unequal compare would be more useful. They are difficult to simulate efficiently, so it is suggested that code using these instructions be rewritten, which in most cases should be quite easy.

The following op codes are dropped.

RST 0, RST 8, RST 30h

The remaining RST instructions are kept, but the interrupt vector is relocated to a variable location the base of which is established by the EIR register. RST can be simulated by a call instruction, but this is not done automatically by the assembler since most of these instructions are used for debugging by Dynamic C.

The following instruction has had its op code changed.

```
ex (sp),hl    - old opcode 0E3h,  new opcode - 0EDh-054h
```

The following instructions use different register names.

```
ld a,eir
ld eir,a      ; was R register
ld iir,a
ld a,iir      ; was I register
```

The following Z80/Z180 instructions have been dropped, but the code shown will be substituted automatically by the assembler (planned feature).

```
call cc,adr          jr (jp)  ncc,xxx ; reverse condition
                    call adr
xxx:
tst r ( (hl),n)      push de
                    push af
                    and r ( (hl), n)
                    pop de  ; get a in h
                    ld a,d
                    pop de
mlt hl               call .mlthl ; simulates z180 mlt hl
mlt de               call .mltde
mlt bc               call .mltbc
```



## 20. Instructions in Alphabetical Order With Binary Encoding

### Key

n-8-bit data item

d-8 bit offset

ss - 00-BC, 01-DE, 10-HL, 11-SP

r - 000-b, 001-C, 010-D, 011-E, 100-H, 101-L, 110- (hl), 111-A

f - alternate destination prefix permitted, only flags stored in F'

r - in "A" column - alternate destination prefix stores to alternate

s - in "I" column - I/O prefixes are allowed (IOE, IOI) I/O is source

d - in "I" column - I/O prefixes are allowed (IOE, IOI) I/O is dest

V - overflow flag set on overflow

L - overflow flag set if upper 4 bits of data word are zero

- - flag not affected

\* - flag is set by operation

Flag labels: S- sign, Z- zero, V- overflow, C- carry

Instruction	Byte 1	Byte 2	Byte 3	Byte 4	clk	A	I	S	Z	V	C
ADC A, (HL)	10001110				5	fr	s	*	*	V	*
ADC A, (IX+d)	11011101	10001110	----d---		9	fr	s	*	*	V	*
ADC A, (IY+d)	11111101	10001110	----d---		9	fr	s	*	*	V	*
ADC A,n	11001110	----n---			4	fr		*	*	V	*
ADC A,r	10001-r-				2	fr		*	*	V	*
ADC HL,ss	11101101	01ss1010			4	fr		*	*	V	*
ADD A, (HL)	10000110				5	fr	s	*	*	V	*
ADD A, (IX+d)	11011101	10000110	----d---		9	fr	s	*	*	V	*
ADD A, (IY+d)	11111101	10000110	----d---		9	fr	s	*	*	V	*
ADD A,n	11000110	----n---			4	fr		*	*	V	*
ADD A,r	10000-r-				2	fr		*	*	V	*
ADD HL,ss	00ss1001				2	fr		-	-	-	*
ADD IX,xx	11011101	00xx1001			4	f		-	-	-	*
ADD IY,yy	11111101	00yy1001			4	f		-	-	-	*
ADD SP,d	00100111	----d---			4	f		-	-	-	*
ALTD	01110110				2			-	-	-	-
AND (HL)	10100110				5	fr	s	*	*	L	0
AND (IX+d)	11011101	10100110	----d---		9	fr	s	*	*	L	0
AND (IY+d)	11111101	10100110	----d---		9	fr	s	*	*	L	0
AND HL,DE	11011100				2	fr		*	*	L	0
AND IX,DE	11011101	11011100			4	f		*	*	L	0
AND IY,DE	11111101	11011100			4	f		*	*	L	0
AND n	11100110	----n---			4	fr		*	*	L	0
AND r	10100-r-				2	fr		*	*	L	0
BIT b, (HL)	11001011	01-b-110			7	f	s	-	*	-	-
BIT b, (IX+d)	11011101	11001011	----d---	01-b-110	10	f	s	-	*	-	-
BIT b, (IY+d)	11111101	11001011	----d---	01-b-110	10	f	s	-	*	-	-
BIT b,r	11001011	01-b--r-			4	f		-	*	-	-
BOOL HL	11001100				2	fr		*	*	0	0
BOOL IX	11011101	11001100			4	f		*	*	0	0
BOOL IY	11111101	11001100			4	f		*	*	0	0
CALL mn	11001101	----n---	----m---		12			-	-	-	-
CCF	00111111				2	f		-	-	-	*
CP (HL)	10111110				5	f	s	*	*	V	*

CP (IX+d)	11011101	10111110	----d---	9	f	s	*	*	V	*
CP (IY+d)	11111101	10111110	----d---	9	f	s	*	*	V	*
CP n	11111110	-----n---		4	f		*	*	V	*
CP r	10111-r-			2	f		*	*	V	*
CPL	00101111			2	r	-	-	-	-	
DEC (HL)	00110101			8	f	b	*	*	V	-
DEC (IX+d)	11011101	00110101	----d---	12	f	b	*	*	V	-
DEC (IY+d)	11111101	00110101	----d---	12	f	b	*	*	V	-
DEC IX	11011101	00101011		4		-	-	-	-	
DEC IY	11111101	00101011		4		-	-	-	-	
DEC r	00-r-101			2	fr		*	*	V	-
DEC ss	00ss1011			2	r	-	-	-	-	
ss= 00-BC, 01-DE, 10-HL, 11-SP										
DJNZ j	00010000	--(j-2)-		5	r	-	-	-	-	
EX (SP),HL	11101101	01010100		15	r	-	-	-	-	
EX (SP),IX	11011101	11100011		15		-	-	-	-	
EX (SP),IY	11111101	11100011		15		-	-	-	-	
EX AF,AF'	00001000			2		-	-	-	-	
EX DE,HL	11101011			2	s	-	-	-	-	
EX DE',HL	11100011			2	s	-	-	-	-	
EX DE,HL'	01110110	11100011		4	s	-	-	-	-	
EX DE',HL'	01110110	11100011		4	s	-	-	-	-	
EXX	11011001			2		-	-	-	-	
INC (HL)	00110100			8	f	b	*	*	V	-
INC (IX+d)	11011101	00110100	----d---	12	f	b	*	*	V	-
INC (IY+d)	11111101	00110100	----d---	12	f	b	*	*	V	-
INC IX	11011101	00100011		4		-	-	-	-	
INC IY	11111101	00100011		4		-	-	-	-	
INC r	00-r-100			2	fr		*	*	V	-
INC ss	00ss0011			2	r	-	-	-	-	
ss= 00-BC, 01-DE, 10-HL, 11-SP										
IOE	11011011			2		-	-	-	-	
IOI	11010011			2		-	-	-	-	
IP 0	11101101	01000110		4		-	-	-	-	
IP 1	11101101	01010110		4		-	-	-	-	
IP 2	11101101	01001110		4		-	-	-	-	
IP 3	11101101	01011110		4		-	-	-	-	
IPRES	11101101	01011101		4		-	-	-	-	
JP (HL)	11101001			4		-	-	-	-	
JP (IX)	11011101	11101001		6		-	-	-	-	
JP (IY)	11111101	11101001		6		-	-	-	-	
JP f,mn	11-f-010	----n---	----m---	7		-	-	-	-	
JP mn	11000011	----n---	----m---	7		-	-	-	-	
JR cc,e	001cc000	--(e-2)-		5		-	-	-	-	
JR e	00011000	--(e-2)-		5		-	-	-	-	
Note: If byte following op code is zero next sequential instruction is executed. If byte is -2 (11111110) jr is to itself.										
LCALL nbr,mn	11001111	---n---	---m---	xpc---	19		-	-	-	-
LD (BC),A	00000010			7	d	-	-	-	-	
LD (DE),A	00010010			7	d	-	-	-	-	
LD (HL),n	00110110	----n---		7	d	-	-	-	-	
LD (HL),r	01110-r-			6	d	-	-	-	-	
LD (HL+d),HL	11011101	11110100	----d---	13	d	-	-	-	-	
LD (IX+d),HL	11110100	----d---		11	d	-	-	-	-	

LD (IX+d),n	11011101	00110110	----d---	----n---	11	d	-	-	-	-
LD (IX+d),r	11011101	01110-r-	----d---		10	d	-	-	-	-
LD (IY+d),HL	11111101	11110100	----d---		13	d	-	-	-	-
LD (IY+d),n	11111101	00110110	----d---	----n---	11	d	-	-	-	-
LD (IY+d),r	11111101	01110-r-	----d---		10	d	-	-	-	-
LD (mn),A	00110010	----n---	----m---		10	d	-	-	-	-
LD (mn),HL	00100010	----n---	----m---		13	d	-	-	-	-
LD (mn),IX	11011101	00100010	----n---	----m---	15	d	-	-	-	-
LD (mn),IY	11111101	00100010	----n---	----m---	15	d	-	-	-	-
LD (mn),ss	11101101	01ss0011	----n---	----m---	15	d	-	-	-	-
LD (SP+n),HL	11010100	----n---			11		-	-	-	-
LD (SP+n),IX	11011101	11010100	----n---		13		-	-	-	-
LD (SP+n),IY	11111101	11010100	----n---		13		-	-	-	-
LD A,(BC)	00001010				6	r	s	-	-	-
LD A,(DE)	00011010				6	r	s	-	-	-
LD A,(mn)	00111010	----n---	----m---		9	r	s	-	-	-
LD A,EIR	11101101	01010111			4	fr	*	*	-	-
LD A,IIR	11101101	01011111			4	fr	*	*	-	-
LD A,XPC	11101101	01110111			4	r		-	-	-
LD dd,(mn)	11101101	01dd1011	----n---	----m---	13	r	s	-	-	-
LD dd',BC	11101101	01dd1001			4			-	-	-
LD dd',DE	11101101	01dd0001			4			-	-	-
LD dd,mn	00dd0001	----n---	----m---		6	r		-	-	-
ld bc,mn	00000001	...								
ld de,mn	00010001	...								
ld hl,mn	00100001	...								
ld sp,mn	00110001	...								
LD EIR,A	11101101	01000111			4			-	-	-
LD HL,(HL+d)	11011101	11100100	----d---		11	r	s	-	-	-
LD HL,(IX+d)	11100100	----d---			9	r	s	-	-	-
LD HL,(IY+d)	11111101	11100100	----d---		11	r	s	-	-	-
LD HL,(mn)	00101010	----n---	----m---		11	r	s	-	-	-
LD HL,(SP+n)	11000100	----n---			9	r		-	-	-
LD HL,IX	11011101	01111100			4	r		-	-	-
LD HL,IY	11111101	01111100			4	r		-	-	-
LD IIR,A	11101101	01001111			4			-	-	-
LD IIR,A	11101101	01001111			4			-	-	-
LD IX,(mn)	11011101	00101010	----n---	----m---	13		s	-	-	-
LD IX,(SP+n)	11011101	11000100	----n---		11			-	-	-
LD IX,HL	11011101	01111101			4			-	-	-
LD IX,mn	11011101	00100001	----n---	----m---	8			-	-	-
LD IY,(mn)	11111101	00101010	----n---	----m---	13		s	-	-	-
LD IY,(SP+n)	11111101	11000100	----n---		11			-	-	-
LD IY,HL	11111101	01111101			4			-	-	-
LD IY,mn	11111101	00100001	----n---	----m---	8			-	-	-
LD r,(HL)	01-r-110				5	r	s	-	-	-
LD r,(IX+d)	11011101	01-r-110	----d---		9	r	s	-	-	-
LD r,(IY+d)	11111101	01-r-110	----d---		9	r	s	-	-	-
LD r,g	01-r--r'				2	r		-	-	-
LD r,n	00-r-110	----n---			4	r		-	-	-
LD SP,HL	11111001				2			-	-	-
LD SP,IX	11011101	11111001			4			-	-	-
LD SP,IY	11111101	11111001			4			-	-	-
LD XPC,A	11101101	01100111			4			-	-	-

LDD	11101101	10101000			10	d	-	-	*	-
LDDR	11101101	10111000			6+7i	d	-	-	*	-
LDI	11101101	10100000			10	d	-	-	*	-
LDIR	11101101	10110000			6+7i	d	-	-	*	-
LDP (HL),HL	11101101	01100100			12		-	-	-	-
LDP (IX),HL	11011101	01100100			12		-	-	-	-
LDP (IY),HL	11111101	01100100			12		-	-	-	-
LDP (mn),HL	11101101	01100101	----n---	----m---	15		-	-	-	-
LDP (mn),IX	11011101	01100101	----n---	----m---	15		-	-	-	-
LDP (mn),IY	11111101	01100101	----n---	----m---	15		-	-	-	-
LDP HL,(HL)	11101101	01101100			10		-	-	-	-
LDP HL,(IX)	11011101	01101100			10		-	-	-	-
LDP HL,(IY)	11111101	01101100			10		-	-	-	-
LDP HL,(mn)	11101101	01101101	----n---	----m---	13		-	-	-	-
LDP IX,(mn)	11011101	01101101	----n---	----m---	13		-	-	-	-
LDP IY,(mn)	11111101	01101101	----n---	----m---	13		-	-	-	-
LJP nbr,mn	11000111	----n---	----m---	--nbr---	10		-	-	-	-
LRET	11101101	01000101			13		-	-	-	-
MUL	11110111				12		-	-	-	-
NEG	11101101	01000100			4	fr	*	*	V	*
NOP	00000000				2		-	-	-	-
OR (HL)	10110110				5	fr	s	*	*	L 0
OR (IX+d)	11011101	10110110	----d---		9	fr	s	*	*	L 0
OR (IY+d)	11111101	10110110	----d---		9	fr	s	*	*	L 0
OR HL,DE	11101100				2	fr	*	*	L	0
OR IX,DE	11011101	11101100			4	f	*	*	L	0
OR IY,DE	11111101	11101100			4	f	*	*	L	0
OR n	11110110	----n---			4	fr	*	*	L	0
OR r	10110-r-				2	fr	*	*	L	0
POP IP	11101101	01111110			7		-	-	-	-
POP IX	11011101	11100001			9		-	-	-	-
POP IY	11111101	11100001			9		-	-	-	-
POP zz	11zz0001				7	r	-	-	-	-
PUSH IP	11101101	01110110			9		-	-	-	-
PUSH IX	11011101	11100101			12		-	-	-	-
PUSH IY	11111101	11100101			12		-	-	-	-
PUSH zz	11zz0101				10		-	-	-	-
RES b,(HL)	11001011	10-b-110			10	d	-	-	-	-
RES b,(IX+d)	11011101	11001011	----d---	10-b-110	13	d	-	-	-	-
RES b,(IY+d)	11111101	11001011	----d---	10-b-110	13	d	-	-	-	-
RES b,r	11001011	10-b--r-			4	r	-	-	-	-
RET	11001001				8		-	-	-	-
RET f	11-f-000				8/2		-	-	-	-
RETI	11101101	01001101			12		-	-	-	-
RL (HL)	11001011	00010110			10	f	b	*	*	L *
RL (IX+d)	11011101	11001011	----d---	00010110	13	f	b	*	*	L *
RL (IY+d)	11111101	11001011	----d---	00010110	13	f	b	*	*	L *
RL DE	11110011				2	fr	*	*	L	*
RL r	11001011	00010-r-			4	fr	*	*	L	*
RLA	00010111				2	fr	-	-	-	*
RLC (HL)	11001011	00000110			10	f	b	*	*	L *
RLC (IX+d)	11011101	11001011	----d---	00000110	13	f	b	*	*	L *
RLC (IY+d)	11111101	11001011	----d---	00000110	13	f	b	*	*	L *
RLC r	11001011	00000-r-			4	fr	*	*	L	*

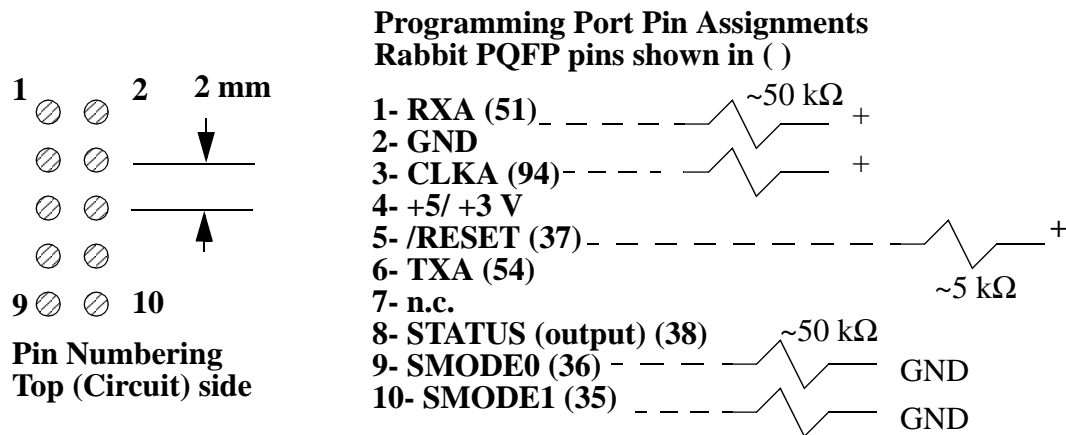
RLCA	00000111			2	fr	- - - *
RR (HL)	11001011	00011110		10	f b	* * L *
RR (IX+d)	11011101	11001011	----d---	00011110	13	f b * * L *
RR (IY+d)	11111101	11001011	----d---	00011110	13	f b * * L *
RR DE	11111011			2	fr	* * L *
RR HL	11111100			2	fr	* * L *
RR IX	11011101	11111100		4	f	* * L *
RR IY	11111101	11111100		4	f	* * L *
RR r	11001011	00011-r-		4	fr	* * L *
RRA	00011111			2	fr	- - - *
RRC (HL)	11001011	00001110		10	f b	* * L *
RRC (IX+d)	11011101	11001011	----d---	00001110	13	f b * * L *
RRC (IY+d)	11111101	11001011	----d---	00001110	13	f b * * L *
RRC r	11001011	00001-r-		4	fr	* * L *
RRCA	00001111			2	fr	- - - *
RST v	11-v-111	[v=2,3,4,5,7 only]		8		- - - -
SBC (IX+d)	11011101	10011110	----d---	9	fr s	* * V *
SBC (IY+d)	11111101	10011110	----d---	9	fr s	* * V *
SBC A,(HL)	10011110			5	fr s	* * V *
SBC A,n	11011110	----n---		4	fr	* * V *
SBC A,r	10011-r-			2	fr	* * V *
SBC HL,ss	11101101	01ss0010		4	fr	* * V *
SCF	00110111			2	f	- - - 1
SET b,(HL)	11001011	11-b-110		10	b	- - - -
SET b,(IX+d)	11011101	11001011	----d---	11-b-110	13	b - - - -
SET b,(IY+d)	11111101	11001011	----d---	11-b-110	13	b - - - -
SET b,r	11001011	11-b--r-		4	r	- - - -
SLA (HL)	11001011	00100110		10	f b	* * L *
SLA (IX+d)	11011101	11001011	----d---	00100110	13	f b * * L *
SLA (IY+d)	11111101	11001011	----d---	00100110	13	f b * * L *
SLA r	11001011	00100-r-		4	fr	* * L *
SRA (HL)	11001011	00101110		10	f b	* * L *
SRA (IX+d)	11011101	11001011	----d---	00101110	13	f b * * L *
SRA (IY+d)	11111101	11001011	----d---	00101110	13	f b * * L *
SRA r	11001011	00101-r-		4	fr	* * L *
SRL (HL)	11001011	00111110		10	f b	* * L *
SRL (IX+d)	11011101	11001011	----d---	00111110	13	f b * * L *
SRL (IY+d)	11111101	11001011	----d---	00111110	13	f b * * L *
SRL r	11001011	00111-r-		4	fr	* * L *
SUB (HL)	10010110			5	fr s	* * V *
SUB (IX+d)	11011101	10010110	----d---	9	fr s	* * V *
SUB (IY+d)	11111101	10010110	----d---	9	fr s	* * V *
SUB n	11010110	----n---		4	fr	* * V *
SUB r	10010-r-			2	fr	* * V *
XOR (HL)	10101110			5	fr s	* * L 0
XOR (IX+d)	11011101	10101110	----d---	9	fr s	* * L 0
XOR (IY+d)	11111101	10101110	----d---	9	fr s	* * L 0
XOR n	11101110	----n---		4	fr	* * L 0
XOR r	10101-r-			2	fr	* * L 0
ZINTACK (interrupt)				10		- - - -



# Appendix A

## A.1 Rabbit Programming Port

The programming port provides a standard physical and electrical interface between a Rabbit-based system and the Dynamic C programming platform. A special interface cable and converter connects a PC serial port to the programming port. The programming port is implemented by means of a 10-pin standard 2 mm connector. (Of course the user can change the physical implementation of the connector if he so desires.) With this setup the PC can communicate with the target, reset it and reboot it. The DTR line on the PC serial interface is used to drive the target reset line, which should be drivable by an external CMOS driver. The STATUS pin is used to by the Rabbit-based target to request attention when a breakpoint is encountered in the target under test. The SMODE pins are pulled up by a +5/+3 volt level from the interface. They should be pulled down on the board when the interface is not in use by approximately 5k resistors to ground. The target under test provides the +5 V or +3 V to the interface cable which is used to power the RS-232 driver and receiver.



**Figure 42. Rabbit Programming Port**

### A.1.1 Use of the Programming Port as a Diagnostic/Setup Port

The programming port, which is already in place, can serve as a convenient communications port for field setup, diagnosis or other occasional communication need (for example, diagnostic port). There are several ways that the port can be automatically integrated into the user's software scheme. If the purpose of the port is simply to perform a setup function, that is, write setup information to flash memory, then the controller can be reset through the programming port and a cold boot performed to start execution of a special program dedicated to this functionality.

The standard programming cable adapter connects the programming interface to a PC programming port. The /RESET line can be asserted by manipulating DTR on the PC serial port and the STATUS line can be read by the PC as DSR on the serial port. The PC can re-

start the target by pulsing reset and then, after a short delay, sending a special character string at 2400 bps. To simply restart the BIOS, the string 24h, 80h, 80h can be sent. When the BIOS is started, it can tell if the programming cable is connected because the SMODE1, SMODE0 pins are sensed as high. This will cause the BIOS to think that it should enter programming mode. The Dynamic C programming mode then can have an escape message that will enable the diagnostic serial port function.

Another approach to enabling the diagnostic port is to poll the serial port periodically to see if communication needs to begin or to enable the port and wait for interrupts. The SMODEx pins can be used for signaling and can be detected by a poll. However, recall that the SMODEx pins have a special function after reset and will inhibit normal reset behavior if not held low. The pull-up resistors on RXA and CLKA prevent spurious data reception that might take place if the pins floated.

If the clocked serial mode is used, the serial port can be driven by having two toggling lines that can be driven and one line that can be sensed. This allows a conversation with a device that does not have an asynchronous serial port but that has two output signal lines and one input signal line.

The line TXA (also called PC6) is zero after reset if cold boot mode is not enabled. A possible way to detect the presence of a cable on the programming port is for the cable to connect TXA to one of the SMODE pins and then test for the connection by raising PC6 and reading the SMODE pin after the cold boot mode has been disabled.

### **A.1.2 Alternate Programming Port**

The programming port uses serial port A. If the user needs to use serial port A in his application, an alternate method of programming is possible using the same 10-pin programming port. For his own application the user should use the alternate I/O pins for port A that share pins with parallel port D. The TXA and RXA pins on the 10-pin programming port are then a parallel port output and parallel port input using pins 6 and 7 on parallel port C. Using these two ports plus the STATUS pin as an output clock the user can create a synchronous clocked communication port using instructions to toggle the clock and data. Another Rabbit-based board can be used to translate the clocked serial signal to an asynchronous signal suitable for the PC. Since the target controls the clock for both send and receive, the data transmission proceeds at a rate controlled by the target board under development.

This scheme does not allow for an interrupt, and it is not desirable to use up an external interrupt for this purpose. The serial port may be used, if desired, During program load because there is no conflict with the user's program at compile load time. However, the user's program will conflict during debugging. The nature of the transmissions during debugging is such that the user program starts at a break point or otherwise wants to get the attention of the PC. The other type of message is when the PC wants to read or write target memory while the target is running.



The target toggling the clock can simply send a clocked serial message to get the attention of the PC. The intermediate communications board can accept these unsolicited messages using its clocked serial port. To prevent overrunning the receiver, the target can wait for a handshake signal on one of the SMODE lines or there can be suitable pre-arranged delays.

If the PC wants attention from the target it can set a line to request attention (SMODEx). The target will detect this line in the periodic interrupt routine and handle the complete message in the periodic interrupt routine. This may slow down target execution, but the interrupts will be enabled on the target while the message is read. The intermediate board could split long messages into a series of shorter messages if this is a problem.

## **A.2 Suggested Rabbit Crystal Frequencies**

Table 44 on page 134 provides a list of suggested Rabbit operating frequencies. The crystal can be half the operating frequency if the clock doubler is used up to approximately 25 MHz. Beyond this operating clock speed, it is necessary to use an X1 crystal or an external oscillator because asymetry in the waveform generated by the oscillator becomes a variation in the clock speed if the clock speed is doubled.



## **Legal Notice**

Rabbit Semiconductor products are not authorized for use as critical components in life-support devices or systems unless a specific written agreement regarding such intended use is entered into between the customer and Rabbit Semiconductor prior to use. Life-support devices or systems are devices or systems intended for surgical impantation into the body or to sustain life, and whose failure to perform, when properly used in accordance with instructions for use provided in the labeling and user's manual, can be reasonably expected to result in significant injury.

No complex software or hardware system is perfect. Bugs are always present in a system of any size. In order to prevent danger to life or property, it is there responsibility of the system designer to incorporate redundant protective mechanisms appropriate to the risk involved.