



PRODUCT MANUAL

Rabbit[®] 4000 Microprocessor Designer's Handbook

019-0156_H

Rabbit[®] 4000 Microprocessor Designer's Handbook

Part Number 019-0156 • Printed in the U.S.A.

Digi International Inc. © 2007-2010 • All rights reserved.

Digi International Inc. reserves the right to make changes and improvements to its products without providing notice.

Trademarks

Rabbit[®] Dynamic C[®] and RabbitCore[®] are registered trademarks of Digi International Inc.

Windows[®] is a registered trademark of Microsoft Corporation.

The latest revision of this manual is available at www.rabbit.com.

TABLE OF CONTENTS

Chapter 1. Introduction	7
1.1 Summary of Design Conventions	7
Chapter 2. Rabbit Hardware Design Overview	9
2.1 Design Conventions	9
2.1.1 Rabbit Programming Connector	10
2.1.2 Memory Chips.....	10
2.1.3 Oscillator Crystals.....	10
2.2 ESD Design Guidelines.....	11
2.3 Operating Voltages	11
2.4 Power Consumption	11
2.5 Through-Hole Technology	12
2.6 Moisture Sensitivity	12
Chapter 3. Core Design and Components.....	13
3.1 Clocks.....	13
3.2 Floating Inputs.....	14
3.3 Basic Memory Design.....	15
3.3.1 Memory Access Time	16
3.3.2 Interfacing External I/O with Rabbit 4000 Designs.....	16
3.4 PC Board Layout and Memory Line Permutation	17
3.5 PC Board Layout and Electromagnetic Interference.....	17
3.5.1 Rabbit 4000 Low EMI Features.....	17
Chapter 4. How Dynamic C Cold Boots the Target System.....	19
4.1 How the Cold Boot Mode Works In Detail.....	20
4.2 Program Loading Process Overview.....	21
4.2.1 Program Loading Process Details	21
Chapter 5. Rabbit Memory Organization.....	23
5.1 Physical Memory.....	23
5.1.1 Flash Memory	23
5.1.2 SRAM	24
5.1.3 Basic Memory Configuration.....	24
5.2 Memory Segments.....	25
5.2.1 Definition of Terms.....	26
5.2.2 The Base (or Root) Segment.....	27
5.2.2.1 Types of Code Best-Suited for the Base Segment	27
5.2.3 The Data Segment.....	27

5.2.4	The Stack Segment	27
5.2.5	The Extended Memory Segment	28
5.3	Separate I&D Space	29
5.3.1	Enable Separate I&D Space	31
5.3.2	Separate I&D Space Mappings in Dynamic C	31
5.3.2.1	Compiling to RAM	32
5.3.2.2	Compiling to Flash	33
5.3.3	Customizing Interrupts	34
5.4	How The Compiler Compiles to Memory	34
5.4.1	Placement of Code in Memory	34
5.4.2	Paged Access in Extended Memory	34
5.5	Memory Planning	35
5.5.1	Flash	35
5.5.2	Static RAM	35
Chapter 6.	The Rabbit BIOS	37
6.1	Startup Conditions Set by the BIOS	38
6.1.1	Registers Initialized in the BIOS	38
6.1.2	Origins	38
6.2	BIOS Flowchart	39
6.3	Internally-Defined Macros	40
6.4	Modifying the BIOS	40
6.4.1	Macros that Affect the BIOS	41
6.4.2	Advanced Options	42
6.5	Memory Mapping in Dynamic C	43
6.5.1	Origins Starting with Dynamic C 10.21	43
6.5.1.1	Example of Origin Declarations	44
6.5.1.2	Origin Declaration Syntax	47
6.5.1.3	Origin Declaration Semantics	47
6.5.1.4	Origin Declaration Start and End Syntax	50
6.5.1.5	Origin Application Syntax	50
6.5.1.6	Origin Macro Declaration Syntax	50
6.5.2	Origins Prior to Dynamic C 10.21	51
6.5.2.1	Origin Directive Semantics	51
6.5.2.2	Defining a Memory Region	52
6.5.2.3	Action Qualifiers	52
6.5.2.4	I&D Qualifiers	52
6.5.2.5	Follow Qualifiers	52
6.5.2.6	Origin Directive Examples	54
6.5.2.7	Origin Directives in Program Code	54
6.5.2.8	Origin Directive to Reserve Blocks of Memory	55
Chapter 7.	The System Identification and User Blocks	57
7.1	System ID Block Details	58
7.1.1	Definition of SysIDBlock	58
7.1.2	Reading the System ID Block	60
7.1.2.1	Determining the Existence of the System ID Block	61
7.1.3	Writing the System ID Block	63
7.2	User Block Details	63
7.2.1	Boot Block Issues	63

7.2.2 Reserved Flash Space.....	64
7.2.3 Reading the User Block	65
7.2.4 Writing the User Block	67
Chapter 8. BIOS Support for Program Cloning.....	69
8.1 Overview of Cloning.....	69
8.2 Creating a Clone.....	70
8.2.1 Steps to Enable and Set Up Cloning	70
8.2.2 Steps to Perform Cloning.....	70
8.2.3 LED Patterns	70
8.3 Cloning Questions	71
8.3.1 MAC Address	71
8.3.2 Different Flash Types	71
8.3.3 Different Memory Sizes.....	71
8.3.4 Design Restrictions	71
Chapter 9. Low-Power Design and Support	73
9.1 Details of the Rabbit 4000 Low-Power Features	74
9.1.1 Special Chip Select Features.....	74
9.1.2 Reducing Clock Speed	75
9.1.3 Preferred Crystal Configuration.....	75
9.2 To Further Decrease Power Consumption	76
9.2.1 What To Do When There is Nothing To Do	76
9.2.2 Sleepy Mode	76
9.2.3 External 32 kHz Oscillator.....	77
9.2.4 Conformal Coating of 32.768 kHz Oscillator Circuit.....	77
9.2.5 Software Support for Sleepy Mode.....	77
9.2.6 Baud Rates in Sleepy Mode.....	78
9.2.7 Debugging in Sleepy Mode.....	78
Chapter 10. Supported Flash Memories	79
10.1 Supporting Other Flash Devices	79
10.2 Writing Your Own Flash Driver.....	80
Chapter 11. Troubleshooting Tips for New Rabbit-Based Systems	81
11.1 Initial Checks.....	81
11.2 Diagnostic Tests.....	81
11.2.1 Program to Transmit Diagnostic Tests	81
11.2.2 Diagnostic Test #1: Toggle the Status Pin.....	83
11.2.2.1 Using serialIO.exe	83
11.2.3 Diagnostic Test #2	84
Appendix A. Supported Rabbit 4000 Baud Rates	89
Index	91

1. Introduction

This manual is intended for the engineer designing a system using the Rabbit 4000 microprocessor and Rabbit's Dynamic C development environment. It explains how to develop a system that is based on the Rabbit 4000 and can be programmed with Dynamic C.

With Rabbit 4000 microprocessors and Dynamic C, many traditional tools and concepts are obsolete. Complicated and fragile in-circuit emulators are unnecessary. EPROM burners are not needed. Rabbit 4000 microprocessors and Dynamic C work together without elaborate hardware aids, provided that the designer observes certain design conventions.

For all topics covered in this manual, further information is available in the *Rabbit 4000 Microprocessor User's Manual*.

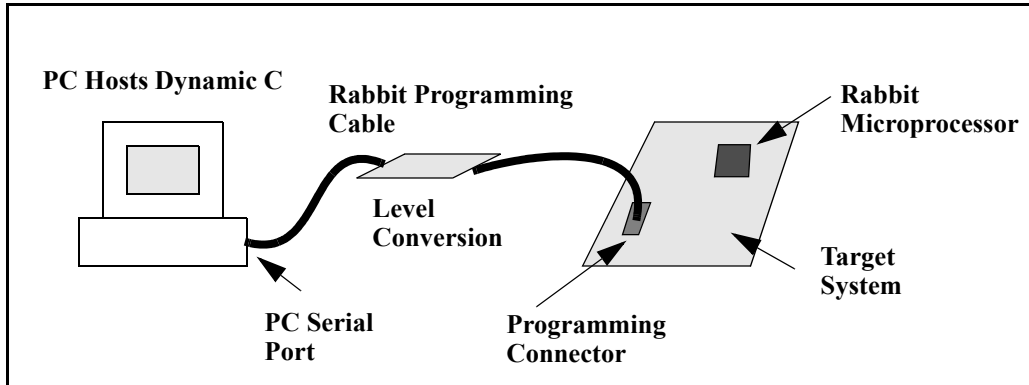
1.1 Summary of Design Conventions

Rabbit-based systems should be designed using the following conventions:

- Include a programming connector.
- Connect a static RAM having at least 128 KB to the Rabbit 4000 using /CS1, /OE1 and /WE1.
- Connect a flash memory that is on the approved list and has at least 128 KB of storage to the Rabbit 4000 using /CS0, /OE0 and /WE0.
- Install a crystal oscillator with a frequency of 32.768 kHz to drive the battery-backable clock. (Battery-backing is optional, but the clock is used in the cold boot sequence to generate a known baud rate of 2400 bps.)
- Install a crystal or oscillator for the main processor clock that is a multiple of 614.4 kHz, or better, a multiple of 1.8432 MHz.
- Do not use pin PB1 in your design if cloning is to be used.
- Be sure unused inputs are not floating.

As shown in [Figure 1-1](#), the Rabbit programming cable connects a PC serial port to the programming connector of the target system. Dynamic C or the Rabbit Field Utility (RFU) runs as an application on the PC, and can cold boot the Rabbit 4000 based target system with no pre-existing program installed in the target. A USB to RS232 converter may also be used instead of a PC serial port. Rabbit 4000-based targets may also be programmed and debugged remotely over a local network or even the Internet using a RabbitLink card.

Figure 1-1 The Rabbit 4000 Microprocessor and Dynamic C



Dynamic C programming uses serial port A for software development. However, it is possible for the user's application to also use serial port A, with the restriction that debugging is not available.

2. Rabbit Hardware Design Overview

Because of the glueless nature of the external interfaces, especially the memory interface, it is easy to design hardware in a Rabbit 4000-based system. More details on hardware design are given in the *Rabbit 4000 Microprocessor User's Manual*.

2.1 Design Conventions

Rabbit-based systems designed using the following conventions will provide a hardware base that is compatible with running Dynamic C applications.

- Include a standard Rabbit programming cable. The standard 10-pin programming connector provides a connection to serial port A and allows the PC to reset and cold boot the target system.
- Connect a static RAM having at least 128 KB to the processor using /CS1, /OE1 and /WE1. It is useful if the PC board footprint can also accommodate a RAM large enough to hold all the code anticipated. Although code residing in some flash memory can be debugged, debugging and program download is faster to RAM.
- Connect a flash memory that is on the approved list and has at least 128 KB of storage to the processor using /CS0, /OE0 and /WE0. Non-approved memories can be used, but it may be necessary to modify several files. Some systems designed to have their program reloaded by an external agent on each power-up may not need any flash memory.
- Install a crystal oscillator with a frequency of 32.768 kHz to drive the battery-backable real-time clock (RTC), the watchdog timer (WDT) and the Periodic Interrupt.
- Install a crystal or oscillator for the main processor clock that is a multiple of 614.4 kHz, or better, a multiple of 1.8432 MHz. These preferred clock frequencies make possible the generation of standard serial baud rates. Common crystal frequencies to use are 7.3728 MHz, 11.0592 MHz, 14.7456 MHz, 22.1184 MHz, 29.4912 MHz or double these frequencies.

NOTE: The internal clock doubler can double these oscillations for a higher operating frequency.

- Digital I/O line PB1 should not be used in the design if cloning is to be used. PB1 should be pulled up with 50K or so pull up resistor if cloning is used. (See “[BIOS Support for Program Cloning](#)” on page 69 for more information on cloning.)

2.1.1 Rabbit Programming Connector

The user may be concerned that the requirement for a programming connector places added cost overhead on the design. The overhead is very small—less than \$0.25 for components and board space that could be eliminated if the programming connector were not made a part of the system.

The programming connector can also be used for a variety of other purposes, including user applications. A device attached to the programming connector has complete control over the system because it can perform a hardware reset and load new software. If this degree of control is not desired for a particular situation, then certain pins can be left unconnected in the connecting cable, limiting the functionality of the connector to serial communications. Rabbit develops products and software that assume the presence of the programming connector.

2.1.2 Memory Chips

Most systems have one static RAM chip and one or two flash memory chips, but more memory chips can be used when appropriate. Static RAM chips are available in 128K x 8, 256K x 8, and 512K x 8 sizes. They are all available in 3 V versions. Suggested flash memory chips between 128K x 8 and 512K x 8 are given in [Chapter 10, “Supported Flash Memories.”](#) That chapter also includes instructions for writing your own flash driver. The list of supported flash memories is in Technical Note 226, “Supported Flash Memories.”

Dynamic C and a PC are not necessary for the production programming of flash memory since the flash memory can be copied from one controller to another by *cloning*. This is done by connecting the system to be programmed to the same type of system that is already programmed. This connection is made with the Rabbit Cloning Board. The cloning board connects to the programming ports of both systems. A push of a button starts the transfer of the program and an LED displays the progress of the transfer.

Please visit www.rabbit.com/store/index.shtml to purchase the Rabbit Cloning Board.

2.1.3 Oscillator Crystals

Generally, a system will have two oscillator crystals:

- A 32.768 kHz crystal oscillator to drive the battery-backable timer,
- A crystal that has a frequency that is a multiple of 614.4 kHz or a multiple of 1.8432 MHz. Typical values are 7.3728, 11.0592, 14.7456, 22.1184, and 29.4912 MHz.

These crystal frequencies (except 614.4 kHz and 1.8432 MHz) allow generation of standard baud rates up to at least 115,200 bps. The clock frequency can be doubled by an on-chip clock doubler, but the doubler should not be used to achieve frequencies higher than about 60 MHz on a 3.3 V system. A quartz crystal should be used for the 32.768 kHz oscillator. For the main oscillator, a ceramic resonator that is accurate to 0.5% will usually be adequate and less expensive than a quartz crystal for lower frequencies.

2.2 ESD Design Guidelines

The following guidelines are recommended for designs incorporating a Rabbit 4000 processor with electrostatic discharge (ESD) sensitivity on VBAT. These guidelines are good recommendations for all Rabbit processors.

1. The 1.8 V supply for VBAT should be provided by a regulator with at least 2 kV ESD protection (human body model).
2. The 3.3 V supply should have smaller 0.1 μF , 0.01 μF , and 2.2 nF bypass capacitors throughout the layout. In addition, the 3.3 V supply should have a large value bulk capacitor (10 μF).

The power going to VBAT should also be protected by a diode and two resistors. See a schematic for a RabbitCore[®] module based on the Rabbit 4000 for more details.

2.3 Operating Voltages

The operating voltage in Rabbit 4000 based systems will usually be 1.8 V $\pm 10\%$ for the processor core and 3.3 V $\pm 10\%$ for the I/O. The I/O ring can also be run at 1.8 V $\pm 10\%$.

The maximum computation per watt is obtained in the range of 3.0 V to 3.6 V. The highest clock speed requires 3.3 V. The maximum clock speed with a 3.3 V supply is 54 MHz (26.7264 x 2), but it will usually be convenient to use a 14.7456 MHz crystal, doubling the frequency to 29.4912 MHz. Good computational performance, but not the absolute maximum, can be implemented for a 3.3 V system by using an 11.0592 crystal and doubling the frequency to 22.1184 MHz. Such a system will operate with 70 ns memories. A 29.4912 MHz system will require memories with 55 ns access time. A table of timing specification is in the *Rabbit 4000 Microprocessor User's Manual*.

2.4 Power Consumption

Various mechanisms contribute to the current consumption of the Rabbit 4000 processor while it is operating, including current that is proportional to the voltage alone (leakage current) and dependent on both voltage and frequency (switching and crossover current).

Table 2-1 shows typical current draw as a function of the main clock frequency. The values shown do not include any current consumed by external oscillators or memory. It is assumed that approximately 30 pF is connected to each address line.

NOTE: VDDCORE = 1.8 V $\pm 10\%$, VDDIO = 3.3 V $\pm 10\%$, TA = -40°C to 85°C

Table 2-1 Preliminary Current vs. Clock Frequency

Frequency (MHz)	I _{core} (mA)	I _{IO} (mA)	I _{Total} (mA)
7.3728	4	10	14
14.7456	6	11	17
29.4912	10	12	22
58.9824	18	15	33

2.5 Through-Hole Technology

Most design advice given for the Rabbit 4000 assumes the use of surface-mount technology. However, it is possible to use the older through hole technology and develop a Rabbit 4000 system. One can use a Rabbit 4000-based Core Module, a small circuit board with a complete Rabbit 4000 core that includes memory and oscillators. Another possibility is to solder the Rabbit 4000 processors by hand to the circuit board. This is not difficult and is satisfactory for low production volumes if the right technique is used.

2.6 Moisture Sensitivity

Surface-mount processing of plastic packaged components such as Rabbit microprocessors typically involves subjecting the package body to high temperatures and various chemicals such as solder fluxes and cleaning fluids during solder wave and reflow operations. The plastic molding compounds used for IC packaging (encapsulation) is hygroscopic, that is, it readily absorbs moisture. The amount of moisture absorbed by the package is proportional to the storage environment and the amount of time the package is exposed to the humidity in the environment. During the solder reflow process, the package is heated rapidly, and any moisture present in the package will vaporize rapidly, generating excessive internal pressures to various interfaces in the package. The vapors escaping from the package may cause cracks or delamination of the package. These cracks can propagate through the package or along the lead frame, thus exposing the die to ionic contaminants and increasing the potential for circuit failures. The damage to the package may or may not be visible to the naked eye. This condition is common to all plastic surface-mount components and is not unique to Rabbit microprocessors.

Rabbit microprocessors are shipped to customers in moisture-barrier bags with enough desiccant to maintain their contents below 20% relative humidity for up to 12 months from the date of seal. A reversible Humidity Indicator Card is enclosed to monitor the internal humidity level. The loaded bag is then sealed under a partial vacuum. The caution label (IPC/JEDEC J-STD-020, LEVEL 3) included with each bag outlines storage, handling, and bake requirements.

The requirements outlined on the label only apply to components that will be exposed to SMT processing. This means that completed board-level products that will not be subjected to the solder reflow processing do not have to be baked or sealed in special moisture barrier bags.

3. Core Design and Components

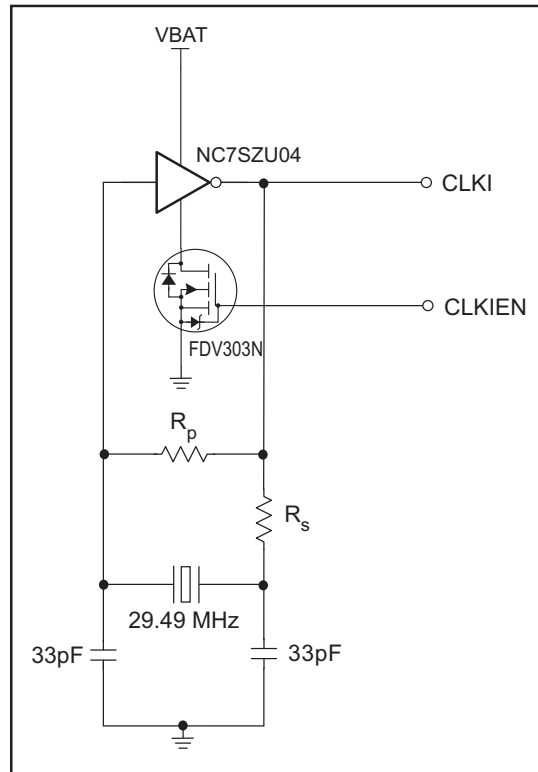
Core designs can be developed around the Rabbit 4000 microprocessor. A core design includes memory, the microprocessor, oscillator crystals, the Rabbit 4000 standard programming port, and in some cases, a power controller and power supply. Although modern designs usually use at least four-layer printed circuit boards, two-sided boards are a viable option with the Rabbit 4000, especially if the clock speed is not high and the I/O is intended to operate at 3.3 V—factors that reduce edge speed and electromagnetic radiation.

Schematics illustrating the use of the Rabbit 4000 microprocessor are available online via links in the manuals for the products that are using the Rabbit 4000. Each board-level or core module product has a user manual with an appendix labeled “Schematics.” Go to: www.rabbit.com and select “Product Documentation” from the “Support” tab; this will take you to a list of links for available user manuals.

3.1 Clocks

The Rabbit 4000 has input pins for both the fast clock and the 32.768 kHz clock. The fast clock drives the Rabbit 4000 CPU and peripheral clocks, whereas the 32.768 kHz clock is used for the battery-backable clock (also known as the real-time clock), the watchdog timer, the periodic interrupt timer and the asynchronous cold boot function.

Figure 3-1 Main Oscillator Circuit



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. The startup delay may be as much as 5 seconds, but will usually be about 200 ms. Crystals with low series resistance ($R < 35 \text{ k}\Omega$) will start faster.

For more information on the 32.768 kHz oscillator please see Technical Note 235, “External 32.768 kHz Oscillator Circuits.” This document is available on our website: www.rabbit.com.

3.2 Floating Inputs

Floating inputs or inputs that are not solidly either high or low can draw current because both N and P FETs can turn on at the same time. To avoid excessive power consumption, floating inputs should not be included in a design (except that some inputs may float briefly during power-on sequencing). Most unused inputs on the Rabbit 4000 can be made into outputs by proper software initialization to remove the floating property. Pull-up resistors will be needed on a few inputs that cannot be programmed as outputs. An alternative to a pull-up resistor is to tie an unused output to the unused inputs. If pull-up (or pull-down) resistors are required, they should be made as large as possible if the circuit in question has a substantial part of its duty cycle with current flowing through the resistor.

3.3 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.

For Dynamic C to work out of the box, the basic RAM memory must be connected to /CS1, /OE1, and /WE1.

/CS1 has a special property that makes it the preferred chip select for battery-backed RAM. The BIOS defined macro, `CS1_ALWAYS_ON`, may be redefined in the BIOS to 1 which will set a bit in the MMIDR register that forces /CS1 to stay enabled (low). 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 always be 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.

Table 3-1 Typical Interface between the Rabbit 4000 and Memory

Primary Flash	SRAM	Secondary Flash
/CS0, /OE0 and /WE0	/CS1, /OE1 and /WE1	/CS2, /OE0 and /WE0

3.3.1 Memory Access Time

Memory access time 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 the *Rabbit 4000 Microprocessor User's Manual*.

3.3.2 Interfacing External I/O with Rabbit 4000 Designs

The Rabbit 4000 provides on-chip facilities for glueless interfacing to many types of external I/O peripherals. The processor provides a common I/O read and I/O write strobe in addition to eight user configurable I/O strobes that can be used as read, write, read/write, or chip select signals. The Rabbit 4000 also provides the option of enabling a completely separate bus for I/O accesses. The Auxiliary I/O Bus, which uses many of the same pins used by Parallel Port A and the Slave Port, provides 8 data lines and 6 to 8 address lines that are active only during I/O operations. By connecting I/O devices to the auxiliary bus, the fast memory bus is relieved of capacitive loading that would otherwise slow down memory accesses. For core modules based on the Rabbit 4000, fewer pins are required to exit the core module since the slave port and the I/O bus can share the same pins and the memory bus no longer needs to exit the module to provide I/O capability.

As far as external I/O timing is concerned, the Rabbit 4000 provides:

- half a clock cycle of address and chip select hold time for I/O write operations, and
- zero clock cycles of address and chip select hold times for I/O read operations.

These can both be increased to a full clock of hold time. These hold times are true if an I/O device is interfaced to the common memory and I/O bus. However, if an I/O peripheral is interfaced to the Auxiliary I/O bus, address hold time is no longer an issue as the address does not change until the next external I/O operation.

For more information on I/O timing please refer to the *Rabbit 4000[®] Microprocessor User's Manual*.

Some I/O peripherals such as LCD controllers and Compact Flash devices require address and chip select hold times for both read and write operations. If the peripheral is interfaced to the Auxiliary I/O bus, address hold time is not an issue. If chip select hold time is required, an unused auxiliary I/O address line can be used to generate the chip select. In situations where I/O peripherals are interfaced to the common memory and I/O bus, address and chip select hold times can be extended under software control or with minor hardware changes. Please refer to Technical Note 227, "Interfacing External I/O with Rabbit 2000/3000 Designs" for additional information. This document is available online at:

www.rabbit.com/docs/app_tech_notes.shtml.

3.4 PC Board Layout and Memory Line Permutation

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 and must be avoided in practical systems.

3.5 PC Board Layout and Electromagnetic Interference

Most design failures are related to the layout of the PC board. A good layout results when the effects of electromagnetic interference (EMI) are considered. For detailed information regarding this subject please see Technical Note 221, “PC Board Layout Suggestion for the Rabbit 3000 Microprocessor.” This document is available at: www.rabbit.com/docs/app_tech_notes.shtml.

3.5.1 Rabbit 4000 Low EMI Features

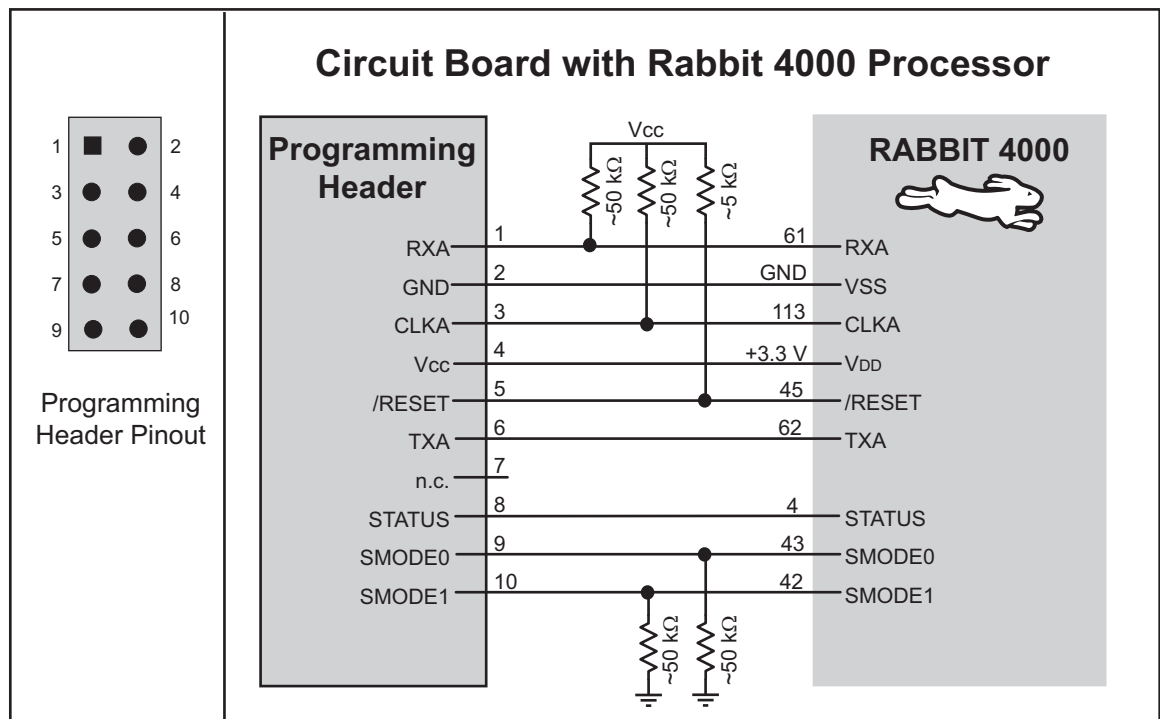
The Rabbit 4000 has powerful built-in features to minimize EMI. They are noted here. For details please see *The Rabbit 4000 Microprocessor User's Manual*.

- Separate power pins exist for core and I/O rings.
- The I/O bus can be separate from the memory bus.
- The external processor bus cycles are not all the same length.
- The external processor bus does not require running the clock around the PCB.
- The clock spectrum spreader option modulates the clock frequency.
- Some gated internal clocks are enabled only when needed.
- An internal clock doubler allows the external crystal oscillator to operate at 1/2 frequency.

4. How Dynamic C Cold Boots the Target System

Dynamic C assumes that target controller boards using the Rabbit 4000 CPU have no pre-installed firmware. It takes advantage of the Rabbit 4000's bootstrap (cold boot) mode, which allows memory and I/O writes to take place over the programming port.

Figure 4-1 Rabbit Programming Port



The Rabbit programming cable is a smart cable with an active circuit board in its middle. The circuit board converts RS-232 voltage levels used by the PC serial port to CMOS voltage levels used by the Rabbit 4000. The level converter is powered from the power supply voltage present on the Rabbit 4000 programming connector. Plugging the programming cable into the Rabbit programming connector results in pulling the Rabbit 4000 SMODE0 and SMODE1 (startup mode) lines high. This causes the Rabbit 4000 to enter the cold boot mode after reset.

When the programming cable connects a PC serial port to the target controller board, the PC running Dynamic C is connected to the Rabbit 4000 as shown in the table below.

Table 4-1 Programming Port Connections

PC Serial Port Signal	Rabbit 4000 Signal
DTR (output)	/RESET (input, reset system)
DSR (input)	STATUS (general purpose output)
TX (serial output)	RXA (serial input, port A)
RX (serial input)	TXA (serial output, port A)

When Dynamic C cold boots the Rabbit 4000-based target system it assumes that no program is already installed on the target. The flash memory on the target system may be blank or it may contain any data. The cold boot capability permits the use of soldered-in flash memory on the target. Soldered-in memory eliminates sockets, boot blocks and PROM programming devices.

4.1 How the Cold Boot Mode Works In Detail

Cold boot works by receiving triplets of bytes that consist of a high address byte followed by a low address byte, followed by a data byte, and writing the data byte to either memory or I/O space. Cold boot mode is entered by having one or both of the SMODE pins pulled high when the Rabbit is reset. The pin settings determine the source of the incoming triplets:

SMODE1 = 0, SMODE0 = 1 cold boot from slave port.

SMODE1 = 1, SMODE0 = 0 cold boot from clocked serial port A.

SMODE1 = 1, SMODE0 = 1 cold boot from asynchronous serial port A at 2400 bps.

SMODE1 = 0, SMODE0 = 0 start normal execution at address zero.

The SMODE pins can be used as general input pins once the cold boot is complete.

On entering cold boot mode, the microprocessor starts executing a 12-byte program contained in an internal ROM. The program contains the following code.

```

; origin zero
00  ld 1,n          ; n=0c0h for serial port A
                   ; n=020h for parallel (slave port)
02  ioi ld d,(hl)   ; get address most significant byte
04  ioi ld e,(hl)   ; get least significant byte
06  ioi ld a,(hl)   ; get data
08  ioi or nop      ; if the high bit of the MSB of the address is
1                               ; (i.e., d[7] ==1) then ioi, else nop
09  ld (de),A       ; store in memory or I/O
10  jr 0            ; jump back to zero

; note wait states inserted at bytes 3, 5 and 7 waiting
; for serial port or parallel port ready

```

The function of the boot ROM program depends on the settings of the pins SMODE0 and SMODE1 and on whether the high bit of the high address byte (first byte of a received triplet) that is loaded to register D is set. If bit 7 of the high address byte is set, then the data byte (last byte of the triplet) is written to I/O space when received. If the bit is clear, then the data byte gets written to memory. Boot mode is terminated by storing 80h to I/O register 24h, which causes an instruction fetch to begin at address zero.

Wait states are automatically inserted during the fetching of bytes 3, 5 and 7 to wait for the serial or parallel port ready. The wait states continue indefinitely until the serial port is ready. This will cause the processor to be in the middle of an instruction fetch until the next character is ready. While the processor is in this state the chip select, but not the output enable, will be enabled if the memory mapping registers are such as to normally enable the chip select for the boot ROM address. The chip select will stay low for extended periods while the processor is waiting for the serial or parallel port data to be ready.

4.2 Program Loading Process Overview

On start up, Dynamic C first uses the PC's DTR line on the serial port to assert the Rabbit 4000 RESET line and put the processor in cold boot mode. Next, Dynamic C uses a four stage process to load a user program:

1. Load an initial loader (cold loader) to RAM via triplets sent at 2400 baud from the PC to a target in cold boot mode.
2. Run the initial loader and load a secondary loader (pilot BIOS) to RAM at 57600 baud.
3. Run the secondary loader and load the BIOS and user program to flash after compiling them to a file, optionally negotiating with the Pilot BIOS to increase the baud rate to 115200 or higher so the loading can happen quickly.
4. Run the BIOS. Then run and debug the user program at the baud rate selected in Dynamic C.

NOTE: Step 4 is combined with step 3 when using 4 K (or greater) sector flash.

4.2.1 Program Loading Process Details

When Dynamic C starts to compile a program, the following sequence of events takes place:

1. The serial port is opened at 2400 baud with the DTR line high, and after a 500 ms delay, the DTR line is lowered. This pulses the reset line on the target low (the programming cable inverts the DTR line), placing the target into bootstrap mode.
2. A group of triplets defined in the file COLDLLOAD.BIN consisting of 2 address bytes and a data byte are sent to the target. The first few bytes sent are sent to I/O addresses to set up the MMU and MIU and do system initialization. The MMU is set up so that RAM is mapped to 0x00000, and flash is mapped to 0x80000.
3. The remaining triplets place a small initial loader program at memory location 0x00000. The last triplet sent is 0x80, 0x24, 0x80, which tells the CPU to ignore the SMODE pins and start running code at address 0x00000.
4. The initial loader measures the crystal speed to determine what divisor is needed to set a baud rate of 19200. The divisor is stored at address 0x3F02 for later use by the BIOS, and the programming port is set to 57600 baud.
5. The PC now bumps the baud rate on the serial port being used to 57600 baud.

6. The initial loader then reads 7 bytes from the serial port. First a 4-byte address field: the physical address to place the secondary loader, followed by a 2-byte length field: the number of bytes in the secondary loader. The 7th byte is a checksum (simple summation) of the previous 6 bytes. Whether or not the checksum matched, it is echoed back as an acknowledgement.
7. The data segment is then mapped to the given physical location, using the `DATASEG` register. The data segment boundary will also be set to `0x6000`, so the secondary loader will always be located at the same place in logical space, regardless of where it physically resides.
8. The initial loader finally enters a loop where it receives the specified number of bytes that compose the secondary loader program (`pilot.bin` sent by the PC) and writes those bytes starting at `0x6000` (logical). The first byte sent this way **MUST** be `0xCC`, as an indicator to the initial loader. This byte will be stored as `0x00` (nop), instead of `0xCC`. A 2-byte checksum will be sent after the secondary loader has been received, using the 8-bit Fletcher Algorithm (see RFC1145 for details), such that the load can be verified. After all of the bytes are received, and the checksum has been sent, program execution jumps to `0x6000`.
9. The secondary loader does a wrap-around test to determine how much RAM is available, and reads the flash and CPU IDs. This information is made available for transmittal to Dynamic C when requested.
10. The secondary loader now enters a finite state machine (FSM) that is used to implement the Dynamic C/Target Communications protocol. Dynamic C requests the CPU ID, flash ID, RAM size, and 19200 baud rate divisor to define internally defined constants and macros. Dynamic C uses the flash ID to lookup flash parameters that are sent back to the secondary loader so that it can initialize flash write/erase routines. At this stage, the compiler can request the baud rate be increased to a higher value. The secondary loader is now ready to load a BIOS and user program.
11. Dynamic C now compiles the BIOS and user programs. Both are compiled to a file, then the file is loaded to the target using the Pilot BIOS' FSM. After the loading is complete, Dynamic C, using the Pilot BIOS' FSM, tells the Pilot BIOS to map flash to `0x00000`, map RAM to `0x80000`, and start program execution at `0x0000`, thereby running the compiled BIOS.
12. If the Pilot BIOS detects a RAM compile or small-sector flash that uses sector-write mode, Dynamic C uses a slightly different loading procedure. The BIOS will be compiled as normal, and loaded using the Pilot BIOS. After the BIOS is loaded, Dynamic C will tell the Pilot BIOS to start it, and the rest of the program will be loaded through the compiled BIOS.
13. Once the compiled BIOS starts up, it runs some initialization code. This includes setting up the serial port for the debug baud rate (set in the Communications tab in Options | Project Options), setting up serial interrupts and starting the BIOS FSM. Dynamic C sets a breakpoint at the beginning of `main()` and runs the program up to the breakpoint. The board has been programmed, and Dynamic C is now in debug mode.
14. If the programming cable is removed and the target board is reset, the user's program will start running automatically because the BIOS will check the `SMODE` pins to determine whether to run the user application or enter the debug kernel.

5. Rabbit Memory Organization

The architecture of earlier Rabbit processors was derived from the original Z80 microprocessor. The original Z80 instruction set used 16-bit addresses to address a 64 KB memory space. All code and data had to fit in this 64 KB space. To expand the available memory space, the Rabbit 4000 adopts a scheme similar to that used by the Z180.

The 64 KB space is divided into segments and the Rabbit's Memory Mapping Unit (MMU) maps each segment to a block in a larger memory. The larger memory is 1 MB by default, although the Rabbit 4000 allows this larger address space to be resized. The segments are effectively windows to the larger memory. The view from the window can be adjusted so that the window looks at different blocks in the larger memory. Note also that the Rabbit 4000 has many new instructions that allow direct access to the larger memory space. [Figure 5-1](#) shows the memory mapping schematically.

NOTE: Please see Technical Note 202, “Rabbit Memory Management in a Nutshell,” for more details on how memory mapping works on the Rabbit 2000 and 3000. This document is available at:

www.rabbit.com/support/techNotes_whitePapers.shtml

5.1 Physical Memory

The Rabbit 4000 has a configurable physical address space. The default addressable space on the 4000 is 1 MB, the same as that used on the Rabbit 2000 and 3000. However, on the Rabbit 4000, the physical address space can be reconfigured to use additional address lines to resize the physical memory from 512 K to 16 MB. The physical memory can be increased to 4 MB without the use of additional address lines by mapping in 1 MB memory devices into the four available physical memory banks. In special circumstances more than 16 MB of memory can be installed and accessed using auxiliary memory mapping schemes. Typical Rabbit 4000 systems have two types of directly addressable physical memory: flash memory and static RAM.

5.1.1 Flash Memory

Flash memory in a Rabbit 4000-based system may be small-sector or large-sector type. Small-sector memory typically has sectors of 128 to 4096 bytes. Individual sectors may be separately erased and written. In large-sector memory the sectors are often 16 KB to 64 KB or more. Large-sector memory is less expensive and has faster access time. The best solution will usually be to lay out a design to accept several different types of flash memory, including the flexible small-sector memories and the fast large-sector memories.

Flash memory follows a write-once-in-a-while and read-frequently model. Depending on the particular type of flash used, the flash memory may wear out after it has been written approximately 10,000 to 100,000 times.

5.1.2 SRAM

Static RAM may or may not be battery-backed. If the SRAM is battery-backed it retains its data when primary power is disconnected. SRAM chips typically used for Rabbit systems are 128 KB, 256 KB, 512 KB, or 1 MB. With the configurable physical memory of the Rabbit 4000 and support in Dynamic C 10.21 and later versions, static RAM chips of 1 MB and larger may also be used.

When the memory is battery-backed, power is supplied at 2 V to 3 V from a battery. While preserving memory contents with battery power, the shutdown circuitry must keep the chip select line high.

5.1.3 Basic Memory Configuration

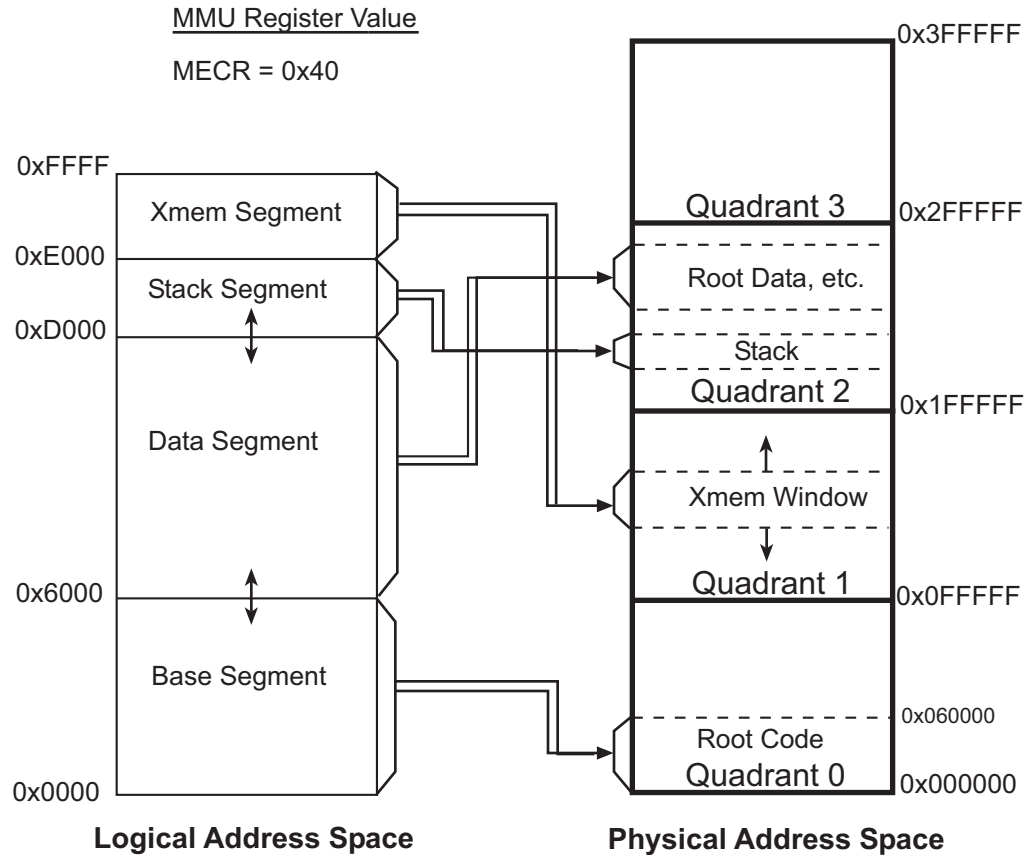
A basic Rabbit system typically contains two or three static memory devices: one flash memory device and one or two RAM devices. Additional static memory devices may be added. If an application requires storing a lot of data in flash memory, it is recommended that a mass storage flash device be added such as NAND or serial flash. Dynamic C contains drivers for both NAND and serial mass storage devices. Alternatively, another parallel flash memory device could be added, although these devices tend to be smaller and more expensive and are not as suitable for larger amounts of data. Note that some board designs may only contain a serial boot flash and SRAM. On these boards, the program is copied into the SRAM at boot time from the serial flash. The program is then executed from static RAM.

Trying to use a single, parallel flash memory chip to store both a program and live data that must be frequently changed can create software latency problems. When data is written to a small-sector flash memory, the memory is inoperative during the 5 to 20 ms that it takes to write a sector. If the same memory is used to hold data and the program, then the execution of code must cease during this write time. The 5-20 ms is timed out by a small routine executing from root RAM while system interrupts are disabled, effectively freezing the system for 5-20 ms. The 5-20 ms lockup period can adversely affect real-time operation.

5.2 Memory Segments

From the point of view of a Dynamic C programmer, there are a number of different uses of memory. Each memory use occupies a different segment in the logical 16-bit address space. The four segments are shown in Figure 5-1.

Figure 5-1 Memory Map of 16-bit Logical Address Space



The figure above shows that the segments of the 16-bit logical address space map to the physical address space. The extended register set and additional 32 bit registers provided by the Rabbit 4000 make it easy to access the physical memory directly, bypassing the logical to physical mapping and allowing linear access of up to 16 MB. The size of the physical address space is determined by the quadrant size.

The quadrant size is determined by the MMU Expanded Code Register (MECR). This register contains the Bank Select Address setting. The Bank Select Address represents the two most significant bits of the physical address that will be used to select among the different quadrants. By default, the MECR selects A19 and A18, thus leaving 18 bits for the address, which results in a quadrant size of 256 KB. Table 5-1 shows the possible MECR values and the resulting quadrant sizes.

Table 5-1 Selecting the Quadrant Size

MECR Value	Address Bits Used to Select Quadrant	Quadrant Size	Physical Address Space
11100000b	A18, A17	128 KB	512 KB
00000000b	A19, A18	256 KB	1 MB (default)
00100000b	A20, A19	512 KB	2 MB
01000000b	A21, A20	1 MB	4 MB
01100000b	A22, A21	2 MB	8 MB
10000000b	A23, A22	4 MB	16 MB

One advantage of retaining the Rabbit 16-bit logical memory organization is that 16-bit addresses and pointers can reduce code size and execution times.

NOTE: The relative size of the base and data segments can be adjusted by increasing or decreasing the BIOS macro `DATAORG` in increments of `0x1000`.

5.2.1 Definition of Terms

The following definitions clarify some of the terms that will be encountered in this chapter.

Extended Code (a.k.a., `xmem` code): Instructions located in the extended memory segment.

Extended Constants (a.k.a., `xmem` constants): C constants located in the extended memory segment. They are mixed together with the extended code.

Extended Memory (a.k.a., `xmem`): Logical addresses in `0xE000 - 0xFFFF` range.

Extended RAM: RAM not used for root variables or stack. Extended memory in RAM may be used for large buffers to save root RAM space. The Dynamic C compiler supports the `far` keyword to allow C data types to be declared and defined in extended memory. The code generation for the far data types makes use of the expanded Rabbit 4000 instructions and registers. The function `xalloc()` also allocates space in extended RAM memory. See the *Dynamic C User's Manual* for more information on the `far` keyword.

Far Constants: C constants declared with the “`far`” keyword currently located in the extended memory segment. The location of far constants may be changed in the future.

Root Code: Instructions located in the base segment.

Root Constants: C constants, such as quoted strings, initialized variables or data tables, that are located in the base segment. Root constants share space with root code unless separate I&D space is enabled.

Root Memory: Logical addresses below `0xE000`. Please note that root memory is not the same as the root segment. The root segment is contained in root memory, as are the data and stack segments. The root segment is also known as the base segment.

Root Variables: C variables, including structures and arrays that are not initialized to a fixed value, are located in the data segment.

5.2.2 The Base (or Root) Segment

The base segment has a typical size of 24 KB. The larger the base segment, the smaller the data segment and vice-versa. Base segment address zero is always mapped to physical address zero. Sometimes the base segment is mapped to flash memory since root code and root constants do not change except when the system is reprogrammed. It may be mapped to RAM for debugging, or to take advantage of the faster access time offered by RAM. Serial flash boot configurations always map the base segment to RAM since there is no parallel flash.

With separate I&D space disabled, the base segment holds a mixture of code and constants. C functions or assembly language programs that are compiled to the base segment are interspersed with data constants. Data constants are inserted between blocks of code. Data constants defined inside a C function are placed after the end of the code belonging to the function. Data constants defined outside of C functions are placed in memory where they are encountered in the source code.

Except in small programs, the bulk of the code in a program is executed using the extended memory (xmem) segment. Code operates at the same speed whether addressed through the base segment or the xmem segment, except that calling and returning from xmem functions takes a few extra clock cycles. It just takes a few cycles longer to call xmem functions and return from them.

5.2.2.1 Types of Code Best-Suited for the Base Segment

- **Short subroutines of about 20 instructions or less that are called frequently** will use less execution time if placed in root memory because of the faster calling linkage for 16-bit versus 20-bit addresses. For a call and return, 20 clocks are used compared to 32 clocks for xmem calls and returns. This reduction in execution time becomes more significant when the call/return sequence is a substantial portion of the total execution time.
- **Interrupt routines.** Interrupt vectors use 16-bit addressing so the entry to an interrupt routine must be in the base segment.
- **The BIOS core.** The initialization code of the BIOS must be at the start of the base segment.
- **A function that modifies the XPC** must always be executed from root memory.

5.2.3 The Data Segment

The data segment has a typical size of 28 KB, starting at 24 KB (0x6000 above root code) and ending at 52 KB (0xCFFF). The data segment is mapped to RAM and contains C variables. Data allocation starts at or near the top and proceeds in a downward direction. It is also possible to place executable code in the data segment if it is copied from flash to the data segment. This can be desirable for code that is self-modifying, code to implement debugging aids or code that controls writes to the flash memory.

In separate I&D space, the data segment is twice as big (~54 KB), but code cannot be executed from it.

5.2.4 The Stack Segment

Usually the stack segment is assigned to the range of logical addresses 0xD000 to 0xDFFF. It is always mapped to RAM and holds the system stack. Multiple stacks may be implemented by defining them in the 4 KB space, by remapping the 4 KB space to different locations in physical RAM memory, or by using both approaches. Multiple stack allocation is handled by μ C/OS-II internally. For example, if sixteen 1 KB stacks are needed then four stacks can be placed in each 4 KB mapping and four different mappings for the window can be used.

5.2.5 The Extended Memory Segment

This 8 KB segment from logical address 0xE000 to 0xFFFF is a sliding window into extended code and it can also be used by routines that manipulate data located in extended memory. The xmem window uses up only 8 KB of the 16-bit addressing space. While executing code the mapping is shifted by 4 KB each time the code passes the halfway point in the 8 KB xmem window. The halfway point corresponds to the root address 0xF000, or 60KB. On all Rabbit processors, up to 1 MB of code can be efficiently executed by moving the mapping of the 8 KB window using special instructions that are designed for this purpose: long call (LCALL), long jump (LJP) and long return (LRET). Dynamic C currently supports up to 1MB of code using these instructions. The Rabbit 4000 processor allows up to 16 MB of code using new extended versions of these instructions: long long call (LLCALL), long long jump (LLJP), and long long return (LLRET).

The xmem segment is a window into the physical address space. Using the appropriate segment register (XPC or LXPC) any logical address in the range 0xE000 to 0xFFFF can be mapped to any address in the physical address space. Consider the following examples:

Table 5-2 Mapping Xmem Addresses

Segment Register	Logical Address	Mapping Equation	Physical Address
XPC = 0xFE	0xE74F	$0xFE000 + 0xE74F = 0x10C74F$	0x10C74F
LXPC = 0x0FE	0xE74F	$0x0FE000 + 0xE74F = 0x10C74F$	0x10C74F
XPC = 0xF2	0xE000	$0xF2000 + 0xE000 = 0x100000$	0x100000
LXPC = 0xFF0	0xFFFF	$0xFF0000 + 0xFFFF = 0xFFFFFFF$	0xFFFFFFF

WARNING: The XPC is used for addressing up to 1 MB, and the LXPC is used for addressing up to 16 MB. Mixing the use of the XPC and LXPC is dangerous.

Please see Technical Note 202, “Rabbit Memory Management in a Nutshell,” for more details on how memory mapping works on the Rabbit 2000 and Rabbit 3000. This document is available at: rabbit.com.

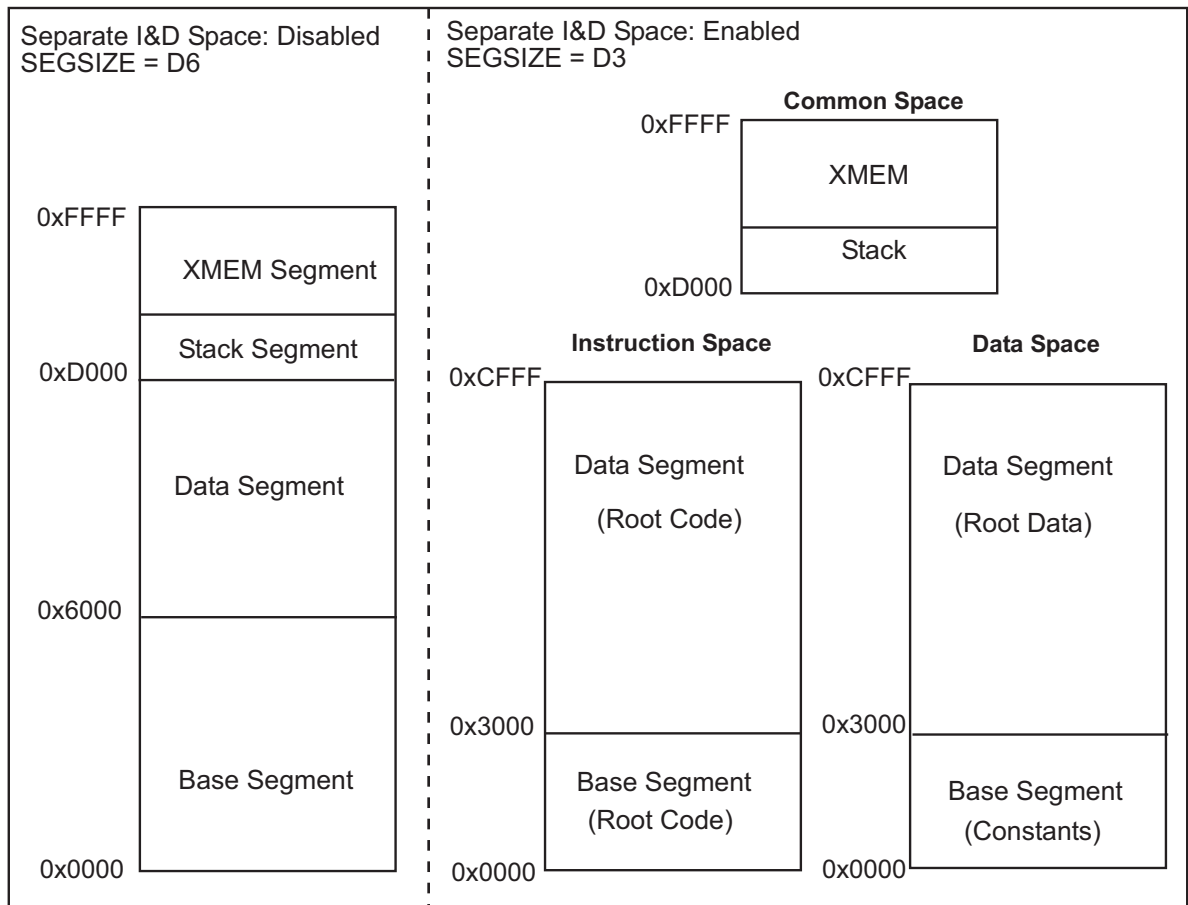
5.3 Separate I&D Space

Separate instruction and data space is a hardware memory management scheme that uses address line inversion to double the amount of logical address space in the base and data segments. In other words, this doubles the amount of root code and root data available for an application program.

Without separate I&D space, recall that in a typical memory map of the 16-bit address space, the base segment holds a mixture of code and constants and is mapped to flash; the data segment holds C variables and is mapped to RAM. With separate I&D space, code and data no longer have to divide this space because they share logical addresses by inverting address lines depending on whether the CPU is fetching instructions or data.

The drawing in [Figure 5-2](#) shows the logical address space when separate I&D space is both enabled and disabled. Typical SEGSIZE values are shown. The boundary at 0x3000 (and 0x6000) is determined by the macro `ROOT_SIZE_4K` in the BIOS. The value of this macro is the number of 4 kilobyte pages used for the base segment. The boundary may be changed, but care must be taken. To change the boundary, define `ROOT_SIZE_4K` to the desired number of 4K pages on the “Defines” tab in Options | Project Options.

Figure 5-2 16-Bit Logical Address Space



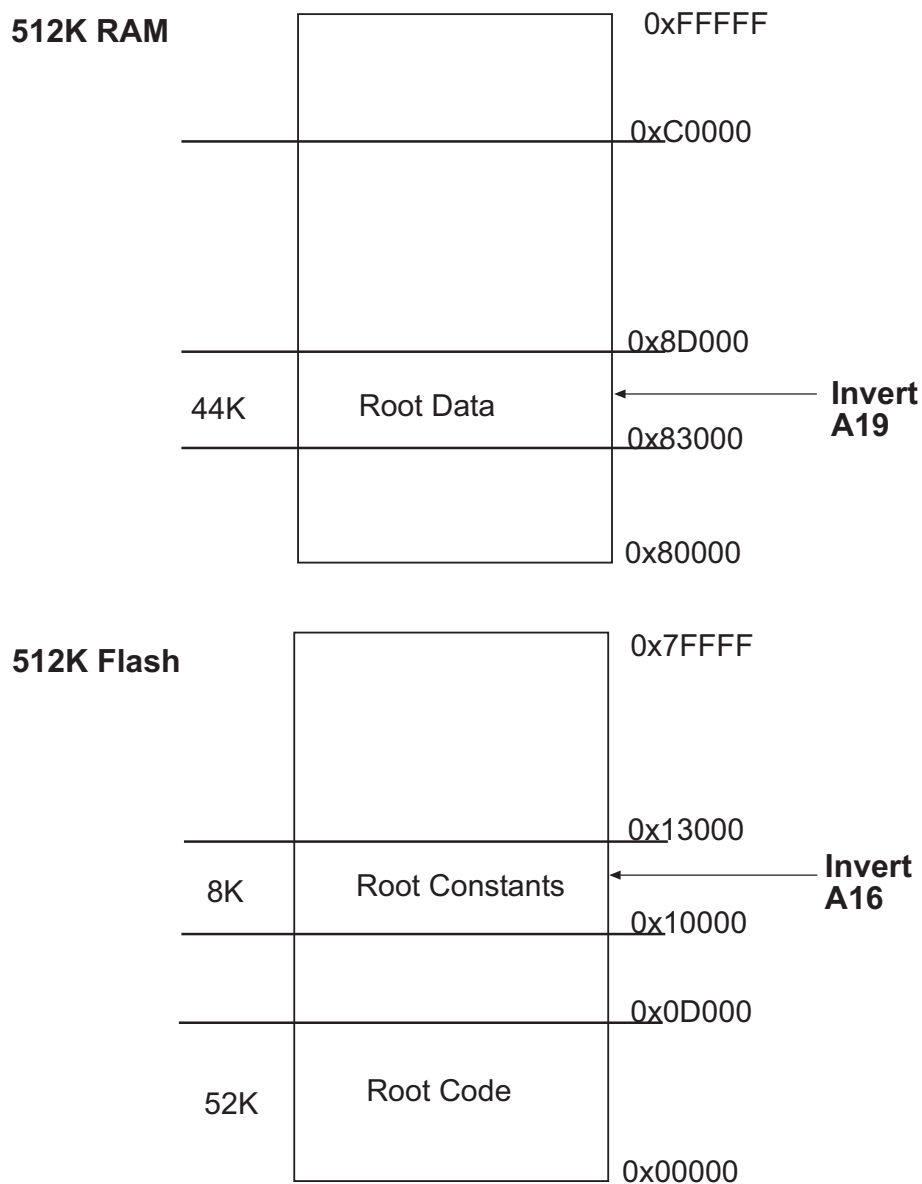
NOTE: This diagram illustrates how separate I&D space works; the actual values used in the BIOS may differ from those shown here.

Separate I&D logical addresses map to physical addresses by inverting address lines A16, the most significant address bit or both. The most significant address bit may be A18-A23, depending on the MECR setting. The MMU Instruction/Data Register (MMIDR) determines which lines are inverted. Please see the *Rabbit 4000 Microprocessor User's Manual* for more information about the MMIDR.

The following diagram (Figure 5-3) shows the physical address space when separate I&D space is enabled, SEGSIZE = 0xD3 and code is compiled to flash.

The inversion of A16 causes the root constants in the data space to be addressed in the second 64 KB block of the flash. The inversion of MSB (A19 in this example) causes the root data in the data space to be located in RAM (RAM is mapped at 0x80000), starting at 0x83000 as directed by the lower nibble of SEGSIZE.

Figure 5-3 Physical Address Space when Separate I&D Space is Enabled and the Quadrant Size is 256 KB



When using separate I&D space you can not reference code as data or data as code in logical memory below the stack. When using separate I&D space, the processor makes a distinction between fetching an instruction from memory and fetching data from memory. The RAM segment register determines the window in RAM where root code may be executed.

Embedded applications that do not need more code or data space do not require any changes for separate I&D space. By default, Dynamic C compiles without separate I&D space enabled.

5.3.1 Enable Separate I&D Space

To use separate I&D space, check the enable separate I&D space option on the Compiler tab of the Options | Project Options dialog. The Dynamic C command line compiler equivalent is `-id+` (enable I&D space) and `-id-` (disable I&D space). Please see the *Dynamic C User's Manual* for more information about the command line compiler.

The BIOS and the compiler handle the memory mappings so the user does not need to know the details. However, if you want to change the way an interrupt vector is handled or you need to write a flash driver, the rest of this chapter provides you with the necessary information.

5.3.2 Separate I&D Space Mappings in Dynamic C

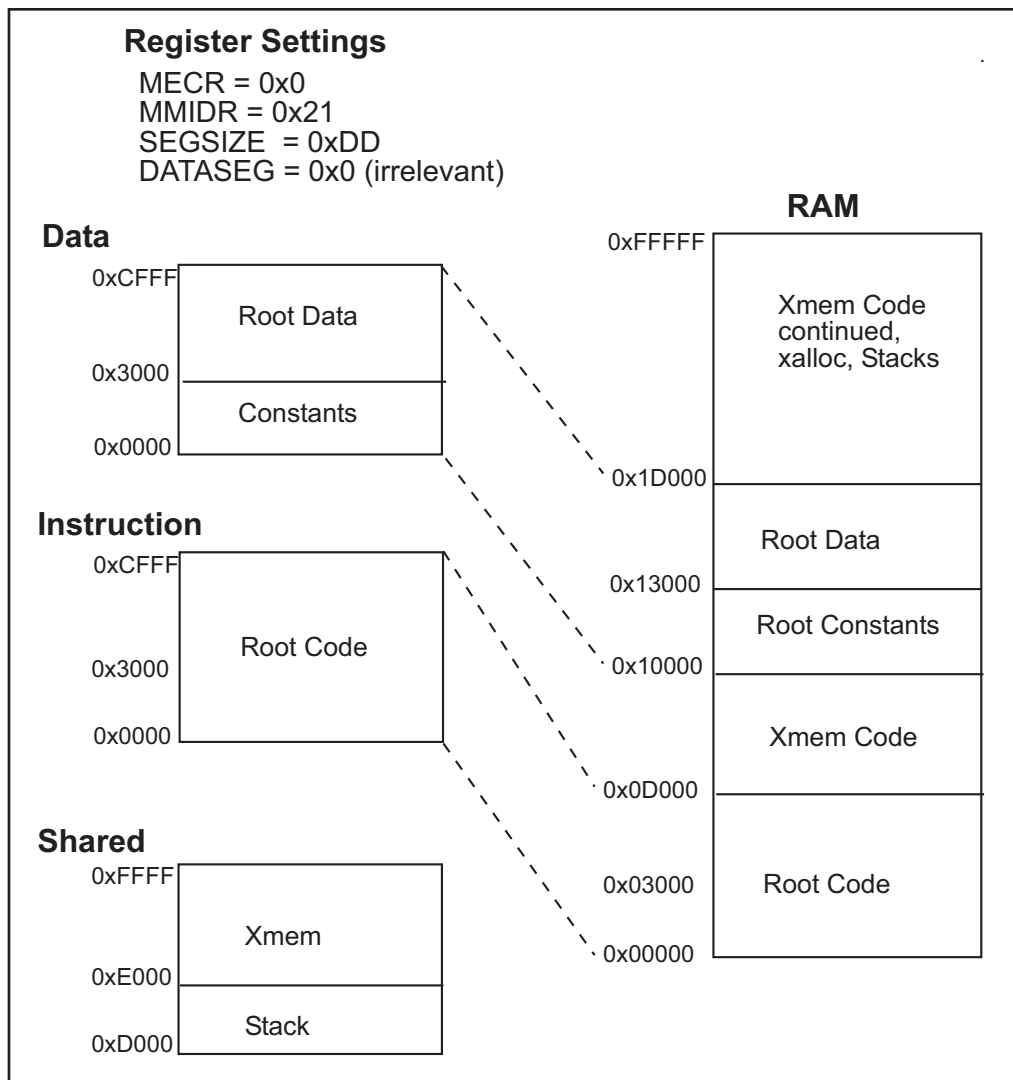
The next two subsections show the default MMU settings that Dynamic C uses when separate I&D space is enabled.

5.3.2.1 Compiling to RAM

For RAM compiles, all banks (quadrants) are mapped to RAM. In a 20-bit physical address space (i.e., 1 MB physical address space), a 512 KB memory would be mapped with the lower 256 KB mapped to banks 0 and 2. The higher 256 KB are mapped to banks 1 and 3. In this configuration, A16 is inverted to provide access to the constants and data starting at the 64K boundary. The standard configuration is to set the SEGSIZE register to 0xDD so that the base segment occupies the entire 52 KB region up to the stack segment. Note that this configuration causes the DATASEG register to be irrelevant.

The BIOS sets the MMIDR to 0x21. Bit 5 of this register enables the instruction/data split and bit 0 causes the inversion of A16.

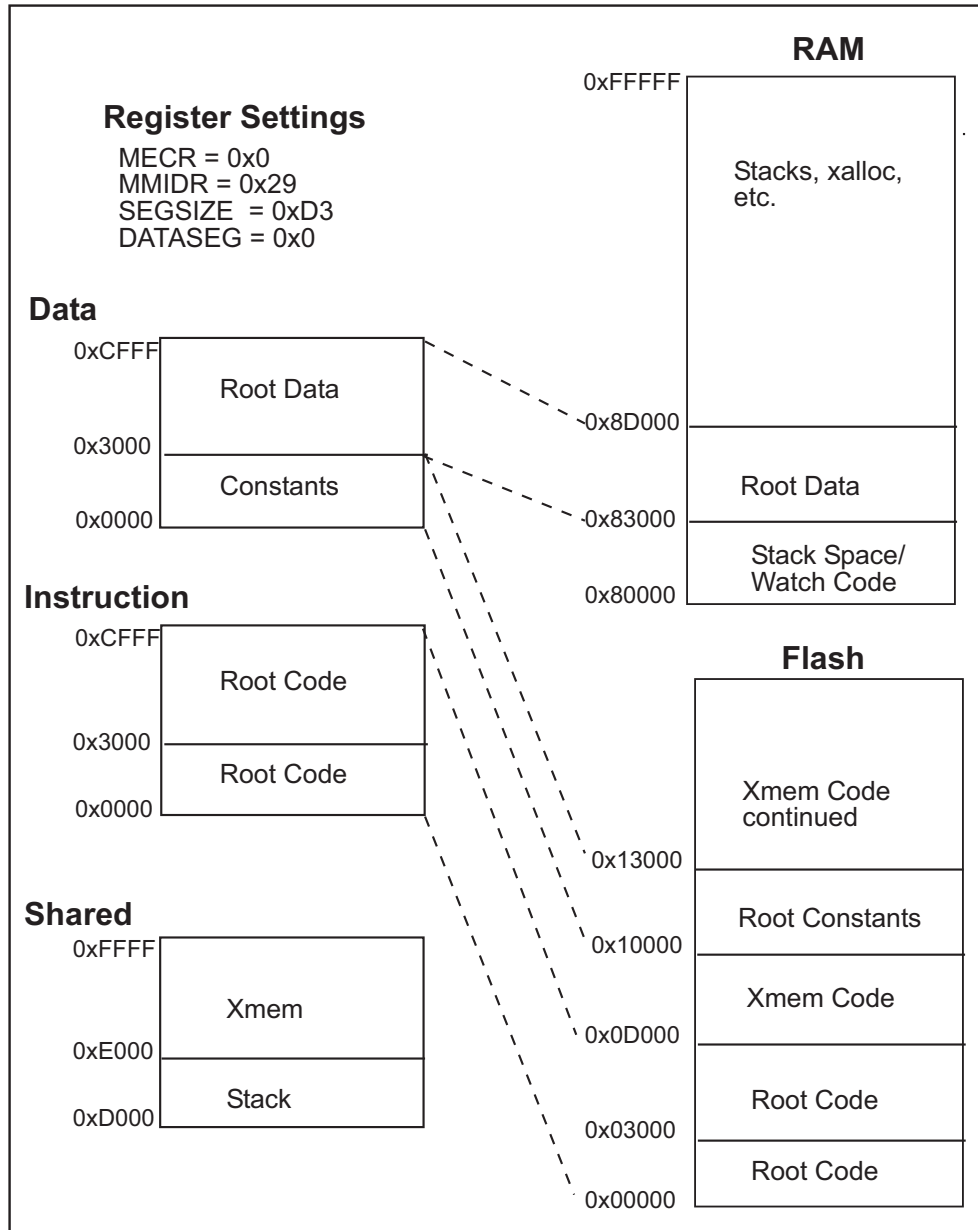
Figure 5-4 RAM Compile Memory Mapping



5.3.2.2 Compiling to Flash

For flash compiles, flash is mapped to banks 0 and 1. The address range depends on the size of the physical address space. For example, a 20-bit address space with 512 KB of flash would mean that flash is mapped from 0x00000 to 0x7FFFF. Alternatively, a 22-bit address space (1 MB quadrants) with 1 MB of flash would mean that the flash is mapped to 0x000000 to 0x0FFFFFFF in bank 0 and is repeated again in bank 1 from 0x100000 to 0x1FFFFFFF. RAM is mapped to banks 2 and 3 (address range 0x80000 to 0xFFFFF for 20 bit, and 0x200000 to 0x3FFFFFFF for 24 bit, respectively).

Figure 5-5 Flash Compile Memory Mapping



The BIOS sets the MMIDR to 0x29 to enable the I&D space for flash compilation. Bit 5 of this register enables the I&D split, bit 0 enables inversion of A16 for the data space base segment (i.e., the logical

address space for constants) and bit 3 enables inversion of MSB for the data space data segment (i.e., the logical address space for root data).

5.3.3 Customizing Interrupts

No special code is required to customize interrupts using separate I&D space on the Rabbit 4000 with the addition of the RAM segment register (RAMSR). Use `SetVectIntern()` and `SetVectExtern()` to set interrupts. Please see the *Dynamic C Function Reference Manual* for more information on these functions.

5.4 How The Compiler Compiles to Memory

The compiler generates code for root code, root constants, extended code, extended constants, and far constants. It allocates space for data variables, but, except for constants, does not generate data to be stored in memory. Any initialization of RAM variables must be accomplished by code since the compiler is not present when the program starts in the field. (Please see `#GLOBAL_INIT` in the *Dynamic C User's Manual*.)

Static variables are not zeroed out by default. The BIOS macro `ZERO_OUT_STATIC_DATA` may be set to "1" which will only zero out static variables on board power-up or reset. Zeroing out static variables is not compatible with the use of "protected" variables because they will be zeroed out along with the rest of the static data.

5.4.1 Placement of Code in Memory

Code may be placed in either extended memory or root memory. Functions execute at the same speed, but calls to functions in root memory are slightly more efficient than calls to functions in extended memory.

In all but the smallest programs, most of the code is compiled to extended memory. Root constants share the memory space needed for root code (when separate I&D space is disabled), so as the memory needed for root constants increases, the amount of code that can be stored in root memory decreases and code must be moved to extended memory.

Please see the *Dynamic C User's Manual* regarding the compiler directive `#memmap` for more information about controlling the placement of code in memory.

5.4.2 Paged Access in Extended Memory

The code in extended memory executes in the 8 KB window from `0xE000` to `0xFFFF`. This 8 KB 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 64 KB logical space. Special instructions, particularly `LCALL`, `LJP`, and `LRET`, are used to access code outside of the 8 KB window for addresses below `0x100000`. Similarly, `LLCALL`, `LLJP`, and `LLRET` can be used to access code outside of the 8KB window to any place in the physical address space. When one of these transfer-of-control instructions is executed, both the address and the view through the 8 KB window change, allowing transfer to any instruction in the physical memory space. The 12-bit `LXPC` register controls which of two consecutive 4 KB pages the 8 KB window aligns with (there are 256 pages in a 1 MB physical address space). The 16-bit `PC` controls the address of the instruction, usually in the region `0xE000` to `0xFFFF`. The advantage of paged access is that most instructions continue to use 16-bit addressing. Only when a page change is needed does a physical address transfer of control need to be made.

As the compiler compiles code for the extended code window, it checks to see if the code has passed the midpoint of the window or 0xF000. When the code passes 0xF000, the compiler generates code to slide the window down by 4 KB so that the code at F000+x becomes resident at 0xE000+x. This automatic paging results in the code being divided into segments that are typically 4 KB long, but which can be very short or as long as 8 KB. Transfer of control within each segment can be accomplished by 16-bit addressing. Between segments, physical addressing (19- to 24-bit depending on configuration) is required. Assembly blocks are limited to 4 KB because the compiler cannot generate automatic paging code in assembly.

5.5 Memory Planning

Design conventions for memory configuration of a Rabbit 4000-based system specify flash and SRAM.

Table 5-3 Typical Interface Between the Rabbit 4000 and Memory

Primary Flash	SRAM	Secondary Flash
/CS0, /OE0 and /WE0	/CS1, /OE1 and /WE1	/CS2, /OE0 and /WE0

5.5.1 Flash

Code is typically stored in flash memory, so the size of code must be anticipated. Usually code size up to 1 MB is handled by one or two flash memory chips. If you are writing a program from scratch, remember that 1 MB of code is equivalent to 50,000 to 100,000 C statements, and such a large program can take years to write. If you are using Dynamic C libraries, it is fairly easy to have this much code in your application.

Constant data tables can be conveniently placed in extended memory using the `xdata` and `xstring` declarations supported by Dynamic C, so the amount of space needed for constant data can be added to the amount of space needed for code. The `far` keyword can also be used to create constants in `xmem` using standard C variables.

5.5.2 Static RAM

C programs vary in how much RAM will be required and having more RAM is necessary for debugging. Since debugging and program testing generally operate more powerfully and faster when sufficient RAM is available to hold the program and data, most controllers based on the Rabbit 4000 use a dual footprint for RAM that can accommodate 128K x 8 or 512K x 8, which are both in 32-pin packages. The base RAM is interfaced to /CS1 and /WE1, and /OE1.

RAM is required for the following items:

- **Root Variables** - maximum of 40-44 KB, and about 4 KB more if separate I&D space is enabled.
- **Stack Pages** - stack is usually 4 KB, rarely more than 20 KB.
- **Debugging** - as a convenience on prototype units, 1 MB is usually enough to accommodate programs. It is not necessary to debug in RAM, but may be desirable.
- **Extended Memory (a.k.a., `xmem`)** - can be used for code and data, such as communications applications or data logging applications. The amount needed depends on the application.

6. The Rabbit BIOS

When Dynamic C compiles a user's program to a target board, the BIOS (Basic Input/Output System) is compiled first as an integral part of the user's program. The BIOS comprises files that contain the code needed by the user program to interface with Dynamic C and the Rabbit hardware. The BIOS may also contain a software interface to the user's particular hardware. Certain drivers in the Dynamic C library suite require BIOS routines to perform tasks that are hardware-dependent.

The BIOS also:

- Takes care of microprocessor system initialization, such as the setup of memory.
- Provides the communications services required by Dynamic C for downloading code and performing debugging services such as setting breakpoints or examining data variables.
- Provides flash drivers.

The file `RabbitBIOS.c` is a wrapper that permits a choice of which BIOS to compile. A more modular design has been implemented by moving many of the configuration macros to separate configuration libraries. The main BIOS file (`Stdbios.c`) and the multiple configuration libraries are located in `LIB\Rabbit4000\BIOSLIB`.

Dynamic C 10.21 introduces a change in the BIOS files: Origin declarations have been redesigned. One of the most dramatic results of the redesign is the ability to define relative relationships between origins during the setup of memory. This eliminates many of the macro definitions that were necessary before.

The supplied BIOS allows Dynamic C to boot up on any Rabbit-based system that follows the basic design rules needed to support Dynamic C. The BIOS requires either a 128 KB RAM or both a flash device and a 32 KB or larger RAM for it to be possible to compile and run Dynamic C programs. If the user uses a flash memory from the list of flash memories that are already supported by the BIOS, the task will be simplified. A list of supported flash devices is listed in Technical Note 226, available online at:

rabbit.com/docs/app_tech_notes.shtml

If the flash device is not already supported, the user will have to write a driver to perform the write operation on the flash memory. This is not difficult provided that a system with 128 KB of RAM and the flash memory to be used are available for testing.

An existing BIOS can be used as a skeleton to create a new BIOS. Frequently it will only be necessary to change `#define` statements at the beginning of the file. In this case it is unnecessary for the designer to understand or work out the details of the memory setup and other processor initialization tasks. Refer to the *Dynamic C User's Manual* for details on creating a user-defined BIOS.

6.1 Startup Conditions Set by the BIOS

The BIOS performs initialization tasks and #use's library files that contain setup information.

6.1.1 Registers Initialized in the BIOS

The BIOS sets up initial values for the following registers by means of code and declarations.

MBxCR

There are four memory bank control registers: MB0CR, MB1CR, MB2CR, and MB3CR. They are 8-bit registers, each one associated with a quadrant of the physical memory space. A memory bank control register determines which memory chip is mapped into its quadrant, how many wait states will be used for accessing that memory chip, and whether the memory chip will be write protected.

MECR

8-bit register that determines the quadrant size and thus the size of the physical address space.

STACKSEG(H/L)

16-bit register that determines the location of the stack segment in the physical memory space.

DATASEG(H/L)

16-bit register that determines the location of the data segment in the physical memory space, normally the location of the data variable space.

SEGSIZE

8-bit register holding two 4-bit values. Together the values determine the relative sizes of the base segment, data segment and stack segment in the 64 KB logical memory space.

MMIDR

8-bit register used to control separate I&D space and to force /CS1 to be always enabled or not. Having /CS1 always enabled reduces access time if /CS1 is routed through an external battery backup device and the propagation delay through the external device may slow the transition of /CS1 during memory cycles.

SP

The SP register is the system stack pointer. It is frequently changed by the user's code. The BIOS sets up an initial value.

6.1.2 Origins

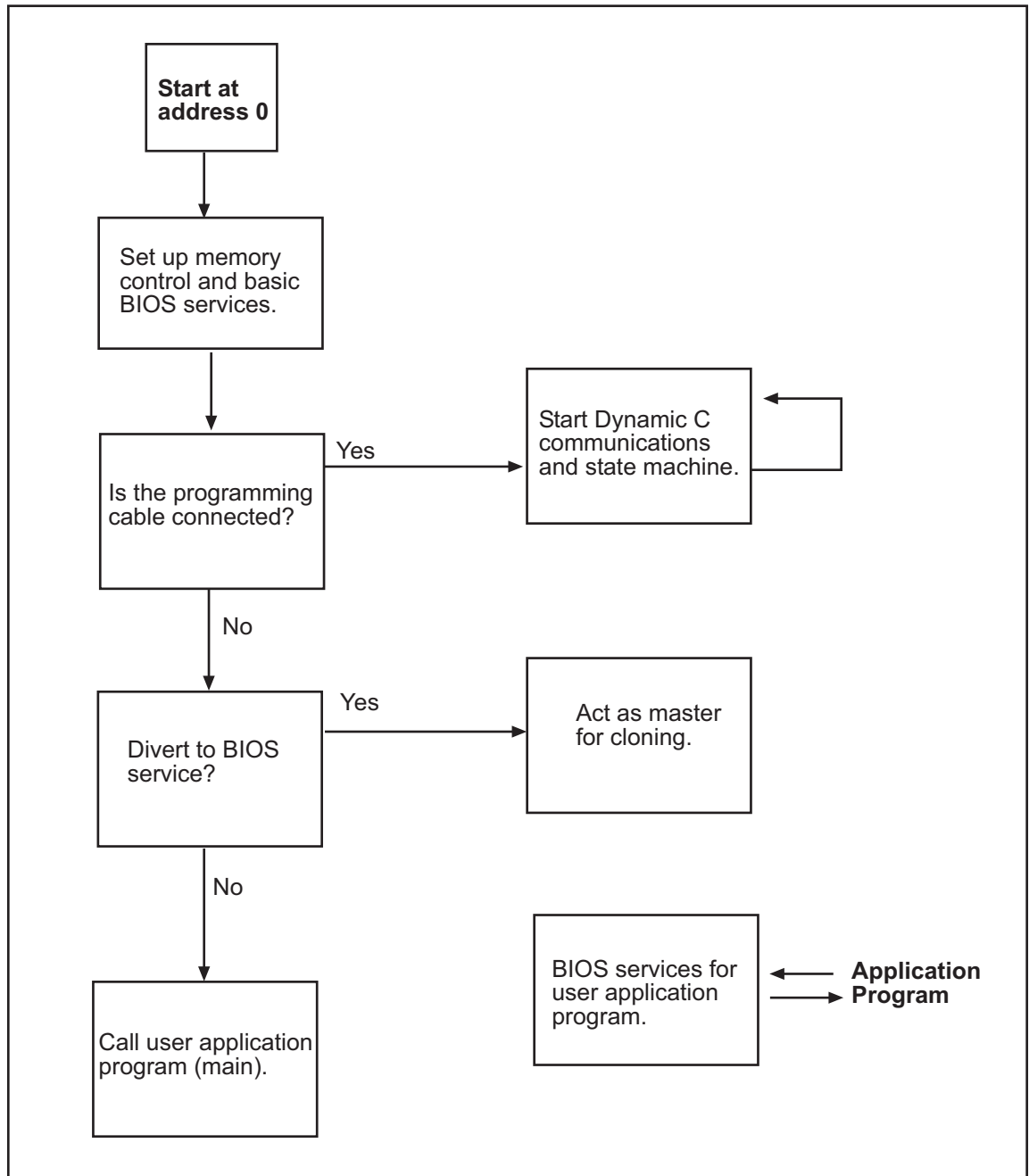
Dynamic C uses a mechanism known as an "origin" to define regions of memory for different purposes. The BIOS declares several origins to tell the Dynamic C compiler where to place different types of code and data. Starting with Dynamic C 10.21 the origin directives are in the library file `memory_layout.lib`, which is #use'd in the BIOS.

For more information about the MMU and MIU registers please see the *Rabbit 4000 Microprocessor User's Manual*.

6.2 BIOS Flowchart

The following flowchart summarizes the functionality of the BIOS:

Figure 6-1 BIOS Flowchart



NOTE: To use the diagnostic port on the RCM43xx, you must first reset the board and then plug in the “Diag” header of the programming cable.

NOTE: If the programming cable is connected at power-up, the Rabbit will never execute the BIOS, since the cable holds the board in coldboot mode.

6.3 Internally-Defined Macros

Some macros used in the BIOS are defined internally by Dynamic C before the BIOS is compiled. They are defined using tests done in the bootstrap loading, or by reading variables set in the GUI or set by the CLC (command line compiler).

Table 6-1 Partial List of Compiler-Defined Macros

Macro Name	Macro Description
<code>__BOARD_TYPE__</code>	This is read from the System ID block or defaulted to 0x100 (the BL1810 JackRabbit board) if no System ID block is present. This can be used for conditional compilation based on board type.
<code>CC_VER</code>	Gives the Dynamic C version in hex, i.e., version 10.21 is 0x0A21.
<code>__CPU_ID__</code>	This macro identifies the CPU type, e.g., R4000 is the Rabbit 4000 microprocessor.
<code>__FLASH__, __RAM__</code>	Used for conditional compilation of the BIOS to distinguish between compiling to RAM and compiling to flash. These are set in the Compiler tab in the Options Project Options dialog.
<code>__RAM_SIZE__, __FLASH_SIZE__</code>	Used to set the MMU registers and code and data sizes available to the compiler. The values of these macros represent the number of 0x1000 blocks of memory available.
<code>__SEPARATE_INST_DATA__</code>	Flag for identifying whether separate I&D space is enabled or disabled.
<code>FLASH_COMPILE, RAM_COMPILE, FAST_RAM_COMPILE</code>	Used to determine compile mode in code.

See the *Dynamic C User's Manual* for other internally-defined macros.

6.4 Modifying the BIOS

The BIOS that is supplied with Dynamic C may be modified or replaced. Prudence demands that any changes made to this important piece of software be done one step at a time in order to more easily detect and isolate any problems that may arise.

`RabbitBios.c` is still used, but is more of a wrapper file that brings in some configuration and definition files, checks a few error conditions and then, before starting compilation of the application, selects which BIOS file to activate. The default BIOS is `\Lib\Rabbit4000\BIOSLIB\StdBios.c`.

6.4.1 Macros that Affect the BIOS

There are several macros that may be modified for custom-designed boards or for special situations involving off-the-shelf Rabbit 4000-based boards. The following list is not exhaustive.

CLOCK_DOUBLED

Default value of 1 causes the clock speed to be doubled. Setting this to zero means the clock speed will not be doubled.

To override the default, define `CLOCK_DOUBLED` to zero in the project by using the Defines tab of the Project Options dialog.

CS1_ALWAYS_ON

Default value of 0 disables the feature of keeping /CS1 always active.

To override the default, define `CS1_ALWAYS_ON` to 1 in the project by using the Defines tab of the Project Options dialog. Keeping /CS1 always active is useful if your system is pushing the limits of RAM access time. It will increase power consumption a little.

DATAORG

This macro is deprecated. Use `ROOT_SIZE_4K` instead.

ROOT_SIZE_4K

This macro defines the number of 4 kilobyte pages used for the base segment. The default is 3 when separate I&D space is enabled, and 6 otherwise.

To override the default, define `ROOT_SIZE_4K` in the project by using the Defines tab of the Project Options dialog. Increasing this value increases the size of root constants when separate I&D space is enabled, and root code when it is disabled. The sum of available root and data is constant, such that increasing one decreases the other. This macro can be changed to as high as 11 and as low as 1 when separate I&D space is enabled or as low as 3 when separate I&D space is disabled.

ENABLE_CLONING

Default value of 0 disables cloning.

To override the default, define `ENABLE_CLONING` to 1 in the project by using the Defines tab of the Project Options dialog. This slightly increases the code size of the BIOS.

If cloning is used, PB1 should be pulled up with 50K or so pull-up resistor. On some Rabbit core modules, such as the RCM4200, the PB1 (CLKA) signal is either not available or not pulled up on the programming port. The master can be forced to invoke cloning support by setting `CL_FORCE_MASTER_MODE` to 1. This will cause the BIOS to assume a cloning cable is attached on every startup, assuring that only the cloning code will run. Note that defining `CL_FORCE_MASTER_MODE` to 1 will not allow the program on the board to run, that is, the board will act only as a clone master.

While compiling to the target with `CL_FORCE_MASTER_MODE` set to 1, the loss of target communication is expected and unavoidable. After the program has loaded and target communication is lost the clone master will still correctly perform its cloning function after a cloning cable is attached.

Various cloning options are available when `ENABLE_CLONING` is set to one. For more information on cloning, please see Chapter 8, “BIOS Support for Program Cloning,” in this manual and/or Technical Note 207, “Rabbit Cloning Board,” available at rabbit.com.

FLASH_SIZE

Sets the amount of flash available. The default value is the internally defined `_FLASH_SIZE_`. The units are the number of 4 KB pages of flash. In special situations, such as splitting flash between two coresident programs, this may be modified to a smaller value than the actual available flash.

RAM_SIZE

Sets the amount of RAM available. The default value is the internally defined `_RAM_SIZE_`. The units are the number of 4 KB pages of RAM. In special situations, such as splitting RAM between two coresident programs, this may be modified to a smaller value than the actual available RAM.

USE_TIMER_A_PRESCALE

Uncomment this macro in `Lib/Rabbit4000/BIOSLIB/sysconfig.c` to run the peripheral clock at the same frequency as the CPU clock instead of the standard “CPU clock/2.” This allows higher baud rates if Timer A is used as the baud rate generator. `USE_TIMER_A_PRESCALE` affects the resolution of the PWM, Input Capture and Quadrature Decoder systems.

WATCHCODESIZE

This macro defines the size in bytes of the region used for interrupt vectors, debug kernel special variables, and watch expressions. This macro must only be set to 0x800 or 0x1000 if the debug kernel is enabled, and can be set to 0x400 otherwise.

To override the default, change its value in `Lib/Rabbit4000/BIOSLIB/StdBios.c`.

6.4.2 Advanced Options

The following macros are defined in `STDBIOS.c`. See the top of the BIOS source code and/or the various configuration libraries for more options.

ENABLE_SPREADER

Default value is 1, which enables the clock spectrum spreader in normal mode to reduce EMI.

To override the default, define `ENABLE_SPREADER` in the project by using the Defines tab of the Project Options dialog. Define the macro to 0 to disable spectrum spreading and to 2 for strong spreading.

NUM_RAM_WAITST, NUM_RAM2_WAITST, NUM_FLASH_WAITST

These macros are defined in `boardtypes.lib`. They define the number of wait states to be used for read access to RAM and flash. Write access requires one more wait state than read access. These macros are used to determine the relevant bit values in the memory bank control registers.

The only valid values for these wait state macros are 4, 2, 1 and 0.

MBxCR_INVRT_A18, MBxCR_INVRT_A19

These macros determine whether the MIU registers for each quadrant are set up to invert address lines A18 and A19 after the logical to physical address conversion. This allows each quadrant of physical memory access up to four 256 KB pages on the actual memory device. These would be used for special compilations of programs to be coresident on flashes between 512 KB and 1 MB in size. For more information, please see Technical Note 202, “Rabbit Memory Management In a Nutshell.”

6.5 Memory Mapping in Dynamic C

The Dynamic C compiler uses the information provided by origin directives to decide where to place code and data in both logical and physical memory. The term “origin” is the mechanism by which a memory region is initially described. Dynamic C version 10.21 introduces a greatly improved version of the origin directives. The newer version of origin directives is described in [Section 6.5.1](#) and the older version is described in [Section 6.5.2](#).

Origin directives allow the programmer to tell the compiler where devices should be mapped in the Rabbit processor memory space. The origins are further used to describe what each device is and what properties these devices may have. Although origins are normally defined in the BIOS or one of its configuration libraries, they may also be useful in an application program for certain tasks, such as compiling a pilot BIOS or a cold loader, or special situations where a user wants two applications coresident within a single 256K quadrant of flash. See Technical Note 218, “Implementing a Serial Download Manager for a 256K Byte Flash,” for more information on the later. This document is available at: rabbit.com/docs/app_tech_notes.shtml.

6.5.1 Origins Starting with Dynamic C 10.21

The origin directives are all collected in a single library, `memory_layout.lib`, which is #use'd in the BIOS. The origins are arranged as a hierarchy of child and parent “origins” (memory regions) with a common parent called the “root” origin, which is essentially an abstract representation of all memory. The root origin is never explicitly defined, but any origin declaration not having a named parent origin is a child of the root origin.

The hierarchical arrangement provides a mechanism by which child origins recursively inherit the properties of their parent origins. This allows for better error checking of the memory mapping itself, since the compiler can check that a data origin is not defined in a region mapped as flash memory, for example. The example in [Section 6.5.1.1](#) illustrates this functionality.

In addition to the inheritable properties, the origin directives use relative memory mapping. Relative memory mapping allows for more flexible descriptions of memory configurations, since the syntax allows rules to be enforced on the placement of particular regions of memory with respect to one another without having to account for the actual boundaries of those regions.

6.5.1.1 Example of Origin Declarations

The code from `memory_layout.lib` provides a practical application of origins. In this section, several of the origin declarations in the memory layout library are explained. (To simplify matters, the conditional compilation macros regarding board type, compile mode and separate I&D settings in `memory_layout.lib` are not shown in this example.) A graphical representation of the regions defined by the origin declarations follows their explanation (see [Figure 6-2](#)).

The origin declaration syntax and semantics used in the following example are explicitly defined in [Section 6.5.1.2](#) and [Section 6.5.1.3](#), respectively.

The origin declarations shown below were taken from `memory_layout.lib` in Dynamic C 10.21. This library may change in future releases of Dynamic C.

```
// Macros to help declare origins
#define ORG_FLASH_SIZE (_FLASH_SIZE_*0x1000UL)
#define ORG_RAM_SIZE (_RAM_SIZE_*0x1000UL)
```

The macros `_FLASH_SIZE_` and `_RAM_SIZE_` are the number of 0x1000 (4 KB) blocks of memory available for Flash and RAM, respectively.

```
// In flash compile mode, the flash is always mapped at address 0x0
#define ORG_FLASH_START (0x0)
#define ORG_RAM_START (RAM_START*0x1000UL)
```

RAM_START is currently defined in the main BIOS file, `StdBios.lib`.

```
#orgdef flashorg flash above phy ORG_FLASH_START size ORG_FLASH_SIZE
```

The above line defines the flash device mapping. All origin definitions start with the compiler directive “`#orgdef`”. Note that the origin type is “`flashorg`”, and the origin has the user-defined name “`flash`”. The `flashorg` origin type has the property of being non-volatile. The syntax “`above phy ORG_FLASH_START`” indicates that the origin is a child of the root origin starting at the physical address defined by the macro `ORG_FLASH_START`. The final piece “`size FLASH_SIZE`” defines the size of the flash region (this is often the size of the device for `flashorg` and `ramorg` origin types).

```
#orgdef resvorg user_block in flash below end size MAX_USERBLOCK_SIZE
```

The above line defines the user block space as an origin called “`user_block`”. The type “`resvorg`” indicates that the region is a reserved origin that should not be touched by the compiler. The syntax “`in flash`” makes the origin a child of the origin named “`flash`”, defined previously. Following the inheritance, “`below end`” indicates that the origin should be at the end of its parent (in this case, the origin called “`flash`”). The region has size `MAX_USERBLOCK_SIZE`.

```
#orgdef bbramorg ram above phy ORG_RAM_START size ORG_RAM_SIZE
```

Traditional Rabbit memory configurations have a primary SRAM device that is also battery-backed. This line defines the primary RAM device as a battery-backed RAM origin (“`bbramorg`” origin type) with the name “`ram`”. The origin is a child of the root origin, starts at the physical address `ORG_RAM_START`, and has size `ORG_RAM_SIZE`.

```
#orgdef xcodorg xmemcode in flash above start to user_block
```

Defining origins for the physical memory devices in a particular configuration is not enough for the compiler, so we need to define regions for code and data. The line above defines an origin for xmem code with “xcodorg” origin type, and called “xmemcode”. The compiler knows to use this origin for xmem code because of the origin type. Note that xmemcode is a child of the origin “flash”, so it not only has the property of being a code region, but it is also non-volatile since it inherits that property from its parent “flash”. The final bit of syntax, “above start to user_block” means that the origin region starts at the beginning of its parent and extends to the beginning of the sibling “user_block” origin region. The “to” syntax allows the origin definition to remain unchanged even if the parent origin changes. A “to” terminal in an origin definition can also be followed by the syntax “end” to indicate that the region should occupy the entire parent origin region (this is useful for organizational purposes).

```
#orgdef rcodorg rootcode in xmemcode above start log 0 size  
    ROOTCODE_SIZE
```

The xmem code origin is defined above to take up the entire flash device other than the small space reserved for the user block. The reason for this is that we want to be able to put xmem code anywhere in the flash device. However, we also need root code for the segmented addressing the Rabbit provides. The line above defines the “rootcode” origin to be a child of the “xmemcode” region. This is legal because root code addresses can also be used for xmem code (all root addresses have a corresponding physical address). The origin starts at the beginning of “xmemcode”, but notice the addition of “log 0”, for “logical 0”.

The origin type for “rootcode” is rcodorg, a logical origin. This means that “rootcode” can be accessed through logical addresses, so the compiler needs to know where it is situated in logical memory, i.e., the origin needs a starting logical address. In this case, it starts at logical address “0”.

```
#orgdef rvarorg rootdata in ram above start log ROOTCODE_SIZE size  
    ROOTDATA_SIZE
```

The origin named “rootdata” is a child of “ram” and as such inherits the property of being battery-backed. The origin starts at the beginning of “ram”, and has size ROOTDATA_SIZE. It is a logical origin; its starting logical address is ROOTCODE_SIZE, a macro defined in the BIOS.

```
#orgdef wcodorg watcode in rootdata below end size WATCHCODESIZE
```

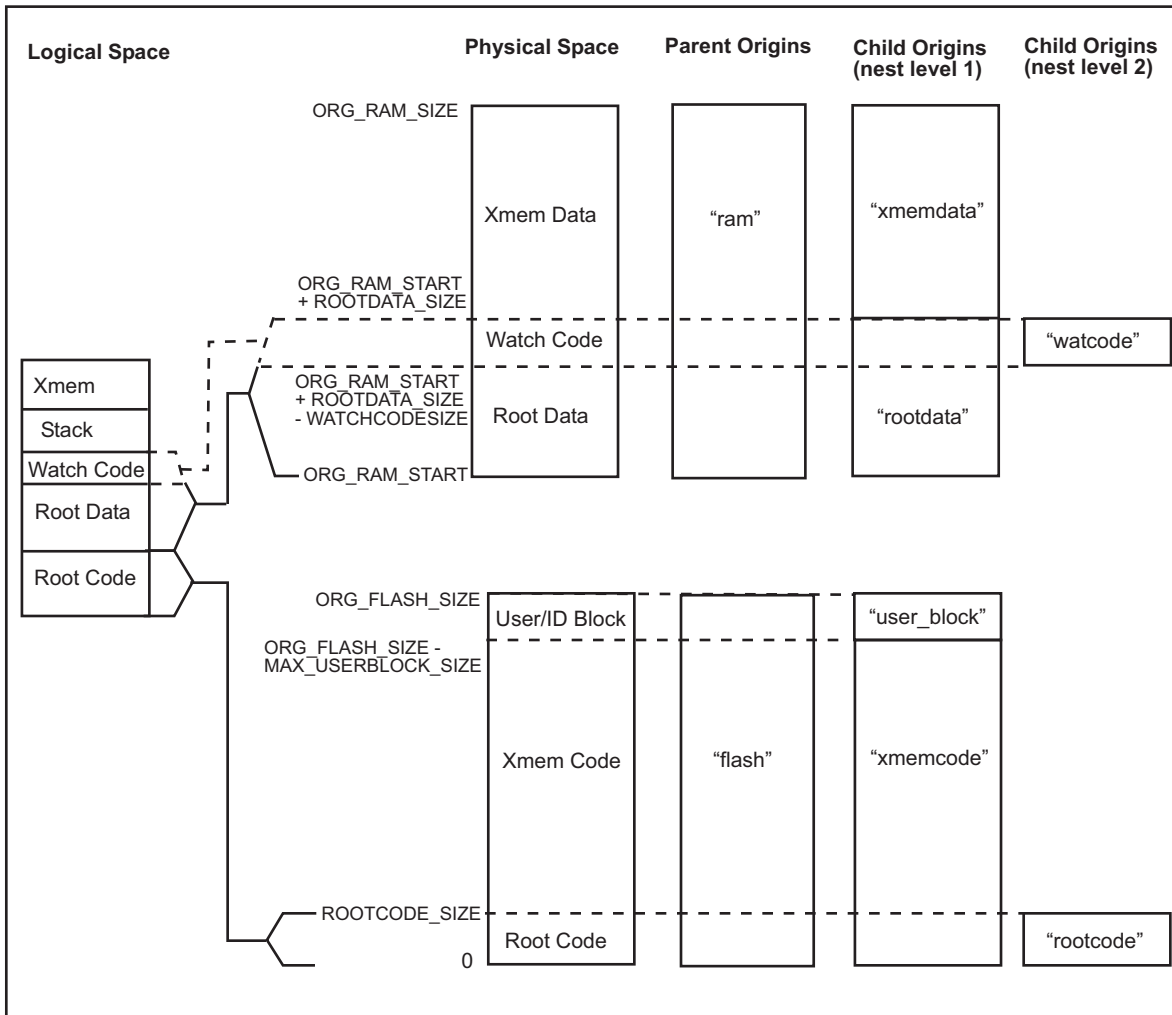
The origin named “watcode” is a child of “rootdata” and as such is also of logical origin type. Its starting logical address is not explicitly stated, but can be determined from the parent origin’s logical extents. Following the inheritance, “below end” indicates that the origin should be at the end of its parent “rootdata”, with size WATCHCODESIZE. This information is used to determine the starting logical address: We know that the starting logical address of the parent “rootdata” is ROOTCODE_SIZE. The logical addresses for child origins necessarily must be relative to their parents. In this case, that means that the starting logical address for “watcode” is then, $ROOTCODE_SIZE + ROOTDATA_SIZE - WATCHCODESIZE$.

```
#orgdef xvarorg xmemdata in ram above rootdata to userdata_buff
```

The origin named “xmemdata” is a child of “ram” and as such inherits the property of being battery-backed. The origin starts where its sibling origin “rootdata” ends; therefore, it starts at $ORG_RAM_START + ROOTDATA_SIZE$. “xmemdata” extends to the beginning of “userdata_buff”. (The declaration for the origin named “userdata_buff” is in memory_layout.lib along with the declarations for

several other regions needed for buffers. For simplicity's sake, none of them are shown here and they are not reflected in [Figure 6-2.](#))

Figure 6-2 Origins from memory_layout.lib



Each origin declaration adds a little more to the overall mapping. We recommend reading the code in `memory_layout.lib` and drawing memory maps like the one in [Figure 6-2.](#) This exercise will help to further your understanding of the complex topic of memory mapping.

6.5.1.2 Origin Declaration Syntax

Following is an EBNF (Extended Backus Naur Form) representation of the origin grammar facets: declarations, actions, and macros. Angle brackets (“<” and “>”) indicate non-terminals while terminals are represented literally with these exceptions: the “[” symbol represents a disjunction, the “:=” represents a definition, and the “[” and “]” symbols are used to enclose optional syntax.

```
<decl> ::= #orgdef <type> <name> [in <name>] <vector> <size> [locate
    <int>]

<type> ::= <phy_org> | <log_org> | wcodorg | resvorg
<phy_org> ::= flashorg | bbramorg | fastramorg | xconorg | xcodorg |
    xvarorg | xmemorg

<log_org> ::= rconorg | rcodorg | rvarorg
<vector> ::= below <offset> | above <offset>
<offset> ::= <position> [ log <int>] | <name> [ log <int> ]
<position> ::= phy <int> | start | end
<size> ::= size <int> | to <offset>

// Origin declaration start and end syntax
<start> ::= #orgstart
<end> ::= #orgend

// Origin application syntax
<orguse> ::= orgact <name> <action>
<action> ::= apply | resume

// Origin macro declaration syntax
<macdef> ::= #orgmac <define>
<define> ::= <name> = <orgval>
<orgval> ::= <name> [ <int> ] [ <boundary> ] <aspect>
<aspect> ::= <quality> <position> | size | fragments
<quality> ::= physical | logical | segment
<position> ::= start | end
```

6.5.1.3 Origin Declaration Semantics

The formal semantics of the origin declaration syntax are explained in this section.

<decl>

The non-terminal “decl” represents an origin declaration. All origin declarations begin with “#orgdef”.

<type>

The non-terminal “type” represents two subcategories of origins: physical origins and logical origins. Physical origins do not require a logical beginning and ending address because they are not accessed through logical addresses, or if they are, then through the xmem window, which is fixed. This distinction has no obvious effect on the grammar as it is written, but influences a semantic restriction discussed later.

Physical origins are represented by the phy_org non-terminal and logical origins by log_org. The origin types “wcodorg” and “resvorg” are exceptional because they may be either physical or logical.

Table 6-2 Origin Type Descriptions

Origin Type Keyword	Description
flashorg	Used for mapping flash; non-volatile.
bbramorg	Used for mapping RAM; battery-backed.
fastramorg	Used for mapping fast RAM.
xconorg	Used for mapping xmem constants.
xcodorg	Used for mapping xmem code
xvarorg	Used for mapping xmem data.
xmemorg	Reserved for future use.
rconorg	Used for mapping root constants.
rcodorg	Used for mapping root code.
rvarorg	Used for mapping root data.
wcodorg	Used for mapping watch code.
resvorg	Reserved origin, meaning that it will not be touched by the compiler.

<name>

The non-terminal “name,” though not explicitly defined in the grammar, is equivalent to a C-style identifier (not in the C namespace) and denotes the name of the origin declared.

in

The terminal “in” denotes a new concept for declaring origins – the idea of a hierarchical organization, or parentage. Given y is a previously declared origin, stating x in y denotes that x is a child of y or that y is the parent of x. The name following “in” qualifier represents the identifier of a previously declared origin. We will use the terms 'parent', 'child' and 'sibling' when referring to relationships between origins. The principle uses of this concept are the creation of boundary dependence and the enforcement of natural boundary constraints. Much of the remaining syntax becomes very natural when following the implications of this concept.

Another key concept in the child-parent model of origin declarations is the notion that child origins remove space from their parents. The reason for this is that we are still modeling a linear memory space. The hierarchy is simply a way to organize the information in a dependent manner, which obviates the macro verbosity that was previously required by origin declarations. As shown in the examples above, each origin represents an entire space of a particular origin type. Child origins transfer space from their parents to themselves. The remaining space in the parent origin is fragmented automatically by the compiler. At the end of the origin declaration section, all that remains is a flattened map that represents the true layout of memory in the physical space.

<placement>

The non-terminal “placement” denotes the placement of an origin within a larger parent space relative to the beginning of that parent (in the absence of the “in” qualifier, the parent can be considered to be all of the physical memory or “root”). The “placement” non-terminal is either the “fill” terminal or the non-terminal <vector> followed by the non-terminal “size.” The “vector” non-terminal is not a vector in the mathematical sense, but rather denotes a position and an orientation. An origin may be declared relative to the beginning or end of its parent or a sibling, and this placement determines its orientation. The orientation determines how other siblings may reference the origin; for example, if a child is placed at the end of a parent origin, no sibling origins may be declared “above” it.

<vector>

The non-terminal “vector” consists of the terminal “above” or the terminal “below” followed by an the non-terminal “offset.” The declaration determines the meaning of offset: “above” indicates that the offset will be the lower boundary of the declared origin while “below” indicates that the offset will be the upper boundary.

<offset>

The non-terminal “offset” is either the non-terminal “position” or the non-terminal “name.” A name must be the identifier of a previously declared sibling origin. When placing an origin “above” a sibling, the upper boundary of the reference origin is used as the lower boundary of the origin being declared. Similarly, when placing an origin “below” a sibling, the lower boundary of the reference origin is used as the upper boundary of the origin being declared.

<position>

A position can be either of the special terminals “start” or “end”, or it can be a physical offset followed by an optional logical address. Though inessential to the grammar, the terminal symbols “phy” and “log” prevent accidental macro expansion problems and add clarity for inexperienced users. The physical offset is measured from the beginning of the parent origin. The logical address is required for origins of logical type that are declared relative to a physical origin, and are optional otherwise. It must be omitted if the declared origin is of physical type. The exceptional origin types “wcodorg” and “resvorg” may be declared with logical offsets, making them logical origins. In the absence of logical offsets, they will still be logical origins if declared as children of a logical origin. The terminals “start” and “end” indicate the lower and upper boundaries of the parent origin, respectively. In the absence of a parent origin, the extents are the physical addressable range defined by the ME CR.

<size>

The non-terminal size is either the terminal symbol “size” followed by an integer or the symbol “to” followed by an integer. The integer following “size” specifies the number of bytes the origin contains. The “to” terminal indicates where the origin ends, in which case its size is the absolute value of the difference between the end and beginning of the origin. Note that since origins can be defined in terms of their lower or upper boundaries, “to” always specifies the complimentary boundary.

6.5.1.4 Origin Declaration Start and End Syntax

<start>

The non-terminal start is simply the terminal “#orgstart”. It must occur exactly once and must occur before the end non-terminal. All origin declarations must follow this non-terminal.

<end>

The non-terminal end is simply the terminal “#orgend”. It must occur exactly once and must occur after the start non-terminal. When the compiler encounters this non-terminal, it locks the definitions of the origins and performs final collision detection. All origin declarations must precede this non-terminal.

6.5.1.5 Origin Application Syntax

<orguse>

The orguse non-terminal specifies which region the compiler should use for the origin type corresponding to name. The name non-terminal must be a previously declared origin. All occurrences of orguse must follow orgend.

<action>

The action non-terminal specifies what action the compiler should take for the named origin and may be the terminal “apply” or “resume”. The “apply” terminal signifies resetting the memory region, and should be used with care since the compiler may have already generated code or data to the origin. The “resume” terminal signifies switching to the origin with its state preserved before a previous orguse refocused the compiler's attention.

The terminals “apply” and “resume” have no effect on a region of type “xvarorg”.

6.5.1.6 Origin Macro Declaration Syntax

<macdef>

The non-terminal macdef is the terminal “#orgdef” followed by the non-terminal define and signifies a macro declaration based on an attribute of a previously declared origin. Although the compiler does not force restrictions on where one may place a macdef, prudence dictates the placing them after orgend is the logical choice in most cases.

<define>

The define non-terminal represents the assignment of an origin attribute to a specific macro name. The non-terminal name must be a valid C macro identifier not previously declared. The orgval non-terminal represents an origin attribute as explained below.

<orgval>

An orgval is the non-terminal name, which must be a previously declared origin, followed by an optional non-negative integer, followed by the optional terminal "boundary", followed finally by the aspect non-terminal. The optional integer specifies an origin fragment if the given origin is fragmented, otherwise the compiler ignores it. A user must access fragments of an origin linearly as if they were elements of a C array. Thus, one accesses the first fragment through index zero, and so forth. The optional terminal "boundary" signals the compiler to return attributes that the origin had before it were fragmented or shortened by the declaration of any child origins.

<aspect>

The aspect non-terminal represents an individual aspect of an origin. This non-terminal may be one of three things. It may be the "size" terminal, in which case the compiler assigns the size of the origin in bytes to the macro. It may be the "fragments" terminal, wherein the compiler will assign the number of fragments within the origin to the macro. Lastly, it may be the non-terminal quality followed by position as explained below.

<quality>

This non-terminal specifies a quality of a particular boundary, and may be any of the terminal symbols "physical", "logical", or "segment". Each correspond to the physical address, logical address, and segment value respectively of the origin boundary in context. If the origin is not a logical origin, then the segment and logical terminals will represent the physical boundary converted to an xxx:Exxx address type.

<position>

The position non-terminal is either "start" or "end", and represents the beginning or end respectively of the origin in context.

6.5.2 Origins Prior to Dynamic C 10.21

The following grammar (in BNF) describes the syntax used for the declaration of origin statements prior to Dynamic C version 10.21.

origin-directive : #*origin-type identifier origin-designator*

origin-designator : *action-expression* | *origin-declaration*

origin-declaration : *physical-address size [follow-qualifier][I&D-qualifier][action-qualifier]*
[debug-qualifier]

origin-type: *rcodorg* | *xcodorg* | *wcodorg* | *wvarorg* | *rvarorg* | *rconorg*

follow-qualifier : follows *identifier* [*splitbin*]

I&D-qualifier : *ispace* | *dspace*

action-qualifier : *resume* | *apply*

size : *constant-expression*

physical-address : *constant-expression constant-expression*

The non-terminals, *identifier* and *constant-expression*, are defined in the ANSI C specification. Basically, an *identifier* is a sequence of letters and digits that must start with a letter. The underscore character is considered a letter. The definition of *constant-expression* is more involved as it winds up the restricted subset of operators that are allowed in the evaluation of the expression, but the result is a constant. For a comprehensive definition of the non-terminals, *identifier* and *constant-expression*, please refer to Appendix A in "The C Programming Language" by Kernighan and Ritchie.

6.5.2.1 Origin Directive Semantics

An origin directive associates a code pointer and a memory region with a particular type of code. The type of code is specified by *#origin-type*.

Table 6-3 Origin Types Recognized by the Compiler

Origin Type	Keyword
root code	rcodorg
xmem code	xcodorg
watch code	wcodorg
watch code	wvarorg
root data	rvarorg
root constants	rconorg

All code sections (`rcodorg`, `xcodorg` code and `wcodorg`) grow up. All non-constant data sections (`rvarorg`) grow down. Root constants are generated to the `rcodorg` region when separate I&D space is disabled. When separate I&D space is enabled, root constants are generated to the `rconorg` region. `xdata` and `xstring` are generated to the current `xcodorg` region.

All origin directives must have a unique ANSI C *identifier*. The scope of this identifier is only with other origin directives or declarations.

6.5.2.2 Defining a Memory Region

Each memory region is defined by calculating a physical address from an 8-bit base address (first *constant-expression of physical-address*) and a 16-bit logical address (second *constant-expression of physical-address*). The size of the memory region is determined by 20-bit *size*. Overflow of these three values is truncated.

6.5.2.3 Action Qualifiers

The keywords `apply` and `resume` are *action-qualifiers*. They tell the compiler to generate code or data in the memory region specified by *identifier*. An `apply` action resets the code or data pointer for the specified region to the starting physical address of the region and makes the region active. A `resume` action does not reset the code or data pointer, but does make the memory region active.

A region remains active (i.e., the compiler will continue to generate code or data to it) until another region of the same *origin-type* is activated with an `apply` or `resume` action or until the memory region is full.

6.5.2.4 I&D Qualifiers

The `ispace` and `dspace` qualifiers suppress compiler warnings regarding collisions between the two logical regions and the physical memory space. When an `ispace` or `dspace` qualifier is used in an origin directive, that directive is no longer collision checked against origin directives in the other space. For example, a `rcodorg` directive with the `ispace` qualifier is not checked against any origin directives with a `dspace` qualifier.

6.5.2.5 Follow Qualifiers

The option *follow-qualifier* is best described with an example. First, let us declare `yourcode` in an origin statement containing an *origin-declaration*. A *follow-qualifier* can only name a region that has already been declared in an *origin-declaration*.

```
#xcodorg yourcode 0x0 0x5000 0x500
```

then the origin statement:

```
#xcodorg mycode 0x0 0x5500 0x500 follows yourcode
```

tells the compiler to activate `mycode` when `yourcode` is full. This action does an implicit resume on the memory region identified by `yourcode`. In this example, the implicit resume also generates a jump to `mycode` when `yourcode` is full. For data regions, the data that would overflow the region is moved to the region that follows. Combined data and code regions (like `#rcodorg`) use both methods, which one is used depends on whether code or data caused the region to overflow. In our example, if data caused `yourcode` to overflow, the data would be written to the memory region identified by `mycode`.

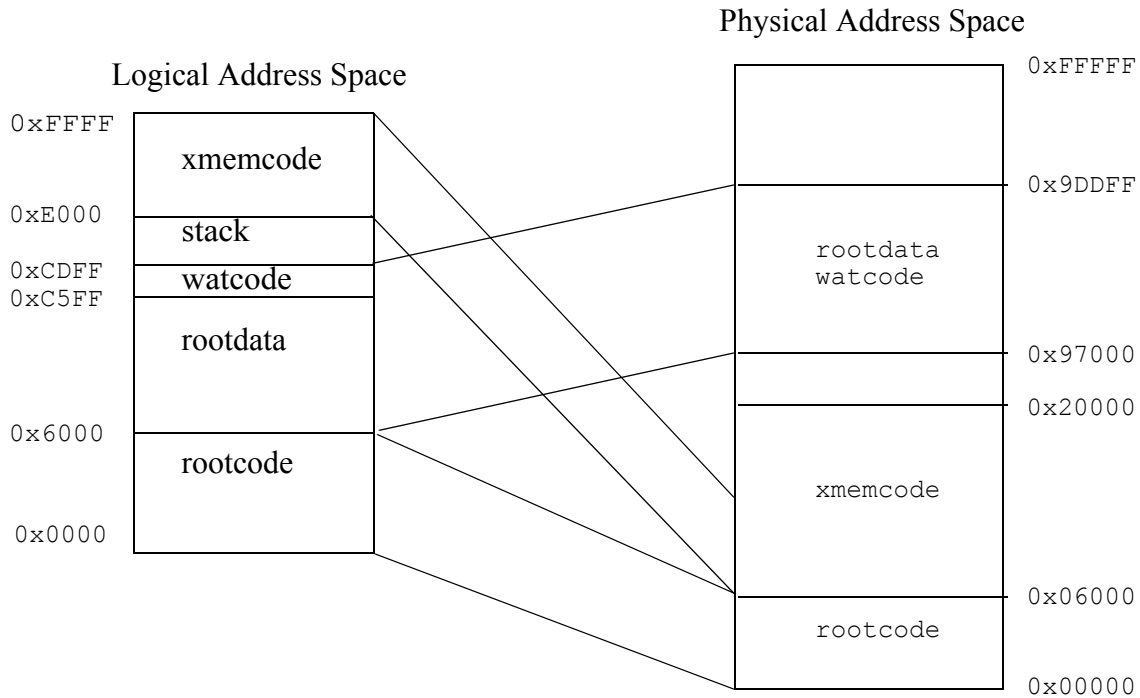
6.5.2.6 Origin Directive Examples

The diagram below shows how the origin directives define the mapping between the logical and physical address spaces.

```
#define DATASEGVAL 0x91
#rvarorg rootdata (DATASEGVAL) 0xC5FF 0x6600 apply // grows down
#rcodorg rootcode 0x00 0x0000 0x6000 apply
#wcodorg watcode (DATASEGVAL) 0xC600 0x0400 apply
#xcodorg xmemcode 0xF8 0xE000 0x1A000 apply

// data declarations start here
```

Dynamic C defines macros that include information about compiling to RAM or flash, and identifying memory device types, memory sizes, and board type. The origin setup shown above differs from that included in the standard BIOS included with Dynamic C as the standard BIOS uses additional macro values for dealing with a wider range of boards and memory device types.



NOTE: This mapping assumes separate I&D space is disabled.

6.5.2.7 Origin Directives in Program Code

To place programs in different places in root memory or to compile a boot strapping program, such as a pilot BIOS or cold loader, origin directives may be used in the user's program code.

For example, the first line of a pilot BIOS program, `pilot.c`, would be

```
#rcodorg rootcode 0x0 0x0 0x6000 apply
```

A program with such an origin directive could only be compiled to a `.bin` file, because compiling it to the target would overwrite the running BIOS.

6.5.2.8 Origin Directive to Reserve Blocks of Memory

With the Rabbit 4000, the compiler generates an origin table that contains the blocks that are reserved for code and data origin or other non-xalloc use. With this change, the method of reserving a block of memory so that `xalloc()` does not use it has also changed. To reserve a block of memory in DC 9.30 and later, the `#resvorg` should be used. All other origins (e.g., `#rcodorg`, `#rvarorg`, etc.) are also tracked by the compiler and those blocks are entered into the origin table generated by the compiler so they are not used by `xalloc()`.

The `#resvorg` is used as follows:

```
#resvorg <NAME> segment offset size [reserve]
```

For example, the following code would reserve the entire flash memory in flash compile mode

```
#resvorg flashmem 0x0 0x0 0x80000 reserve
```

The `reserve` keyword must be added to the end to reserve the entire block of memory.

Some applications may require that fixed regions of RAM be reserved for their own use. For example, you may want to reserve the upper half of a 512K RAM in Flash compile mode. To reserve this you need to add the following line of code to `\LIB\BIOSLIB\STDBIOS.C` just below the “`#resvorg removeflash 0x0 0x0 0x80000 reserve`.”

```
#ifdef RESERVE_UPPER_RAM
    #resvorg reserveupperram 0xC0 0x0 RESERVE_UPPER_RAM
        batterybacked reserve
#endif
```

This tells the compiler to reserve `RESERVE_UPPER_RAM` bytes from physical address `0xC0000` by adding it to the origin table. This removes this memory block from the available `xalloc` memory.

In the Defines tab of the Options | Project Options dialog, enter the amount of memory you want to reserve. For example,

```
RESERVE_UPPER_RAM=0x40000
```

would reserve physical memory from `0xC0000-0xFFFFF` and make it unavailable for `xalloc`. You can then access this memory directly from your program as follows:

```
main() {
    long addr;
    addr = 0xC0000; // point to block reserved for my use
}
```


7. The System Identification and User Blocks

The BIOS supports a System Identification block and a User block. These blocks are placed at the top of the primary flash memory. Identification information for each device can be placed in the System ID block for access by the BIOS, flash driver, and users. This block contains specific part numbers for the flash and RAM devices installed, the product's serial number, Media Access Control (MAC) address if an Ethernet device, and so on. The earliest version of the System ID for Rabbit 4000 products is version 4, which is a mirrored images type.

When mirrored, there are two combined ID/User blocks images placed contiguously at the top of the primary flash, from the top down as follows: ID "A" + User "A" + ID "B" + User "B." Ordinarily, only one of the ID/User blocks images is valid at a time, and the valid ID/User blocks image alternates between "A" and "B" at each call to the `writeUserBlock()` function. If both "A" and "B" images are simultaneously marked valid, the "A" (topmost) image is taken to be correct. Version 5 ID blocks can be configured as described above and can also be configured so that the User block is mirrored and the System ID block is not. If a version 5 ID block is configured so that only the User block is mirrored, the images will be ID + User "A" + User "B".

If Dynamic C does not find a System ID block on a device, the compiler will assume that it is a BL1810 (Jackrabbit) board. It is recommended that board designers include System ID blocks in their products with unused fields zeroed out to maximize future compatibility.

The System ID block has information about the location of the User block. The User block is for storage of calibration constants and other persistent data the user wishes to keep in flash. It is strongly recommended that the User block (using `writeUserBlock()`) or the Flash File System be used for storage of persistent data. Writing to arbitrary flash addresses at run-time is possible using `WriteFlash()` or `WriteFlash2()`, but could lead to compatibility problems if the code were to be used on a different type of flash, such as a huge, non-uniform sector size flash.

For example, some flash types have a single sector as big as 128K bytes at the bottom. Writing to any part of the sector generally requires erasing the whole sector, so a write to store data in that sector would have to save the contents of the whole sector in RAM, modify the section to be changed, and write the whole sector back. This is obviously impractical. Although Rabbit does not currently sell products with this type of flash, there is no guarantee that future flash market conditions won't require that such flash types be used. Other board designers may have to deal with the same flash market issues. The User block is implemented in a way that preserves forward binary compatibility with a wide range of flash devices.

7.1 System ID Block Details

The BIOS will read the System ID block during startup. If the BIOS does not find an ID block, it sets all fields to zero in the data structure `SysIDBlock`. The user may access the information contained in the System ID block by accessing `SysIDBlock`.

7.1.1 Definition of SysIDBlock

The following global data structures are defined in `IDBLOCK.LIB` and are loaded from the flash device during BIOS startup. Users can access this struct in RAM if they need information from it. The `reserved[]` field will expand and/or shrink to compensate for the change in size. Items marked ‘**’ are essential for proper functioning of the System ID block and certain features (e.g., TCP/IP needs the MAC address). Items marked ‘*’ are desirable for future compatibility.

```
typedef struct _SysIDBlockType2 {
    uint8    flashMBC;        // Memory Bank Configurations
    uint8    flash2MBC;
    uint8    ramMBC;
    uint32   devSpecLoc;     // Count of additional memory devices
    immediately
                                // preceding this block
    uint32   macrosLoc;     // Start of the macro table for additional
    board
                                // configuration options.
    uint32   driversLoc;    // offset to preloaded drivers start from
    ID block
                                // start (positive is below ID block)
    uint32   ioDescLoc;     // offset to I/O descriptions start from
    ID block
                                // start (positive is below ID block)
    uint32   ioPermLoc;     // offset to User mode I/O permissions
    start from ID
                                // block start (positive is below ID
    block)
    uint32   persBlockLoc;  // offset to persistent storage block area
    start from
                                // ID block start (positive is below ID
    block)
    uint16   userBlockSiz2; // size of v. 5 "new style" mirrored User
    block image
    uint16   idBlockCRC2;   // CRC of SysIDBlockType2 type with
    idBlockCRC2
                                // member reset to zero and base CRC
    value of
                                // SysIDBlock.idBlockCRC
} SysIDBlockType2;
```

```

typedef struct {
    int tableVersion;           // version number for this table layout**
    int productID;             // Rabbit part #
    int vendorID;              // 1 = Rabbit
    char timestamp[7];         // YY/M/D H:M:S
    long flashID;              // Manufacturer ID/ Product ID, 1st flash
*
    int flashType;             // Write method
    int flashSize;             // in 1000h pages
    int sectorSize;           // size of flash sector in bytes
    int numSectors;           // number of sectors
    int flashSpeed;           // in nanoseconds *
    long flash2ID;            // Manufacturer ID/ Product ID, 2nd flash
*
    int flash2Type;           // Write method, 2nd flash
    int flash2Size;           // in 1000h pages, 2nd flash
    int sector2Size;         // byte size of 2nd flash's sectors
    int num2Sectors;         // number of sectors
    int flash2Speed;         // in nanoseconds, 2nd flash *
    long ramID;              // Rabbit part #
    int ramSize;              // in 1000h pages *
    int ramSpeed;            // in nanoseconds *
    int cpuID;               // CPU type ID *
    long crystalFreq;        // in Hertz *
    char macAddr[6];         // Media Access Control (MAC) address **
    char serialNumber[24];   // device serial number
    char productName[30];   // NULL-terminated string

    // Begin new version 5 System ID block member structure.
    SysIDBlockType2 idBlock2; // idblock
    // End new version 5 System ID block member structure.
    char reserved[1];        // reserved for later use - size can grow
    long idBlockSize;        // number of bytes in the SysIDBlock
} struct **
    unsigned userBlockSize; // User block size, in bytes (right below
ID block)**
    unsigned userBlockLoc; // offset in bytes of start of User block
from this one**
    int idBlockCRC;        // CRC of this block (when this field is
set to 0) **
    char marker[6];        // should be 0x55 0xAA 0x55 0xAA 0x55
0xAA**
} SysIDBlock;

```

7.1.2 Reading the System ID Block

To read the ID block from the flash instead of getting the information from `SysIDBlock`, call `_readIDBlock()`.

`_readIDBlock`

```
int _readIDBlock(int flash_bitmap);
```

DESCRIPTION:

Attempts to read the system ID block from the highest flash quadrant and save it in the system ID block structure. It performs a CRC check on the block to verify that the block is valid. If an error occurs, `SysIDBlock.tableVersion` is set to zero.

This function supports combined System ID/User blocks sizes of `sizeof(SysIDBlock)` and from 4KB to 64KB, inclusive, in 4KB steps. Prior versions of Dynamic C only supported mirrored combined block sizes of `sizeof(SysIDBlock)`, 8KB, 16KB and 24KB or unmirrored combined System ID/User blocks sizes of `sizeof(SysIDBlock)` and from 4KB to 32KB, inclusive, in 4KB steps.

PARAMETER

flash_bitmap Bitmap of memory quadrants mapped to primary flash.

Examples:

0x01 = quadrant 0 only

0x03 = quadrants 0 and 1

0x0C = quadrants 2 and 3

RETURN VALUE:

- 0: Successful
- 1: Error reading from flash
- 2: ID block missing
- 3: ID block invalid (failed CRC check)

LIBRARY

`IDBLOCK.LIB`

7.1.2.1 Determining the Existence of the System ID Block

In Dynamic C versions prior to 7.20, and for ID block versions 1 and 2, the following sequence of events is used by `_readIDBlock()` to determine if an ID block is present:

1. The 16 bytes at the top of the primary flash are read into a local buffer. (If a 256 KB flash is installed, the 16 bytes starting at address 0x3FFF0 will be read.)
2. The last six bytes of the local buffer are checked for an alternating sequence of 0x55, 0xAA, 0x55, 0xAA, 0x55, 0xAA. If this is not found, the block does not exist and an error (-2) is returned.
3. The ID block size (`=SIZE`) is determined from the first 4 bytes of the 16-byte buffer.
4. A block of bytes containing all fields from the start of the `SysIDBlock` struct up to *but not including* the reserved field is read from flash at address 0x40000-SIZE, essentially filling the `SysIDBlock` struct except for the reserved field (since the top 16 bytes have been read earlier).
5. The CRC field is saved in a local variable, then set to 0x0000. A CRC check is then calculated for the entire ID block *except the reserved field* and compared to the saved value. If they do not match, the block is considered invalid and an error (-3) is returned. The CRC field is then restored. The reserved field is avoided in the CRC check since its size may vary, depending on the size of the ID block.

Determining the existence of a valid mirrored ID block may be slightly more complicated, requiring the above sequence of events to be repeated at several locations below the top of the primary flash. See [Figure 7.2](#) below for complete details.

Not all fields are filled in different versions of the ID block. The table below lists the first ID block version that filled each field and whether that field is absolutely required by Dynamic C for normal operation. (Much of the ID block data is useful, but not critical.)

Table 7-1 The System ID Block

Offset from start of block	Size (bytes)	Description	Filled as of Version	Required
00h	2	ID block version number	1	x
02h	2	Product ID	1	x
04h	2	Vendor ID	2	
06h	7	Timestamp (YY/MM/D/H/M/S)	1	
0Dh	4	Flash ID	2	
11h	2	Flash size (in 1000h pages)	2	
13h	2	Flash sector size (in bytes)	2	
15h	2	Number of sectors in flash	2	
17h	2	Flash access time (nanoseconds)	4	
19h	4	Flash ID, 2nd flash	2	
1Dh	2	Flash size (in 1000h pages), 2nd flash	2	
1Fh	2	Flash sector size, 2nd flash (in bytes)	2	

Table 7-1 The System ID Block (Continued)

Offset from start of block	Size (bytes)	Description	Filled as of Version	Required
21h	2	Number of sectors in 2nd flash	2	
23h	2	Flash access time, in nanoseconds, for the 2nd flash	4	
25h	4	RAM ID	2	
29h	2	RAM size, in 1000h pages	2	
2Bh	2	RAM access time, in nanoseconds	4	
2Dh	2	CPU ID	3	
2Fh	4	Crystal frequency (Hertz)	2	
33h	6	Media Access Control (MAC) address	1	x
39h	24	Serial number (as a null-terminated string)		
51h	30	Product name (as a null-terminated string)		
6Fh	27	Version 5 System ID block member structure (SysIDBlockType2)	5	
8Ah	N	Reserved (variable size)		
SIZE - 10h	4	Size of System ID block, in bytes	1	x
SIZE - 0Ch	2	Size of User block, in bytes	1	x
SIZE - 0Ah	2	Offset, in bytes, of User block location from start of this block	1	x
SIZE - 08h	2	CRC value of System ID block (when this field = 0000h)	1	x
SIZE - 06h	6	Marker, should = 55h AAh 55h AAh 55h AAh	1	x

7.1.3 Writing the System ID Block

The `WriteFlash()` function does not allow writing to the System ID block. If the System ID block needs to be rewritten, a utility to do so is available for download from the Rabbit website:

www.rabbit.com/support/downloads/downloads_feat.shtml

or contact Rabbit's Technical Support.

7.2 User Block Details

Starting with the System ID block version 3, two contiguous copies of the combined ID/User blocks are used, or in the case of the version 5 ID block, two contiguous copies of the User block are used. Only one image contains "valid" data at any time. When data is written to a mirrored User block, the currently invalid User block image is updated first and then validated by changing its `marker[5]` byte from 0x00 to 0xAA. This marker is located in the user block itself in version 5 ID blocks where the mirrored user blocks are separate from the System ID Block, and in version 5 and prior ID blocks where the User block and System ID blocks are combined, the marker byte is located in the System ID Block. Next, the previously valid image is invalidated by changing its `marker[5]` byte from 0xAA to 0x00. Finally, the newly invalidated image is updated. In this way, there is only a short period of time in which both images are marked valid, and at no time are both data blocks marked invalid. If a power failure occurs at any time during the User block update, the BIOS will still find a valid ID block and the valid User block will contain data from the last completed update transaction. In addition to making data more secure, this redundancy allows even very large sector flash types to be used without requiring a large RAM buffer to temporarily store the contents of a sector, since sectors must be erased before they can be written.

In Dynamic C 7.20 and later, the possibility of mirrored combined ID/User blocks requires that multiple locations in flash must be checked for a valid ID block. In versions 7.20 through 7.3x, the sequence described above in [Section 7.1.2.1](#) is used to check not only the top of the primary flash, but also 8KB, 16KB and 24KB below the top, and an error is returned only if no valid ID block is found at any of these locations. Note the implication here that mirrored combined ID/User blocks are limited to one of 8KB, 16KB, or 24KB in size. Dynamic C versions 8 and later check more locations in flash, from the top down, at each lower 4KB boundary to 64KB below the top. This allows Dynamic C 8 and up to recognize a combined ID/User blocks size that is any multiple of 4KB up to a maximum of 64KB.

If the version of the System ID block doesn't support the User block, or no System ID block is present, then the 8 KB starting 16 KB from the top of the primary flash are designated the User block area. However, to prevent errors arising from incompatible large sector configurations, this will only work if the flash type is small sector. Rabbit manufactured boards with large sector flash will have valid System ID and User blocks, so this should not be a problem on Rabbit-based boards.

7.2.1 Boot Block Issues

The System ID and User block implementations have been designed to accommodate huge, non-uniform sector flash types, but it is necessary to use 'T' type parts with such flash types so that the smaller boot block sectors at the top can be used for the blocks. 'B' parts have smaller boot block sectors at the bottom.

No code is included with Dynamic C to lock boot blocks, and users should not lock boot blocks unless they are sure they will never write to the blocks after the System ID block is written. If a boot block lock is irreversible, we strongly recommend never locking it.

7.2.2 Reserved Flash Space

The macro `MAX_USERBLOCK_SIZE` (default `0x8000`) in the BIOS tells the Dynamic C compiler how much flash at the top of the primary flash is excluded from use by the compiler for `xmem` functions. For any application, whether compiled to a single target board or for multiple target boards, the `MAX_USERBLOCK_SIZE` macro value defined in `RabbitBios.c` must not be lower than the amount of flash required for the System ID/User blocks on the target board with the largest requirement. Note that in the case of mirrored combined ID/User blocks (version 3 and up), the amount of flash that must be reserved is double the size of one combined ID/User block image. For example, if a target board has mirrored combined ID/User blocks and the size of one image is 16 KB (`0x4000` bytes), then the minimum value defined for the `MAX_USERBLOCK_SIZE` macro is 32 KB (`0x8000` bytes).

All of the default `MAX_USERBLOCK_SIZE` reserved space is not necessarily needed by the System ID and User blocks, but reserving this much space maximizes forward binary compatibility should a product switch to any of various huge, non-uniform sector flash types. Some of these types have sectors of 8 KB, 8 KB and 16 KB at the top, and the mirrored design of the User block requires that these 3 sectors be used. If you do not need the User block and are not concerned with forward binary compatibility, the `MAX_USERBLOCK_SIZE` macro value could be safely lowered (protecting the sector containing the ID block) to as little as `0x4000` (16 KB), but only if the System ID block is rewritten to set the User block size to zero (i.e., no run-time flash writes can occur, such as to the User block or to a flash file system).

Reducing the `MAX_USERBLOCK_SIZE` macro value will only increase available `xmem` code space, not root code space which is generally in shorter supply. To increase available `xmem` code space, the following general procedure should be followed:

1. Determine that binary forward compatibility with large sector flash types as described above is not an issue. This means that the application will only ever run on target boards equipped with small sector flash (i.e., uniform sectors of a size no larger than 4 KB).
2. Determine the application's minimum User block size requirement. If the application does not write to the User block, this size is zero.
3. If the target board has factory calibration constants stored in the User block, add the size reserved for these constants. Consult your hardware manual for the reserved size required.
4. Add the size of the System ID block, which is 132 bytes for versions 2 through 4.
5. Round this total size up to the next higher 4 KB block boundary.
6. If using mirrored combined (version 3 or 4) ID/User blocks, double the size.
7. Calculate the number of 4 KB blocks required for the total size.
8. Edit the `write_idblock.c` utility to set the required number of 4 KB blocks, and write a new ID block onto the target board.
9. Repeat the previous steps for every board which is to be programmed with the application(s) compiled using the updated `MAX_USERBLOCK_SIZE` macro value.
10. Edit the `RabbitBios.c` file to update the `MAX_USERBLOCK_SIZE` macro value.

Note that it is especially difficult to effectively reduce the `MAX_USERBLOCK_SIZE` macro value below `0x4000` (16 KB) for the BL20xx or BL21xx board families, which have their combined ID/User blocks size hard-coded in the `FLASHWR.LIB` and `IDBLOCK.LIB` libraries because their stored calibration constants are in a nonstandard place. For this reason, Rabbit strongly recommends not attempting to make System ID/User block changes on these board families.

7.2.3 Reading the User Block

readUserBlock

```
int readUserBlock(void *dest, unsigned addr, unsigned numbytes);
```

DESCRIPTION:

Reads a number of bytes from the User block on the primary flash to a buffer in root memory.

NOTE: portions of the User block may be used by the BIOS for your board to store values such as calibration constants. See the manual for your particular board for more information before overwriting any part of the User block.

PARAMETERS

dest	Pointer to destination to copy data to.
addr	Address offset in User block to read from.
numbytes	Number of bytes to copy.

RETURN VALUE

- 0: Successful
- 1: Invalid address or range
- 2: No valid System ID block found

LIBRARY

IDBLOCK.LIB

readUserBlockArray

```
int readUserBlockArray(void *dests[], unsigned numbytes[], int
    numdests, unsigned addr);
```

DESCRIPTION

Reads a number of bytes from the User block on the primary flash to a set of buffers in root memory. This function is usually used as the inverse function of `writeUserBlockArray()`.

PARAMETERS

dests	Pointer to array of destinations to copy data to.
numbytes	Array of numbers of bytes to be written to each destination.
numdests	Number of destinations.
addr	Address offset in User block to read from.

RETURN VALUE

- 0: Success
- 1: Invalid address or range
- 2: No valid System ID block found (block version 3 or later)

LIBRARY

IDBLOCK.LIB

7.2.4 Writing the User Block

`writeUserBlock`

```
int writeUserBlock(unsigned addr, void *source, unsigned numbytes);
```

DESCRIPTION:

Rabbit-based boards are released with System ID blocks located on the primary flash. Version 2 and later of this ID block has a pointer to a User block that can be used for storing calibration constants, passwords, and other non-volatile data. This block is protected from normal writes to the flash device and can only be accessed through this function. This function writes a number of bytes from root memory to the User block

NOTE: Portions of the User block may be used by the BIOS for your board to store values such as calibration constants! See the manual for your particular board for more information before overwriting any part of the User block.

Backwards Compatibility:

If the version of the System ID block doesn't support the User block, or no System ID block is present, then the 8 KB starting 16 KB from the top of the primary flash are designated the User block area. However, to prevent errors arising from incompatible large sector configurations, this will only work if the flash type is small sector. Rabbit manufactured boards with large sector flash will have valid System and User ID blocks, so this should not be problem on Rabbit-based boards.

If users create boards with large sector flash, they must install System ID block version 3 or greater to use this function, or modify this function.

PARAMETERS

addr	Address offset in User block to write to.
source	Pointer to destination to copy data from.
numbytes	Number of bytes to copy.

RETURN VALUE

- 0: Successful
- 1: Invalid address or range
- 2: No valid User block found (block version 3 or later)
- 3: Flash writing error

LIBRARY

`IDBLOCK.LIB`

writeUserBlockArray

```
int writeUserBlockArray(unsigned addr, void* sources[], unsigned
    numbytes[], int numsources);
```

DESCRIPTION

Rabbit-based boards are released with System ID blocks located on the primary flash. Version 2 and later of this ID block has a pointer to a User block that can be used for storing calibration constants, passwords, and other non-volatile data. The User block is protected from normal write to the flash device and can only be accessed through this function or `writeUserBlock()`.

This function writes a set of scattered data from root memory to the User block. If the data to be written is in contiguous bytes, using the function `writeUserBlock()` is sufficient. Use of `writeUserBlockArray()` is recommended when the data to be written is in noncontiguous bytes, as may be the case for something like network configuration data. See the *Rabbit Microprocessor Designer's Handbook* for more information about the System ID and User blocks.

Backwards Compatibility:

If the System ID block on the board doesn't support the User block, or no System ID block is present, then the 8K bytes starting 16K bytes from the top of the primary flash are designated User block area. This only works if the flash type is small sector. Rabbit manufactured boards with large sector flash will have valid System ID and User blocks, so is not a problem on Rabbit-based boards. If users create boards with large sector flash, they must install System ID blocks version 3 or greater to use this function, or modify this function.

PARAMETERS

addr	Address offset in the User block to write to.
sources	Array of pointer to sources to copy data from.
numbytes	Array of number of bytes to copy for each source. The sum of the lengths in this array must not exceed 32767 bytes, or an error will be returned.
numsources	Number of data sources.

RETURN VALUE

- 0: Successful.
- 1: Invalid address or range.
- 2: No valid User block found (block version 3 or later).
- 3: Flash writing error.

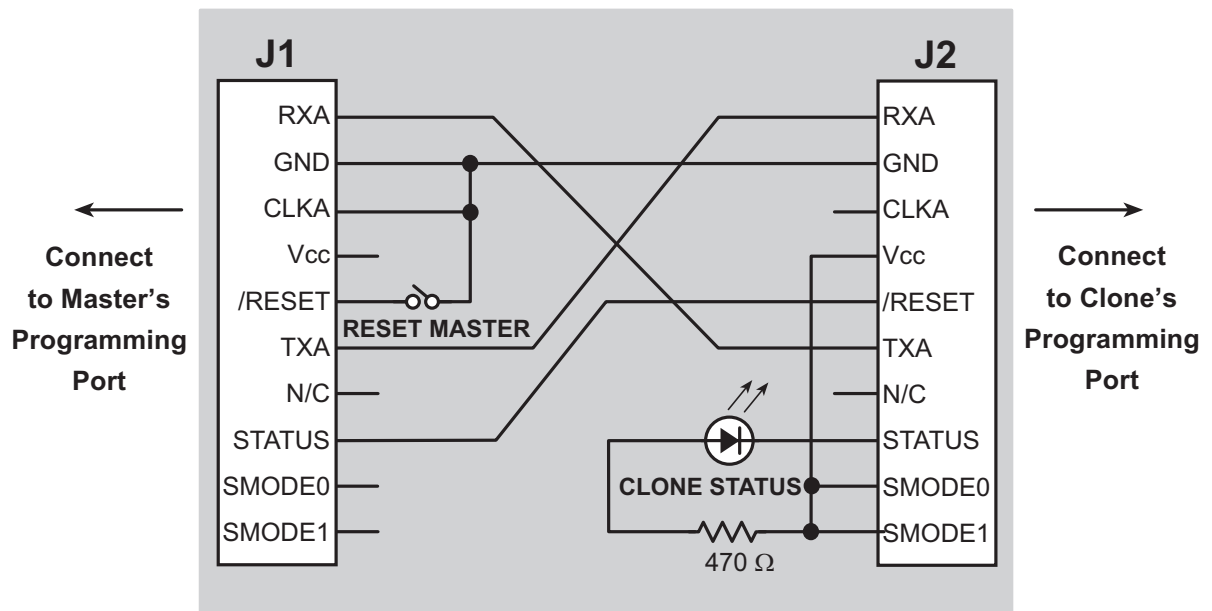
LIBRARY

IDBLOCK.LIB

8. BIOS Support for Program Cloning

The BIOS supports copying designated portions of flash memory from one controller (the master) to another (the clone). The Rabbit Cloning Board connects to the programming port of the master and to the programming port of the clone. This simple circuit can easily be incorporated into test fixtures for fast production.

Figure 8-1 Cloning Board



8.1 Overview of Cloning

If the cloning board is connected to the master, the signal CLKA is held low. This is detected in the BIOS after the reset ends, invoking the cloning support of the BIOS. If cloning has been enabled in the master's BIOS, it will cold boot the target system by resetting it and downloading a primary boot program. The master then sends the entire user program along with other user selected portions of flash memory to the clone, where the boot program receives it and stores it in RAM then copies it to flash. Optionally, the cloned program can begin running on the slave.

For more details on cloning, see Technical Note 207 "Rabbit Cloning Board," available at: rabbit.com.

8.2 Creating a Clone

Before cloning can occur, the master controller must be readied. Once this is done, any number of clones may be created from the same master.

8.2.1 Steps to Enable and Set Up Cloning

The step-by-step instructions to enable and set up cloning on the master are in Technical Note 207. In brief, the steps break down to: attaching the programming cable, running Dynamic C, making any desired changes to the cloning macros, and then compiling the BIOS and user program to the master.

The only cloning macro that must be changed is `ENABLE_CLONING`, since the default condition is that cloning is disabled.

8.2.2 Steps to Perform Cloning

Once cloning is enabled and set up on the master controller, detach the programming cable and attach the cloning board to the master and the clone. Make sure the master end of the cloning board is connected to the master controller (the cloning board is not reversible) and that pin 1 lines up correctly on both ends. Once this is done, reset the master by pressing **Reset** on the cloning board. The cloning process will begin.

8.2.3 LED Patterns

While cloning is in progress the LED on the Cloning board will toggle on and off every 1-1.5 seconds. When cloning completes, the LED stays on. If any error occurs, the LED will start blinking quickly. Older versions of cloning used different LED patterns, but the Rabbit 4000 is only supported by versions that use the pattern described here.

8.3 Cloning Questions

The following subsections answer questions about different aspects of cloning.

8.3.1 MAC Address

Some Ethernet-enabled boards do not have the EEPROM with the MAC address. These boards can still be used as a clone because the MAC address is in the system ID block and this structure is shipped on the board and is not overwritten by cloning unless `CL_INCLUDE_ID_BLOCKS` is set to one.

If you have a custom-designed board that does not have the EEPROM or the system ID block, you may download a program at:

http://www.rabbit.com/support/feature_downloads.html

to write the system ID block (which contains the MAC address) to your board.

To purchase a MAC address go to:

<http://standards.ieee.org/regauth/oui/index.shtml>

8.3.2 Different Flash Types

Since the BIOS supports a variety of flash types, the flash EPROM on the two controllers do not have to be identical. Cloning works between master and clone controllers that have different-type flash chips because the master copies its own universal flash driver to the clone. The flash driver determines the particulars of the flash chip that it is driving.

8.3.3 Different Memory Sizes

It is recommended that the cloning master and slave both have the same RAM and flash sizes.

8.3.4 Design Restrictions

Digital I/O line PB1 should not be used in the design if cloning is to be used.

9. Low-Power Design and Support

With the Rabbit 4000 microprocessor it is possible to design systems that perform their tasks with very low power consumption. The Rabbit has several features that contribute to low power consumption. They are summarized here and explained in greater detail in the following section.

- Special chip select features minimize power consumption by external memories.
- The Rabbit core operates at 1.8 V.
- The I/O ring can operate 3.3 or 1.8 V.
- The main crystal oscillator may be divided by 2, 4, 6 or 8.
- When the main crystal oscillator is divided by 4, 6 or 8, the short chip select option is available.
- The 32 kHz oscillator may be used instead of the main oscillator; this is sleepy mode. The 32 kHz oscillator may be divided by 2, 4, 8 or 16; this is ultra sleepy mode. The self-timed chip select option is available in both sleepy and ultra sleepy modes.

Before looking at the Rabbit 4000 low-power features in greater detail, please note that some of the power consumption in an embedded system is unaffected by the clever design features of the microprocessor. As shown in the table below, the current (and thus power) consumption of a microprocessor-based system generally consists of a part that is independent of frequency and a part that depends on frequency.

Table 9-1 Factors affecting power consumption in the Rabbit 4000 microprocessor

Current Consumption Independent of Frequency	Current Consumption Dependent on Frequency
Current leakage.	CMOS logic switching state. ^a
Special circuits (e.g. pull-up resistors).	
Circuits that continuously draw power.	

a. Ordinary CMOS logic uses power when it is switching from one state to another. The power drawn while switching is used to charge capacitance or is used when both N and P field effect transmitters (FETs) are simultaneously on for a brief period during a transition.

9.1 Details of the Rabbit 4000 Low-Power Features

This section goes into more detail about the Rabbit 4000 low-power features.

9.1.1 Special Chip Select Features

Unlike competitive processors, the Rabbit 4000 has two special chip select features designed to minimize power consumption by external memories. This is significant because, if not handled well, external memories can easily become the dominant power consumers at low clock frequencies. Primarily because most memory chips draw substantial current at zero frequency. (When the chip select and output enable are held enabled and all other signals are held at fixed levels.)

In situations where the microprocessor is operating at slow frequencies, such as 2.048 kHz, the memory cycle is about 488 μ s and the memory chip spends most of its time with the chip enable and the output enable on. The current draw during a long read cycle is not specified in most data sheets. The Hynix HY62KF08401C SRAM, according to the data sheet, typically draws 5mA/MHz when it is operating. When performing reads at 2.048 kHz, we've found that this SRAM consumes about 14 mA. At the same frequency, with the short chip select enabled, the SRAM consumes about 23 μ A—a substantial reduction in power consumption.

As shown, both special chip select modes (i.e. short chip select and self-timed chip select) reduce memory current consumption since the processor spends most of its time performing reads (i.e., instruction fetches).

The self-timed chip select feature is available in sleepy and ultra sleepy mode; i.e., when the processor is running off the 32 kHz oscillator, or when the oscillator is divided by 2, 4, 8 or 16.

The short chip select feature may be used when the main oscillator is divided by 4, 6, or 8. This division can be done regardless of whether the clock doubler is on or off. Currently, interrupts must be disabled when both the short chip select feature is enabled and an I/O instruction is used. Interrupts can be disabled for a single I/O instruction by using code such as:

```
push ip          ; save interrupt state
ipset 3          ; interrupts off
ioe ld a, (hl)   ; typical I/O instruction
pop ip          ; reenable interrupts
```

NOTE: Short chip selects and self-timed chip selects only take place during memory reads. During writes the chip selects behave normally.

For a detailed description of the chip select features, please see the *Rabbit 4000 Microprocessor User's Manual*.

9.1.2 Reducing Clock Speed

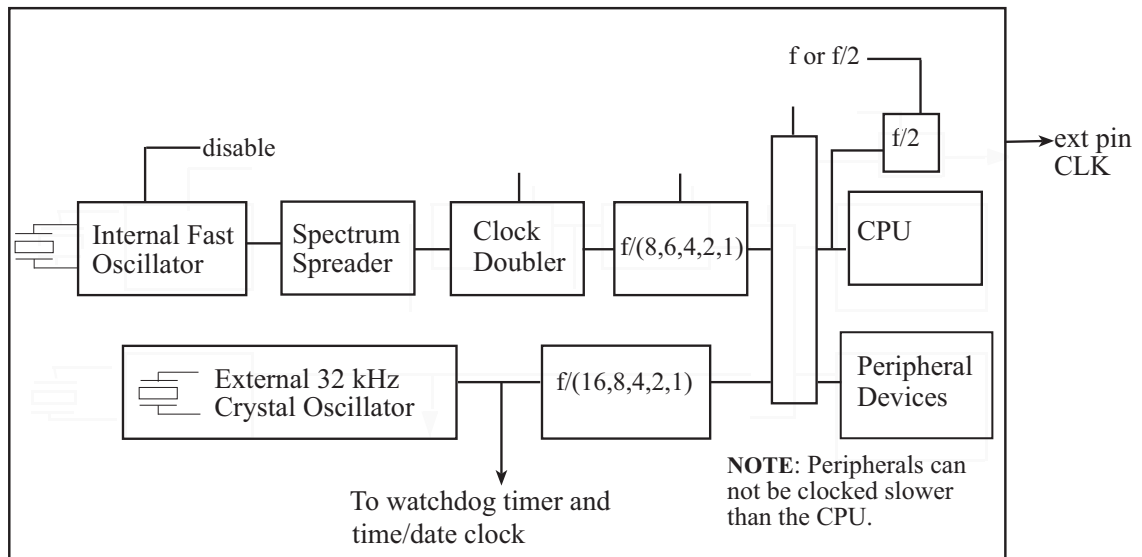
It is important to know that the lowest speed crystal will not always give the lowest power consumption. This is because when the crystal is divided internally, the short chip select option can be used to reduce the chip select duty cycle of the flash memory or fast RAM, greatly reducing the static current consumption associated with some memories.

Some applications, such as a control loop, may require a continuous amount of computational power. Other applications, such as slow data logging or a portable test instrument, may spend long periods with low computational requirements interspersed with short periods of high computational load. At a given operating voltage, the clock speed should be reduced as much as possible to obtain the minimum power consumption that is acceptable.

9.1.3 Preferred Crystal Configuration

The preferred configuration for a Rabbit 4000 based system is to use an external crystal or resonator that has a frequency $\frac{1}{2}$ the maximum internal clock frequency. The oscillator frequency can be doubled and/or divided by 2, 4, 6 or 8, giving a variety of operating speeds from the same crystal frequency. In addition, the 32.768 kHz oscillator that drives the battery-backable clock can be used as the main processor clock. To save the substantial power consumed by the fast oscillator, it can be turned off and the processor can run entirely off the 32.768 kHz oscillator at 32.768 kHz or at 32.768 kHz divided by 2, 4, 8 or 16. This mode of operation (sleepy mode) has a clock speed in the range of 2 kHz to 32 kHz, and a VDD_{core} current consumption in the range of 14 to 22 μA , depending on frequency and voltage.

Figure 9-1 Rabbit 4000 Clock Distribution



9.2 To Further Decrease Power Consumption

In addition to the low-power features of the Rabbit 4000 microprocessor, other considerations can reduce power consumption by the system.

9.2.1 What To Do When There is Nothing To Do

For the very lowest power consumption the processor can execute a long string of `mul` instructions with the DE and BC registers set to zero. Few if any internal registers change during the execution of a string of `mul zero by zero`, and a memory cycle takes place only once in every 12 clocks.

9.2.2 Sleepy Mode

Power consumption is dramatically decreased in sleepy mode. The `VDD_{core}` current consumption is often reduced to the region of 22 μA 3.3 V and 32.768 kHz. The Rabbit 4000 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.

In sleepy mode, most of the power is consumed by:

- memory
- the processor core
- recommended external 32 kHz crystal oscillator circuit

Using the flash memory SST39LF020-45-4C-WH and a self-timed 106 ns chip select, the memory consumed 22 μA at 32 kHz and 1.4 μA at 2 kHz. For a current list of supported flash, please see Technical Note 226 “Supported Flash Devices.” This document is available at:

http://www.rabbit.com/docs/app_tech_notes.shtml

The supported flash devices will give approximately the same values as the flash device that was used for testing. The processor core consumes between 3 and 50 μA at 3.3 V as the frequency is throttled from 2 kHz to 32 kHz, and about 40% as much at 1.8 V. The crystal oscillator circuit consumes 17 μA at 3.3 V. This drops rapidly to about 2 μA at 1.8 V.

Additional power consumption in sleepy mode may come from a low-power reset controller which may consume about 8 μA and CMOS leakage which may consume several μA . The power consumed by CMOS leakage increases with higher temperatures.

NOTE: Periodic interrupts are automatically disabled when the processor is placed in sleepy mode.

Debug is not directly supported in sleepy modes. Please see [Section 9.2.7 on page 78](#) for more information.

9.2.3 External 32 kHz Oscillator

Unlike the Rabbit 2000, the Rabbit 4000 has no internal 32 kHz oscillator. Instead there is a clock input. The recommended external crystal oscillator circuit and the associated battery backup circuit are discussed in Technical Note 235 available on our website:

www.rabbit.com.

9.2.4 Conformal Coating of 32.768 kHz Oscillator Circuit

The 32.768 kHz oscillator circuit consumes microampere level currents. The circuit also has very high input impedance, thus making it susceptible to noise, moisture and environmental contaminants. To avoid leakage due to moisture and ionic contamination it is recommended that the oscillator circuit be conformal coated. This is simplified if all components are kept on the same side of the board as the processor. Feedthroughs that pass through the board and are connected to the oscillator circuit should be covered with solder mask that will serve as a conformal coating for the back side of the board from the processor. Please see Technical Note 303, “Conformal Coating,” and Technical Note 235 “External 32.768 kHz Oscillator Circuits” on the Rabbit website for more information

www.rabbit.com/support/techNotes_whitePapers.shtml

9.2.5 Software Support for Sleepy Mode

In sleepy mode the microprocessor executes instructions too slowly to support most interrupts. Data will probably be lost if interrupt-driven communication is attempted. The serial ports can function but cannot generate standard baud rates when the system clock is running at 32.768 kHz or below.

The 48-bit battery-backable clock continues to operate without interruption.

Usually the programmer will want to reduce power consumption to a minimum for a fixed time period or until some external event takes place. On entering sleepy mode by calling `use32kHzOsc()`, the periodic interrupt is completely disabled, the system clock is switched to 32.768 kHz, and the main oscillator is powered down. The device may be run even slower by dividing the 32kHz oscillator by 2, 4, 8, or 16 with the `set32kHzDivider()` call. When the 32kHz oscillator is divided, these slower modes are called ultra sleepy modes.

On exiting sleepy mode by calling `useMainOsc()`, the main oscillator is powered up, a time delay is inserted to be sure that it has resumed regular oscillation, and then the system clock is switched back to the main oscillator. At this point the periodic interrupt is reenabled.

While in sleepy mode the user may call `updateTimers()` periodically to keep Dynamic C time variables updated. These time variables keep track of seconds and milliseconds and are normally used by Dynamic C routines to measure time intervals or to wait for a certain time or date. `updateTimers()` reads the real-time clock and then computes new values for the Dynamic C time variables. The normal method of updating these variables is the periodic interrupt that takes place 2048 times per second.

NOTE: In ultra sleepy modes, calling `updateTimers()` is not recommended.

Functions are provided to power down the Realtek Ethernet chip as well. By calling the `pd_powerup()` and `pd_powerdown()` functions, the Realtek chip can be placed in and awakened from its own power-down mode. Note that no TCP/IP or Ethernet functions should be called while the Realtek is powered down.

9.2.6 Baud Rates in Sleepy Mode

The available baud rates in sleepy mode are 1024, 1024/2, 1024/3, 1024/4, etc. Baud rate mismatches of up to 5% may be tolerated. The baud rate 113.77 is available as 1024/9 and may be useful for communicating with other systems operating at 110 bps—a 3.4% mismatch. In addition, the standard PC compatible UART 16450 with a baud rate divider of 113 generates a baud rate of 1019 bps, a 0.5% mismatch with 1024 bps. If there is a large baud rate mismatch, the serial port can usually detect that a character has been sent to it, but can not read the exact character.

9.2.7 Debugging in Sleepy Mode

Debugging is not supported in sleepy modes. However, running with no polling (Alt-F9) will avoid the loss of target communications when execution enters sections of code using sleepy mode, and debug communications will resume when the normal operation mode is reenabled.

10. Supported Flash Memories

There are several flash memories that have been qualified for use with the Rabbit 4000 microprocessor. Both small and large sector flash devices are supported. To incorporate a large-sectored flash into an end product, the best strategy is have a small-sectored development board.

Table 10-1 Flash Devices for Rabbit 4000-Based Designs

Device Name	Device Size (bytes)	Write Mode	Operating Voltage	Dynamic C Support
SST39LF040	512x8	byte	3.0-3.6V	Starts w/ version 10
SST39VF040-70-41	512x8	byte	2.7-3.6V	Starts w/ version 10
SST39LF040-45-4C	512Kx8	byte	3.0-3.6V	Starts w/ version 10
SST39LF400A-55-4C	256Kx16	word	3.0-3.6V	Starts w/ version 10
SST39LF800A-55-4C	512Kx16	word	3.0-3.6V	Starts w/ version 10

10.1 Supporting Other Flash Devices

Rabbit does not support flash devices other than those listed in [Table 10-1](#). However, if you wish to use another flash memory, one that still uses the same standard 8-bit JEDEC write sequences as one of the supported flash devices, the existing Dynamic C flash libraries may be able to support it simply by modifying a few values. Not all flash devices can be supported, and the degree of support will vary depending on the flash characteristics.

There are two modifications to be made, depending on the version of Dynamic C that you are using. Step through the list below and perform each action that corresponds to your flash type:

1. The flash device needs to be added to the list of known flash types. This table can be found by searching for the label `FlashData` in the file `LIB\Rabbit4000\BIOSLIB\FLASHWR.LIB`. The format is described in the file and consists of the flash ID code, the sector size in bytes, the total number of sectors, and the flash write mode.

See the comments above the “FlashData:” table in `FLASHWR.LIB` for more information.

2. The same information that was added to the `FlashData` table needs to be added to the `FLASH.INI` file (in the main directory where Dynamic C was installed) for use by the compiler and pilot BIOS. See the top of the file for more information.

10.2 Writing Your Own Flash Driver

Rabbit does not support using a flash memory that is not listed above and that does not use the same standard 8-bit JEDEC write sequences as one of the supported flash memories. If you must use such a flash memory, the flash driver supplied with Dynamic C (`LIB\Rabbit4000\BIOSLIB\FLASHWR.LIB`) provides a model for writing your own flash driver.

11. Troubleshooting Tips for New Rabbit-Based Systems

If the Rabbit design conventions were followed and Dynamic C cannot establish target communications with the Rabbit 4000-based system, there are a number of initial checks and some diagnostic tests that can help isolate the problem.

11.1 Initial Checks

Perform the first two checks with the /RESET line tied to ground.

1. With a voltmeter check for VDDIO, VDDINT, VBAT and VBATIO for the correct voltages. Also check VSSIO and VSSINT for proper connection to ground.
2. With an oscilloscope check the 32.768 kHz oscillator on CLK32K (pin 49). Make sure that it is oscillating and that the frequency is correct.
3. With an oscilloscope check the main system oscillator by observing the signal CLK. With the reset held high and no existing program in the flash memory attached to the processor, this signal should have a frequency one eighth of the main crystal or oscillator frequency.

11.2 Diagnostic Tests

The cold boot mode may be used to communicate with the target system without using Dynamic C. As discussed in Section 4.1, in cold boot mode triplets may be received by serial port A or the slave port. To load and run the diagnostic programs, the easiest method is to use the programming cable and a specialized terminal emulator program over asynchronous serial port A. To use the slave port requires more setup than the serial port method and it is not considered here. Since each board design is unique, it is not possible to give a one-size-fits-all solution for diagnosing board problems. However, using the cold boot mode allows a high degree of flexibility. Any sequence of triplets may be sent to the target.

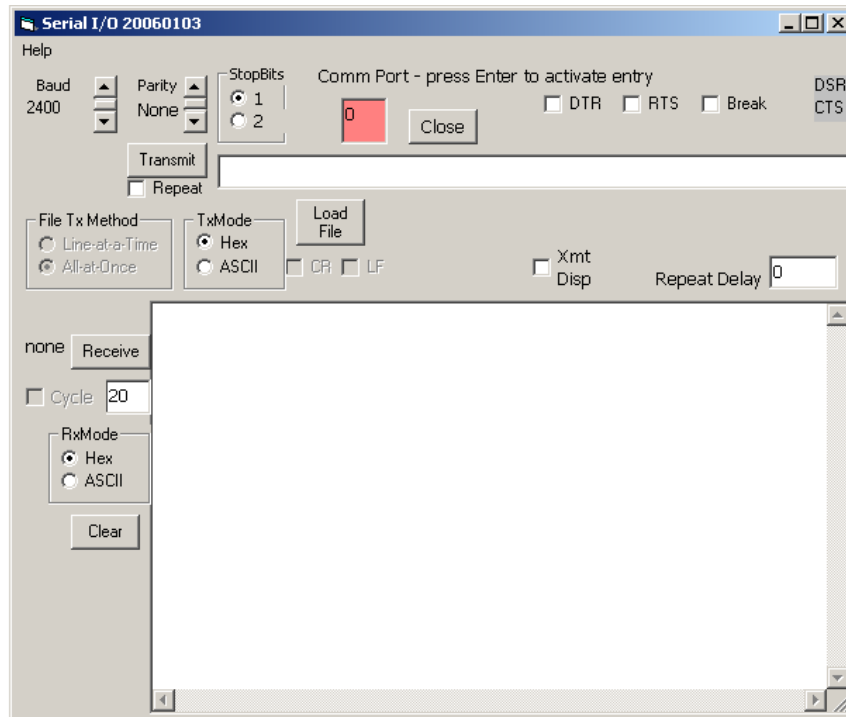
11.2.1 Program to Transmit Diagnostic Tests

The file `SerialIO_1.zip` is available for download at:

http://ftp1.digi.com/support/driver/rabbit_serial_io.zip

The zip file contains the specialized terminal emulator program `serialIO.exe` and several diagnostic programs. The diagnostic programs test a variety of functionality, and allow the user to simulate some of the behavior of the Dynamic C download process.

After extracting the files, double click on `serialIO.exe` to display the following screen.



Click on **Help** at the top left-hand side of the screen for directions for using this program.

A diagnostic program is a group of triplets. You can open the provided diagnostic programs (those files with the extension `.diag`) with Dynamic C or any simple text editor if you would like to examine the triplets that are sent to the target. Also `serialIO.exe` has the option of sending the triplets a line at a time so you can see the triplets in the one-line window next to the Transmit button before they are sent.

NOTE: Connecting the programming cable to the programming connector pulls both SMODE pins high. On reset this allows a cold boot from asynchronous serial port A. The reset may be applied by pushing the reset button on the target board, or by checking then unchecking the box labeled DTR when using `serialIO.exe`.

In the following pages, two diagnostic programs are looked at in some detail. The first one is short and very simple: a toggle of the status line. Information regarding how to check the results of the diagnostic are given. The second diagnostic program checks the processor/RAM interface. This example provides more detail in terms of how the triplets were derived. After reading through these examples, you will be able to write diagnostic programs suited for your unique board design.

11.2.2 Diagnostic Test #1: Toggle the Status Pin

This test toggles the status pin.

1. Apply the reset for at least $\frac{1}{4}$ second and then release the reset. This enables the cold boot mode for asynchronous serial port A if the programming cable is connected to the target's programming connector.
2. Send the following sequence of triplets.

```
80 0E 20      ; sets status pin low
80 0E 30      ; sets status pin high
80 0E 20      ; sets status pin low again
```
3. Wait for approximately $\frac{1}{4}$ second and then repeat starting at step #1.

While the test is running, an oscilloscope can be used to observe the results. The scope can be triggered by the reset line going high. It should be possible to observe the data characters being transmitted on the RXA pin of the processor or the programming connector. The status pin can also be observed at the processor or programming connector. Each byte transmitted has 8 data bits preceded by a start bit which is low and followed by a stop bit which is high (viewed at the processor or programming connector). The data bits are high for 1 and low for 0.

The cold boot mode and the triplets sent are described in [Section 4.1 on page 20](#). Each triplet consists of a 2-byte address and a 1-byte data value. The data value is stored in the address specified. The uppermost bit of the 16-bit address is set to one to specify an internal I/O write. The remaining 15 bits specify the address. If the write is to memory then the uppermost bit must be zero and the write must be to the first 32 KB of the memory space.

The user should see the 9 bytes transmitted at 2400 bps or 416 μ s per bit. The status bit will initially toggle fairly rapidly during the transmission of the first triplet because the default setting of the status bit is to go low on the first byte of an opcode fetch. While the triplets are being read, instructions are being executed from the small cold boot program within the microprocessor. The status line will go low after the first triplet has been read. It will go high after the second triplet is read and return to low after the third triplet is read. The status line will stay low until the sequence starts again.

If this test fails to function it may be that the programming connector is connected improperly or the proper pull-up resistors are not installed on the SMODE lines. Other possibilities are that one of the oscillators is not working or is operating at the wrong frequency, or the reset could be failing.

11.2.2.1 Using serialIO.exe

This test is available as `StatusTgl.Ddiag`, one of the diagnostic samples downloaded in `ser_io_rab20.zip`.

11.2.3 Diagnostic Test #2

The following program checks the processor/RAM interface for an SRAM device connected to /CS1, /OE1, /WE1. The test toggles the first 16 address lines. All of the data lines must be connected to the SRAM and functioning or the program will not execute correctly.

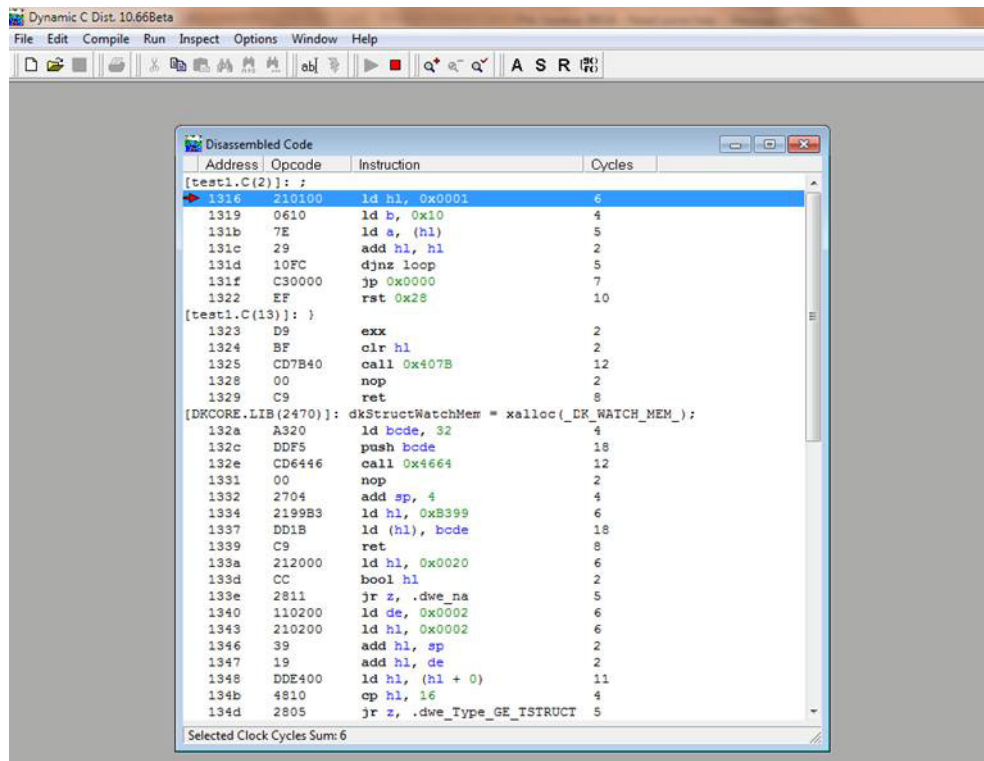
A series of triplets are sent to the Rabbit via one of the bootstrap ports to set up the necessary control registers and write several instructions to RAM. Finally the bootstrap termination code is sent and the program begins executing instructions in RAM starting at address 0x00.

The following steps illustrate one way to create a diagnostic program.

1. Write a test program in assembly:

```
main(){
;
#asm
boot:
  ld hl,1
  ld b,16
loop:
  ld a,(hl)
  add hl,hl ;shift left
  djnz loop ;16 steps
  jp boot ;continue test
#endasm
}
```

2. Compile the program using Dynamic C and open the Assembly window. The disassembled code looks like this:



The screenshot shows the Dynamic C Dist. 10.66Beta interface with the Disassembled Code window open. The window displays the following assembly code:

Address	Opcode	Instruction	Cycles
[test1.C(2)]: ;			
131e	210100	ld hl, 0x0001	6
1319	0610	ld b, 0x10	4
131b	7E	ld a, (hl)	5
131c	29	add hl, hl	2
131d	10FC	djnz loop	5
131f	C30000	jp 0x0000	7
1322	EF	rst 0x28	10
[test1.C(13)]: ;			
1323	D9	exx	2
1324	BF	clr hl	2
1325	CD7B40	call 0x407B	12
1328	00	nop	2
1329	C9	ret	8
[DKCORE.LIB(2470)]: dkStructWatchMem = xalloc(_EK_WATCH_MEM_);			
132a	A320	ld b0de, 32	4
132c	DDF5	push b0de	18
132e	CD6446	call 0x4664	12
1331	00	nop	2
1332	2704	add sp, 4	4
1334	2199B3	ld hl, 0xB399	6
1337	DD1B	ld (hl), b0de	18
1339	C9	ret	8
133a	212000	ld hl, 0x0020	6
133d	CC	bool hl	2
133e	2811	jr z, .dwe_na	5
1340	110200	ld de, 0x0002	6
1343	210200	ld hl, 0x0002	6
1346	39	add hl, sp	2
1347	19	add hl, de	2
1348	DDE400	ld hl, (hl + 0)	11
134b	4810	cp hl, 16	4
134d	2805	jr z, .dwe_Type_GE_ISTRUCT	5

Selected Clock Cycles Sum: 6

3. The opcodes and their data are in the 2nd column of the Assembly window. Since we want each triplet loaded to RAM beginning at address zero, create the following sequence of triplets.

```

;code to be loaded in SRAM
00 00 21
00 01 01
00 02 00
00 03 06
00 04 10
00 05 7E
00 06 29
00 07 10
00 08 FC
00 09 C3
00 0A 00
00 0B 00

```

4. The code to be loaded in SRAM must be flanked by triplets to configure internal peripherals and a triplet to exit the cold boot upon completion.

```

80 14 05 ;MBOCR: Map SRAM on /CS1 /OE1 /WE1 to Bank 0
80 09 51 ;ready watchdog for disable
80 09 54 ;disable watchdog timer
.
. ;code to be loaded in SRAM goes here
.
80 24 80 ;Terminate boot strap, start executing code at address zero

```

The program, `serialIO.exe`, has the ability to automatically increment the address. Instead of typing in all the addresses, you can use some special comments. They are case sensitive and must be at the beginning of the line with no space between the semicolon and the first letter of the special comment.

```

;Address nmmn
;Triplet

```

The first special comment tells the program to start at address nmmn and increment the address for each transmitted data byte. The second special comment disables the automatic address mode and directs the program to send exactly what is in the file. The triplets shown in #3 may be rewritten as:

```

;Address 0000
21 01 00 ;ld hl,1
06 10 ;ld b,16
7E ;ld a,hl
29 ;add hl,hl
10 FC ;djnz loop
C3 00 00 ;jp 0
;Triplet

```

5. The following code is required make diagnostics work for 16-bit data transfers:

```

\\Header Block

```

```

; Insert Rabbit 4000 diagnostic code after this comment.
3E 84; ld a, 0x84

```

```

D3 32 24 00;ioi ld (SPCR), a
; .forever:
D3 3A 30 00;ioi ld a, (PADR)
EE 01;    xor 0x01
D3 32 30 00;ioi ld (PADR), a
01 0C 00;ld bc, 12 (use 24 vs. 12 for clock doubled vs. not doubled)
; .again0:
21 CC FB;ld hl, 64460
; .again1:
2B;      dec hl
B1;      ld de, hl
CC;      bool hl
A1;      ld hl, de
20 FA;   jr nz, .again1
ED 10 F4;dwjnz .again0
18 E5;   jr .forever

```

\\Footer Block

```

; Insert Rabbit 4000 diagnostic code before this comment.
;Triplet
80 1D 00;MACR = 0x00 (set 8-bit operation for both /CS0 and /CS1)
80 14 0D;MB0CR = 0x0D (4WS, write protected, /OE1, /CS1)
80 15 05;MB1CR = 0x05 (4WS, /OE1, /CS1)
80 16 00;MB2CR = 0x00 (4WS, /OE0, /CS0)
80 17 00;MB3CR = 0x00 (4WS, /OE0, /CS0)
80 13 D6;SEGSIZE = 0xD6 (stack @ 0xD000, data @ 0x6000)
80 11 00;STACKSEG = 0x00 (physical stack @ 0x0D000 = 0x00000 + 0xD000)
80 12 00;DATASEG = 0x00 (physical data @ 0x06000 = 0x00000 + 0x6000)
80 24 80;SPCR = 0x80 (terminate bootstrap, start running at address 0)

```

The above mentioned diagnostic code must be used with the following 16-bit header code for 16-bit data transfers:

```

80 0E A0;GOOCR = 0xA0 (set CLK, STATUS outputs low)
80 09 51;WDTTR = 0x51 (prepare to disable the watchdog)
80 09 54;WDTTR = 0x54 (disable the watchdog)
80 00 08;GCSR = 0x08 (CPU = OSC, PCLK = OSC)
80 10 00;MMIDR = 0x00 (8-bit I/O space, shared I&D space, no inversions)
80 16 25;MB2CR = 0x25 (4WS, inverted MSB, /OE1, /CS1)
80 13 D1;SEGSIZE = 0xD1 (stack @ 0xD000, data @ 0x1000)
80 11 80;STACKSEG = 0x80 (physical stack @ 0x8D000 = 0x80000 + 0xD000)
80 12 7F;DATASEG = 0x7F (physical data @ 0x80000 = 0x7F000 + 0x1000)

```

```

80 73 01;PEAHR = 0x01 (preset PE4 as /A0)
80 75 10;PEFR = 0x10 (PE4 is alternate output)
80 77 10;PEDDR = 0x10 (set PE4 as output)
80 1D 20;MACR = 0x20 (set basic 16-bit operation for /CS1)
80 C4 00;SACR = 0x00 (use pport C for Rx, 8-bit async mode, IRQ off)
80 A0 01;TACSR = 0x01 (enable timer A main clock)
;Address 1000
21 21 00;ld hl, 0x0021
00;      nop
2B;      dec hl
2B;      dec hl
2B;      dec hl
2B;      dec hl (HL = 0x001D, i.e. MACR)
1E 1E;   ld e, 0x1E
1C;      inc e
1C;      inc e (E = 0x20)
7B;      ld a, e
7B;      ld a, e (A = 0x20)
D3;      ioi
D3 77;   ioi ld (hl), a (enable basic 16-bit operation on /CS1)
77;      ld (hl), a (harmless write into write-protected MB0CR quadrant)
00;      nop
00;      nop (allow time for 16-bit memory bus to start up)
3E 24;   ld a, 0x24
D3 32 1D 00;ioi ld (MACR), a (set basic 16-bit operation for /CS0 and /CS1)
3E 05;   ld a, 0x05
D3 32 14 00;ioi ld (MB0CR), a (4WS, /OE1, /CS1)
;3E 40;   ld a, 0x00
;D3 32 16 00;ioi ld (MB2CR), a (4WS, /OE0, /CS0)
3E 80;   ld a, 0x80
D3 32 10 00;ioi ld (MMIDR), a (enable 16-bit I/O space)
3E C0;   ld a, 0xC0
D3 32 20 04;ioi ld (EDMR), a (enable R4000 instructions)
31 00 E0;ld sp, 0xE000
; Insert Rabbit 4000 diagnostic code after this comment.

```


Appendix A: Supported Rabbit 4000 Baud Rates

This table contains divisors to put into TATxR registers. All frequencies that allow 57600 baud up to 30MHz are shown (as well as a few higher frequencies). All of the divisors listed here were calculated with the default equation given on the next page.

Crystal Freq. (MHz)	2400 baud	9600 baud	19200 baud	57600 baud	115200 baud	230400 baud	460800 baud
1.8432	23	5	2	0	a	-	-
3.6864	47	11	5	1	0	-	-
5.5296	71	17	8	2	-	-	-
7.3728	95	23	11	3	1	0	-
9.2160	119	29	14	4	-	-	-
11.0592	143	35	17	5	2	-	-
12.9024	167	41	20	6	-	-	-
14.7456	191	47	23	7	3	1	0
16.5888	215	53	26	8	-	-	-
18.4320	239	59	29	9	4	-	-
20.2752	b	65	32	10	-	-	-
22.1184	*	71	35	11	5	2	-
23.9616	*	77	38	12	-	-	-
25.8048	*	83	41	13	6	-	-
27.6480	*	89	44	14	-	-	-
29.4912	*	95	47	15	7	3	1
36.8640	*	119	59	19	9	4	-
44.2368	*	143	71	23	11	5	2
51.6096	*	167	83	27	13	6	-
58.9824	*	191	95	31	15	7	3

a. Baud rate is not available at given frequency.

b. Baud rate is available with further BIOS modification.

The default equation for the divisor is:

$$\text{divisor} = \frac{\text{CPU frequency in Hz}}{32 \times \text{baud rate}} - 1$$

If the divisor is not an integer value, that baud rate is not available for that frequency (identified by a “-” in the table). If the divisor is above 255, that baud rate is not available without further BIOS modification (identified by a “*” in the table). To allow that baud rate, you need to clock the desired serial port via timer A1 (by default they run off the peripheral clock / 2), then scale down timer A to make the serial port divisor fall below 256.

Timer A can be clocked by the peripheral clock (**PCLK**) in addition to the default, which is the peripheral clock/2 (**PCLK/2**). Furthermore, the asynchronous serial port data rate can be 8x the clock in addition to the default of 16x the clock. Therefore, in addition to the equation above, the following equations may be used to find the asynchronous divisor for a given clock frequency.

Timer A clocked by **PCLK/2**, serial data rate = 16 x clock

$$\text{divisor} = \frac{\text{CPU frequency in Hz}}{16 \times 2 \times \text{baud rate}} - 1$$

Timer A clocked by **PCLK**, serial data rate = 16 x clock:

$$\text{divisor} = \frac{\text{CPU frequency in Hz}}{16 \times \text{baud rate}} - 1$$

Timer A clocked by **PCLK/2**, serial data rate = 8 x clock:

$$\text{divisor} = \frac{\text{CPU frequency in Hz}}{8 \times 2 \times \text{baud rate}} - 1$$

Timer A clocked by **PCLK**, serial data rate = 8 x clock:

$$\text{divisor} = \frac{\text{CPU frequency in Hz}}{8 \times \text{baud rate}} - 1$$

INDEX

A

A18 and A19 inversion 43
access times 11, 15, 16, 38, 41, 61, 62

B

bank size 38
base segment 38
baud rates 7, 9, 10, 42
 divisor 21, 90
 sleepy mode 78
bbramorg 48
binary compatibility 64
BIOS 37
 conditional compilation 40
 flowchart 39
 modifying 40
 wait loop 14
board type 40
boot blocks 63
boot ROM 20

C

calibration constants 57
capacitance 16, 73
ceramic resonator 10
chip select 9
 self-timed mode 74
 short mode 74
CLK (pin 2) 81
clock input 77
CLOCK_DOUBLED 41
clocks 7, 13, 17, 74
 common crystal frequencies 9
 speed 11, 16, 41
cloning 9, 10, 41, 69
CMOS 73, 76
cold boot 19, 20, 82
conformal coating 77
crystal 9
crystal oscillator 9
 32 kHz crystal oscillator external logic 77
CS1 38, 41
CS1_ALWAYS_ON 15, 41

D

DATAORG 41
DATASEG register 22, 38
debug mode 22, 76
design conventions 9
 memory chips 10
 oscillator crystals 10
 programming cable connector 10
DHCP_CLIENT_ID_MAC 43
diagnostic tests 81
DTR line 21
Dynamic C start sequence 21
Dynamic C version 40

E

EMI 17
ENABLE_CLONING 41
ENABLE_SPREADER 42
ESD 11

F

fastramorg 48
FETs 14, 73
finite state machine 22
firmware 19
flash
 custom driver 80
 supported devices 79
 write method 22
FLASH_SIZE 42
flashorg 48
Fletcher algorithm 22
floating inputs 14

H

hardware reset 10

I

interrupts 22, 77

M

MAC address 59, 62, 71

macros, defined internally	
_SEPARATE_INST_DATA_	40
_BOARD_TYPE_	40
_CPU_ID_	40
FLASH	40
_FLASH_SIZE_	40
RAM	40
_RAM_SIZE_	40
CC_VER	40
MAX_USERBLOCK_SIZE	64
MBOCR_INVRT_A18	43
MBOCR_INVRT_A19	43
MBxCR	38
MECR	38
memory	
access time	16
bank control registers	38
data segment logical address	41
flash available	42, 64
line permutation	17
organization	23
RAM available	42
segment locations	38
MMIDR register	38
MMU/MIU	21
N	
NUM_FLASH_WAITST	42
NUM_RAM_WAITST	42
O	
operating voltages	11, 75
org	48
origin directives	43
in user code	54
oscillator	9, 10, 13, 81
output enable	9
P	
periodic interrupt	76, 77
power consumption	11, 73
programming cable	8, 9, 19, 21, 22, 82
programming cable connector	10
Q	
quadrant size	38
R	
RAM	
access time	41
wrap-around test	22
RAM_SIZE	42
rcodorg	48
rconorg	48
Realtek Ethernet chip	78
reset	22, 82
rvarorg	48
S	
SEGSIZE register	38
serial port A	9
sleepy mode	
enter and exit	77
interrupts	77
SMODE pins	21, 82
SP register	38
STACKSEG register	38
static variables	34
surface-mount	12
System ID block	40, 57
reading	60
sizes of	60
writing	63
T	
target communications protocol	22
through-hole	12
triplets	21, 83
troubleshooting tips	81
U	
USE_TIMER_PRESCALE	42
User block	57
reading	65
sizes of	60
writing	67
V	
variables	
static	34
W	
wait states	16, 21, 42
watch expressions	42
WATCHCODESIZE	42
wcodorg	48
write enable	9
write method	22
X	
xcodorg	48
xconorg	48
xmemorg	48
xvarorg	48, 50

Z

ZERO_STATIC_DATA	34
------------------------	----

