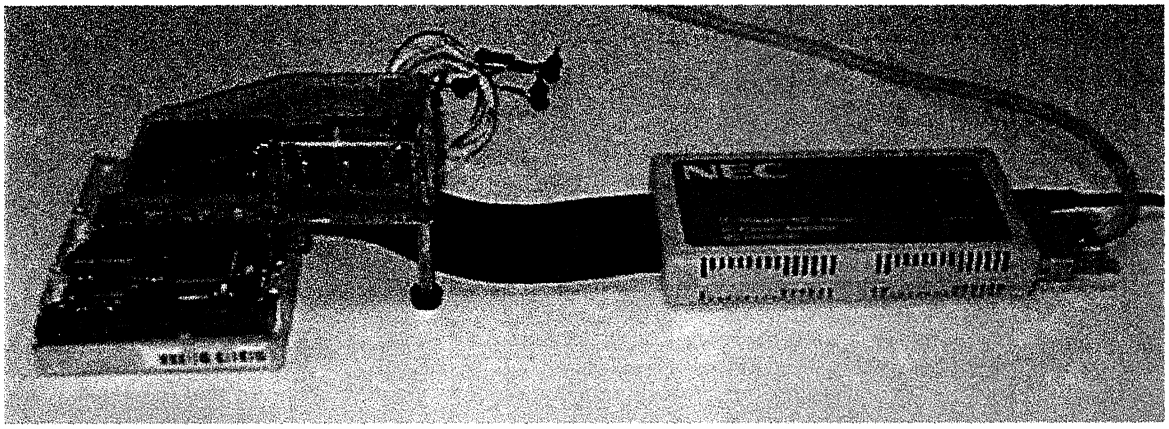
DPSG V85X User Guide



December 1997

Rev. D

Confidential

THIS QUANTUM CONFIDENTIAL DOCUMENT IS FOR INTERNAL USE ONLY. IF IT IS DISPOSED OF, IT SHOULD BE DESTROYED BY SHREDDING.

Rev. D History

This revision of the DPSG V85x User Guide contain the following changes to its initial release versions (Rev. A, B & C) :

1. re-edited sections to the Table of Contents and Chapter 7
2. revised Chapter 6
3. a new chapter about the Eclipse Hardware and Firmware Memory Map
4. a new chapter about the System Cylinder
5. new instructions to the Software Tools Installation, which include instructions for Windows NT 4.0 version

Acknowledgments

This user guide is a product of a consolidated effort contributed by the members of the Eclipse and Tsunami teams, APE, and DPSG Engineering Operations. Special acknowledgment must be made to:

- Joe Cusimano, who wrote chapters 3, 7, and 8
- John Masles, who wrote chapters 5, 6, and 9
- Bruce Peterson, who wrote chapter 2 and edited the contents of this user manual
- Zachary Toh, Frank Inzerillo, John Lauber, and Joe Liu, who also edited the contents of this user manual
- Wilfredo Morales and Lynh Dang, who wrote chapter 4 and edited the production of this user manual
- Bob Condie and Natalie McKinsey, who coordinated this documentation project

A hypertext version of this user guide viewable through Netscape will also be available through:

- DPSG DEVELOPMENT ENGINEERING\ENGR OPS Web server
- ECLIPSEFW Web server

TABLE OF CONTENTS

TABLE OF CONTENTS

TABLE OF CONTENTS

Chapter 5 Codewright

Chapter 6 Make

TABLE OF CONTENTS

TABLE OF CONTENTS

TABLE OF CONTENTS

TABLE OF CONTENTS

Chapter 1 Introduction

The DPSG V85x User Guide is written for Quantum firmware engineers, who will develop firmware for drives using NEC V85x microprocessors. This book not only covers the installation the of NEC Development Tools and the use of Codewright, the Makefile, and the V851 In-Circuit Emulator; it also provides you with vital background information on the NEC microprocessor, the Multi Stack Kernel, Coding in C, and the Eclipse Code.

Because the Eclipse code is largely written in C and the programmer is mostly insulated from the underlying hardware, it is extremely important for all programmers working with this code to understand the impact of certain things on code size. These include but are not limited to:

- the V851 architecture and instruction set, particularly as it relates to the loading of bytes and halfwords
- the compiler's allocation of local variables in registers

Without this knowledge, it is almost certain that poor, inefficient code will be written. Just because we have moved to a radically new architecture does not mean that many of the old limitations no longer apply. RAM and ROM will continue to be precious commodities and we must be sure not to waste them frivolously. Please take the time to read the 'Fundamentals' portions of this document so that you will understand how to write good, efficient code.

The following list outlines the materials covered by the 'Fundamentals'.

(1) C coding guidelines to help you understand how to write good efficient 'C' code. These contain:

- 'C' Coding Conventions
- 'C' Coding Safety Practices
- 'C' Coding Efficiencies

Applying the recommendations suggested in these sections will help you develop a better coding style for creating the most efficient object code possible.

(2) The NEC V851 Microprocessor's Architecture. This section lists essential information about:

- Instruction Set Size and Latency
- V851 Registers-Eclipse Configuration
- Data Formats
- Saturated Math Instructions
- Hardware Multiply and Divide Instructions
- Execution Speed
- Pipeline Stalls

Read this section to learn how your data will be affected by this architecture.

(3) The Multi-Stack Kernel. This section lists the advantages and disadvantages of the Kernel; defines task states, tasks in the Eclipse code, and the Kernel's RAM structures; and illustrates the Kernel's three basic components:

- An Interrupt Service Routine (ISR) handler
- A Scheduler
- A set of functions for pending and posting

(This page was intentionally left blank.)

Chapter 2 Historical Background

In 1992, the APE department determined that the current code base and microprocessor architectures were nearing the end of their life spans and embarked on a radical project whose goals were to:

- develop a new base of drive firmware written in C
- select a new high speed DSP or RISC microprocessor whose performance would be so great as to more than offset any inefficiencies inherent in C
- select new state of the art Windows-based development and debug tools offering far more power and ease of use than previous tools

There were compelling hardware and firmware reasons for wanting new code to be in C rather than assembly language. The hardware reason was that it appeared likely that in the future it might be necessary or desirable to be able to port the code to new and different processors without requiring a total code rewrite. It was hoped that using C would provide a relatively high degree of hardware independence. The firmware reason was that C code should be easier to write and maintain than assembly code. As the complexity of the drive firmware increases, anything that aids in the development and maintenance process is desirable.

Although the driving force behind the project was the development of C code, the first step was choosing a new processor. As it turned out, "selecting" a new off the shelf processor was not really possible. No standard processor had just the right mix of high performance and low cost that was necessary for Quantum's purposes. After much study and analysis, the decision was made to work with NEC in the development of a new RISC microprocessor leveraged from an existing NEC RISC processor. APE worked closely with NEC to insure that architectural features and instructions critical to hard disk drive firmware were included in the new Atlas V851 processor, and that unnecessary features were trimmed out to keep cost down. Some of the key added features were a high speed multiply and saturated math instructions for use by the servo, and a large number of 2-byte instructions for code density.

While NEC was developing the V851, APE began development of the C firmware which was code-named Calvin. At the outset of the project, there was only 1 Kbyte of internal RAM in the V851. For many years, it had been deemed desirable to use a multi-stack kernel, but the 1 Kbyte or high speed RAM in the V851 was too small for that, so the first code that was developed had a cooperative multitasking kernel. The first offshoot of this code that has made it into production is the Viking code.

By the time the Eclipse project adopted the Calvin code, the processor that was available had 2 Kbytes of internal RAM. APE (by then renamed T&E) realized that the 2 Kbytes of high speed RAM was probably a bare minimum necessary to support a multi-stack kernel. So, concurrent with Eclipse code development, T&E embarked on the development of a multi-stack kernel. After a two-week trial period with the new kernel, the Eclipse team made the decision to adopt the new kernel and merged it into the Eclipse code.

Once NEC was chosen as the processor vendor, APE also worked closely with NEC in the area of development tools and emulator hardware and software. In the past, NEC emulators were usually fairly good – a bit clunky perhaps but with most of the necessary features. Their emulator software was another matter altogether. It was ancient command-line technology and generally fairly difficult to use. APE had little confidence that NEC would be able to develop adequate much less state of the art GUI software. Therefore APE pushed NEC to choose a third party vendor who could provide a total package of integrated software tools from compiler through linker to debugger. NEC ended up choosing Green Hills, a software company whose Multi development environment was available for a number of RISC processors. As with the processor, "selecting" a vendor was not enough. APE had to continually work with NEC and Green Hills to insure that all the necessary tools were available and that they worked correctly and optimally. In addition, APE chose Codewright as its editor of choice because of its backward compatibility with Brief and its many enhanced features. The end result is a development environment that is still evolving, but which is a vast improvement over the past.

Chapter 3 Fundamentals

Read This First!

We have an opportunity to produce drive products with better features faster than anyone else out there if we use these powerful new tools right. We also have an opportunity to make our projects late and full of bugs.

We're not trying to imply that your project will take on excessive risk by using the V851 and the 'C' code architecture. What we're saying is that this is a *change* in the way you are accustomed to developing firmware. These are the changes and the challenges:

You Can Debug in 'C' Faster
The debugger (see 'Using the V851 In-Circuit Emulator') is much more powerful than anything you've had before. You will be viewing your code symbolically (without sacrificing any of the nitty-gritty bit level and assembly, traces, and stuff) and stepping through reams of code with each click of the mouse.

Debugging symbolically in 'C' allows you to hack through the logic of your code in order to get to the actual nitty-gritty bug you are looking for without having to slog through hundreds of lines of assembly.

Read 'Calvin Coding Practices! '

You Can Really Mess Up When You Write in 'C'

The problem is that your style may not create the most efficient object code. **Your highest priority is to create the most efficient object code possible.**

Your other priority is to create code that other people can read. Just imagine a whole project, maybe 10 firmware engineers, each writing in his own style. Now imagine the same code getting handed from project to project, going through literally hundreds of firmware authors. **It's not a particular style that makes 'C' code readable, it's *consistency* of style.**

We have produced a document that outlines standards for coding style that most folks have agreed on. Please follow the 'C' coding conventions that we have outlined, even if you don't agree with some of the rules. It will produce code that everyone can read.

'C' Is Not a "Self-Documenting Language"

Please lace your code with lots of comments describing the meanings of your algorithms, what variables are used for, why algorithms do what they do, and where to go look for important information about functions and data.

The names of your variables and functions may be obvious to you but they are not to other engineers. **Do not name your variables and functions so that they are easier to type.** Name them so that they are easier to read. They will be read much more often than they are typed.

For some reason I have seen less comments put into 'C' code than assembly code. I have also seen projects stop so that firmware engineers would go back and type comments into their code. Don't let this happen to you.

Read 'C Coding Safety Practices!'

The Compiler Can Create Some Pretty Mysterious Bugs for You

The compiler was written to generate very efficient code for the V851, which is a 'signed' processor. Because of this, you can make some coding mistakes that don't look like mistakes. The most common problem is that the compiler 'optimizes out' some of your code; in other words, it disappears (you give it a condition that could not possibly exist, so it quietly makes your object code go away). Another problem is that you don't get the math or control flow results you expect.

If you learn the quirks that make this happen up front, you will save yourself a lot of grief later.

Read 'C Coding Efficiency!'

Writing 'C' for Firmware is Different

If you write your 'C' code as if it were going to run in a computer it will be way too inefficient for disk drive firmware. We don't get to use all the features of the 'C' language because of this. 'C' code written for efficient firmware can also be harder to read than code written for a computer.

Even with these two tradeoffs, the efficient 'C' firmware is still much faster to write and debug and easier to read than assembly language.

Learn 'Efficiency' at the Same Time as You Learn 'C'

The most common argument that I hear from developers when changing from assembly to 'C' is that it is difficult enough to learn 'C' without also having to learn to write it efficiently. **This is not true.**

If you take this attitude then you will be forced to re-write your code late in your project. There is no time set aside in your schedule for this, so you will be late and stressed.

If you learn to write your code efficiently the first time then it will save you (and your project) a great deal of problems.

Think about it. If you learn to write efficient code from the start, it will take some extra work up front. However, as you become more familiar with this process it will become much easier. The code you develop in the last two-thirds of your project will take very little additional time to make it efficient.

If you do not learn efficient coding up front, but put it off until the end of your project, then you'll have to go back over all your code (or someone else will, and they won't like you). The total amount of work you'll have to do is more.

Examine Your Listing Files

The 'C' coding efficiency document provides general rules for keeping your code size down. However, because of differences in style the only way to really ensure that you are generating a minimum amount of assembly output is to read the assembly listing outputs of the compiler when your code is built.

This is a must! You cannot learn to develop efficient code any other way.

Managers, Allow Your Engineers to Do This

It is of no use for firmware engineers to commit themselves to developing efficient, maintainable code if managers don't encourage and provide the resources to accomplish it. If you are not committed to the above principles it would be better to delete this section from the documentation. Here are some suggestions regarding how to support the development of efficient, maintainable 'C' code in your group:

- **Express your commitment to the development of efficient, maintainable code 'up front'. Verbally announce this commitment to your team. Get team buy-in.**

- **Forego immediate executable results long enough to allow enough time for staff to get through the initial learning curve. Give your team the resources they need to accomplish this goal.**

- **Seek out team members that need help getting up to speed. Don't assume that everybody will make the transition. Some people in each team always have problems.**

- **Enforce the requirement for *efficient code, adherence to coding conventions,* and *well commented code.* One strategy is to have team members check each other's work.**

- **Know that, although extra effort must be expended at the beginning of your project to transition to 'C', the overall project time will be less than an assembly language project. This will only be true if the above advice is taken, however.**

3.1 C Language - Calvin Coding Practices

Coding practices are defined for three purposes; maintainability, robustness, and efficiency. This chapter is broken into three sections according to these purposes.

Calvin 'C' coding conventions support maintainability and are **mandatory**. The 'C' language invites a wide range of coding styles. The Calvin coding conventions make the code easier to understand by limiting variations in style.

Practices for robustness and efficiency have developed through an understanding of the marriage between compiler and microprocessor target. **These are strongly recommended.** However, you may find exceptions to these rules that are necessary. Do not take these recommendations lightly. If you fail to observe these practices you will create bugs that are very difficult to detect.

For clarity, a glossary is included at the end of this chapter. However, it will not define terms that are un-ambiguously described in ANSI or other standard 'C' publications. **Before starting this document** look up the descriptions of 'Declarations' and 'Definitions'! This is important to many of the standards below.

3.1.1 'C' Coding Conventions

3.1.1.1 Indenting and Bracing
Indents are 4 spaces each. **There shall be no tabs in any code files. Braces shall occupy separate lines.** The 'else' keyword can occupy either a separate line or the same line as its preceding (closing) brace. Indenting and bracing rules apply to all expressions in which they are used including structure declarations and data initialization.

```
if (expression)
{
  statement;
}else
{
  statement;
}

alternatively...

if (expression)
{
  statement;
}
else
{
  statement;
}
```

3.1.1.2 Naming

Prefixes

Prefixes are single lower-case letters that indicate the storage class of their associated variables. Prefixes can be paired and even tripled when appropriate. Prefixes are applied only to data definitions.

```
uUnionName
sStructureName
pPointerName
eEnumeratedType
psPointerToAStructure
puPointerToAUnion
pePointerToAnEnumeratedType
ppPointerToAPointerToAVariable
ppsPointerToAPointerToAStructure
```

Use of Underscores

Underscores ('_') are allowed **only** when imbedded in symbolic constants.

Symbolic Constants

Symbolic constants shall be used in all expressions requiring values except where the acceptability of a hard-coded value is obvious. Symbolic constants will contain all uppercase letters and numbers, and can contain imbedded underscores ('_'). They shall not be preceded by underscores.

```
CONSTANT
MULTI_WORD_CONSTANT

NOT Allowed...

_CONSTANT
```

Portable Storage Types

For code portability, the elementary data types have been redefined. All data declarations shall use these redefinitions. They shall **not** use the ordinary 'C' language types.

```
signed char byte
unsigned char ubyte
signed short hword
unsigned short uhword
signed long word
unsigned long uword
```

Ordinary Variables

Ordinary variables include bytes, halfwords, and words, as well as associated arrays. They shall be named such that the first word is all lower case letters and/or numbers and the remaining words are capitalized.

```
byte    ordinaryVariable;
ubyte   ordinaryVariable2;
hword   tnaCount;
```

Structure, Union, and Enumerated Type Declarations

Structure, union, and enumerated type declarations are named in the same manner as ordinary variables. Note that prefixes are not applied here.

```
struct exampleStructure
{
     byte ordinaryVariable;
};

enum exampleEnumeratedType
{
     FIRST_VALUE,
     SECOND_VALUE
};
```

Structure, union, and enumerated type definitions are prefixed according to the rules for 'prefixes'. Since the prefix forms the initial lower case part of the name, all of the words in the name are capitalized.

```
struct exampleStructure              sDefinedStructure;
struct exampleStructure   *psPointerToDefinedStructure;
enum exampleEnumeratedType eDefinedEnum;
```

Unions for Multiple Field Type Access

Unions are commonly used for code efficiency for bit, byte, half word, and word access to the same memory location. Typically, the location breaks down to bit fields (such as in the case of registers) which are grouped for 8, 16, or 32 bit access. The following names will be given to union members which are used for those accesses; 8 bits = 'all8', 16 bits = 'all16', and 32 bits = 'all32'. The structure declaring the bit fields shall be named 'bit'.

```
union byteAndBitsAccess
{
     byte all8;
     struct
     {
            ubyte bitZeroExample : 1;
            ubyte bitOneExample  : 1;
            ubyte bitTwoExample  : 1;
     }bit;
};
```

ASIC, Processor, and Other Registers

When a structure member or variable name defines an ASIC, processor, or other hardware register, it shall be named with all upper case letters which **appear just as the register does in its hardware manual**. However, certain exceptions are required for registers. For coding efficiency, unions are often used to allow either byte or halfword access to the same registers. A union member that defines a register pair which, in the hardware manual was named with LO/HI suffixes (or some similar naming method), the register pair's union is named without the suffix. If two registers to be paired have dissimilar names, the name of the register with the lowest address shall be used. The halfword member of the union is named 'all16', the register pair structure is named 'reg', and the byte members within the structure shall be named 'lo 'and 'hi' respectively.

```
ubyte SECTROLLO;
ubyte SECTROLHI;

or

union
{
     uhword all16;
     struct
     {
             ubyte lo;
             ubyte hi;
     }reg;
}SECTROL;
```

Typedefs

Typedefs are used to define types that are not built into the language. The type name follows the format for an ordinary variable. The type tag is the type name preceded by a "t". The tag is needed *only* when it is necessary to refer to the type within the typedef itself. When the type name is used in a data definition, the name for the data item follows the rules for the type of data that it is. For example, if the typedef is for a structure, then a definition of a data item of this type will be preceded by an "s".

```
typedef  struct tListEntry
{
     byte info;
     tListEntry *pNext;
     tListEntry *pPrev;
}listEntry;

listEntry sMyList[10];

typedef  struct
{
     byte hour;
     byte minute;
     byte second;
}dayTime;

dayTime sMyTime;
```

Functions

Functions shall be named such that the first letter for all words in the name is capitalized.

```
SomeFunction ();
```

3.1.1.3 Functions

Function Definitions

Function definitions shall contain all the variable names in their parameter lists within the list itself. 'Void' shall be used if there is no return value and/or there is no parameter.

```
word SomeFunction (word inputWord, byte inputByte)
{
     variable1 =  variable2;
     return (word);
}
void OtherFunction (void)
{
     variableA  = variableB;
}
```

Function Declarations (Prototypes)

Function declarations shall contain parameter lists identical to the functions they describe. All functions shall be declared (prototype).

```
word SomeFunction (word inputWord, byte inputByte);
void OtherFunction (void);
```

3.1.1.4 External Declarations

External Data Declarations

Global data shall be declared external in a header file named after the module (i.e., asic.h & asic.c) that defines them. Local data shall not be declared in header files. Symbolic constants used to reference global data shall be coded in the header file in which they are declared.

```
asic.h...

extern byte formatterStatus;

#define STATUS_BUSY 0x01

asic.c...

byte formatterStatus;
```

External Function Declarations

Global functions are prototyped in header files and local functions are prototyped in the module in which they occur.

```
somefunc.h...

extern void SomeFunction(word inputWord,
byte inputByte);

somefunc.c...

void SomeFunction(word inputWord, byte inputByte)
{
}

/* A function local to somefunc.c */

void OtherFunction(void);

void OtherFunction (void)
{
     variablea = variableb;
}
```

3.1.1.5 Function Headers

All functions shall be preceeded with a header of the following format:

```
/********************************************************************************
 * FUNCTION NAME: Actual Function Name
 *
 * DESCRIPTION:  Brief description of the function's purpose. Brief description of how it
 *               accomplishes its purpose. Cautionary notes regarding its use and modification.
 *               Charts, Truth tables, and Other graphic descriptions of important input or
 *               output structures, etc.
 *
 * PARAMETERS: parameterName --  Description of each parameter with its name as coded
 *                               in the function declaration.
 *
 * RETURNS: Description of the value returned, if any.
 *
 * VARIABLES AFFECTED:  Description of important variables changed by the function and how
 *                      they are changed.
 *
 * MAJOR CHANGE HISTORY:
 *
 * DATE          AUTHOR         DESCRIPTION
 * ------------- -------------- ------------------------
 * mm/dd/yy      Your Name      Changes You Made
 *
 ********************************************************************************/
```

Note: 'Your Name' is your full name. This should be enough information for someone to find you in the company phone book for questions. 'Changes You Made' includes initial release.

Note: If you use the 'Codewright' editor, custom buttons are provided which allow insertion of this and file headers.

3.1.1.6 File Headers

All module files shall contain a header of the following format:

```
/*******************************************************************************
 *
 * FILE NAME:  Actual File name
 *
 * PVCS: $Header:  Required for PVCS version control.
 *
 * CONTENTS:  Description of source code file.
 *
 * HOW TO USE FUNCTIONS IN THIS MODULE:

 *       Description of what the functions in this module have in common.
 *       Brief description of each major function within the module (not
 *       necessarily every function). Brief overview of what functions call
 *       these functions and WHY.
 *
 * REVISION HISTORY:
 *
 * PVCS: $Log: List of all checkins from PVCS version control.
 *
 *******************************************************************************/
```

3.1.1.7 File Names

filename.*
A 'RAM' code, header, or other type of file. If this is a code or header file then the functions in it will run in RAM and will not be referenced directly by ROM code.

r_filename.*
'r_' indicates that the file is associated with code that runs in ROM. If the file is a header file, then it can contain definitions both that ROM and RAM functions use. However, it cannot contain definitions that are used by RAM functions and not ROM functions. If the file is a code file, it cannot contain direct references to RAM functions or definitions in RAM header files.

a_filename.*
'a_' indicates that the file is associated with code that is specific to AT interface functions.

ra_filename.*
'ra_' indicates ROM AT interface functions.

s_filename.*
's_' indicates that the file is associated with code that is specific to SCSI interface functions.

rs_filename.*
'rs_' indicates ROM SCSI interface functions.

r_coddef.h & r_global.h
'ROM' header files which have no corresponding code file. These are general purpose definition files which are (potentially) used by all code modules regardless of whether they are RAM or ROM. These names are examples based on existing Calvin code.

3.1.2 'C' Coding Safety Practices

Your code could get optimized out by the compiler and you may not know it! Two variables that get the same hex values stored in them may not be equal when compared! You could increment a number and it decrements instead!

All of these things can happen because 'C' is dramatically affected by the data types that you declare for your variables. The Calvin code has used some practices that avoid pitfalls such as these. The result is 'robust' code. Here is how you can have robust code, too.

3.1.2.1 Run 'Lint'

Lint is a program that is much more fussy about your data types than the actual 'C' compiler. It scans your code as the compiler does and outputs warning messages whenever it sees the slightest potential problem. These messages can indicate some real problems that you would have otherwise found during testing (or customer service).

How to Run 'Lint'

You need to get your own copy of Lint. However, you can check it out by running ape's copy as follows (Assuming the ape server is mapped to drive 't:'):

Running 'Lint' from a Command Line

Here is an example command line:

```
t:\ape\calvin\utils\lint\lint -DXXXXX ghs.lnt idless.c  > idless.err
```

-DXXXXX sets the global environment variable which is recognized in the codedef.h file, which sets all the other compiler switches for the source code.

'ghs.lnt' is a file that contains commands for Lint to ignore certain errors. This will be explained in more detail later.

'idless.c' is an example source code file.

Running 'Lint' from Codewright

Using the Project menu's configure option, enter the following into the 'debug' entry (or another, of your choice):

```
ftee t:\ape\calvin\utils\LINT\LINT --DXXXXX ghs.lnt %b%e >%r.err
```

The options are the same as for command line invocation.

'%b%e' is the codewright keyword for the file currently being edited

'> %r.err' causes the error output to go to a file named after the file being edited, suffixed 'err'.

You can then click on the menu bar item 'Window' and select 'Output' to get the errors.

Controlling the Number of Messages

Lint will generate warnings for some things that are intentional on your part. You can turn off these messages in two ways:

The command line allows you to specify a file which contains all the errors that you wish to be ignored. That is the 'ghs.lnt' from above. An example line in 'ghs.lnt' looks like this:

```
        -e46  // Disable Field type should be int
```

Within the actual source code file you can turn off and on the various errors on a case-by-case basis. Special comments are used to do this. They look like this:

```
        /*lint -e46  Disable Field type should be int */
```

In both cases `'Disable Field type should be int'` is just a comment. The file `'ghs.lnt'` contains lots of control stuff, too, so take a look at it.

The Error Marker

Each error line contains an 'overbar' that points to where Lint thinks the actual error is. This is much more informative than the compiler's output.

Have Faith

When you see the output, your first reaction may be to give up. Don't. When you have sorted through one file, you will have learned enough to use this tool constructively and reduce the outputs generated by other files. It may even save you from some real problems in the future.

3.1.2.2 The 'Basic Rule' of the Atlas Processor

Atlas is a **signed processor.** This means that the high order bit of all **words** (32-bits, as stored in registers) is a sign bit. The processor **sign extends** halfwords and bytes when they are loaded into registers, meaning that their high order bits are replicated all the way out to the high order bit of the target register.

For instance, loading a byte that contains '80' hex into a register will result in 'FFFFFF80' hex.

The compiler knows about this and is affected in a lot of ways by this behavior. You must always predict what the compiler will do with your variables based on their sign.

3.1.2.3 Safe Coding Practices

There are always exceptions to what are considered 'safe' practices. However, any departures from them must be clearly documented! The most common conflict with safe practices is the requirement to generate efficient code.

All Bit Fields Should Be Unsigned (Use uword, uhword, or ubyte)

If you declare a bit field to be signed, you lose one of its bits to the sign bit. In the extreme (and this is the most common case), a single bit field that is declared signed can have only two values; zero and -1 (that's **minus one!**). Declaring a bit field signed can cause some of your code to be optimized out as you attempt to assign or compare it with values it cannot have.

Note: This is **never** in conflict with generating efficient code.

```
struct dangerousBitStructure
{
     byte unsafeBit : 1;
     /* This bit is signed.  */
     byte unused     : 7;
};

if ( unsafeBit == 1)
{
     statement;
}

!!!! The above expression will be 'optimized' out!
```

Don't Use Variables in Expressions With More Than Their Maximum Value

Be aware of the actual maximum of signed variables, remembering that you lose the high order bit to the sign bit. See 'Elementary Data Types' in the glossary. When using hex values, you may lose track of the real sign of your variables. This can conflict with efficiency in 'C' coding. Here is the most common trap:

```
byte  asicRegister;      // This is some ASIC register. We declare it
                         // signed so that loads and stores will not
                         // generate extra masking code to strip off
                         // the sign bit. This is OK as long as you
                         // don't break the rules later.

byte  temporaryVariable; // This is a variable we'll use in our
                         // example. It's the same type as the
                         // register, which should be OK.
main()
{
     temporaryVariable = asicRegister; // We safely (and efficiently)
                                       // load all 8 bits of the
                                       // register.
     if (temporaryVariable == 0xff)    // Oops! 0xff is a positive
                                       // 255! The (signed) variable
     {                                 // couldn't possibly contain
                                       // this (its maximum is 127).
            asicRegister = 0;          // Everything within this 'if'
                                       // statement will be optimized
                                       // out!!!
     };
}
```

Be Aware That Hex Values are Always Positive

All hex values are positive. For instance, Hex 80 is decimal +128. However, the same bit value in a signed variable is -128. Don't create confusion or your code may not behave as expected. Note that, in this example, you could have assigned the variable the value 0x80, which would not get optimized out. It would, however, give the variable an actual value of minus 128. This is often done to generate efficient code.

```
word signedVariable;

signedVariable = -128;

if (signedVariable == 0x80)
{
     statement;
};

Oops! The above are NOT equivalent!
Your code will be optimized out!
```

Ensure Order of Execution by Declaring Variables 'Volatile'

The volatile declaration means that the associated variable is an ASIC register or a variable that is changed during interrupts and can't be expected to remain the same within a code fragment. This forces the compiler to order loads and stores of it exactly as you have coded it. If you don't declare ASIC registers and global variables affected by interrupts volatile, then the optimizer may (cleverly) move your assignments around (or optimize them out) within functions and your code will fail.

```
ubyte resetRegister;         // An ASIC register.
volatile ubyte goodRegister; // This is OK

resetRegister = 0x80;        // Reset the ASIC
resetRegister = 0;           // Clear Reset
resetRegister = 0xFF;        // Tri-state the Reset

Oops! The reset register wasn't declared volatile!
The first two assignments will get optimized out
because nothing depended on them.  The ASIC reset
fails.  'Good' register is declared correctly.
```

Use the 'C' to Assembly Language Interface

Calvin has a tool set which converts defined data structures to symbolic constants that the assembly language code can use to reference those structures. This is documented elsewhere in the handbook. If you write assembly language code that uses hard coded constants for data structure access, then if the data structures change in 'C' and the corresponding changes aren't made in the assembly language, your code will crash.

Use the 'C' to Assembly Language Interface's Bit Mask Header Files

As the 'C' to assembly language interface tool generates symbolic constants, it also generates bit masks for all the bit fields in the structures. It does this for both the assembly language and the 'C' code. Using these masks in the 'C' code ensures that the masks remain consistent with the structures just the same as it does for the assembly language.

3.1.3 'C' Coding Efficiency

'C' programming for Calvin requires maximum performance in a minimum of code space. You must pay up-front attention to the way that you write your code and the way the compiler turns it into assembly. We cannot stress this enough. We have seen overhead as low as 6% above assembly language when the 'C' code is written efficiently.

Don't write your code inefficiently at first, assuming you can squeeze it later. Start out with a basic knowledge of practices that are efficient and code accordingly. At first, examine the assembly language results of each function you write until you become familiar with how the compiler treats your style of coding. Adjust your style to produce the most compact code possible. As time goes on, it will be increasingly unnecessary to check your assembly.

3.1.3.1 Priorities

The first priority is generating the minimum number of assembly instructions taking into consideration the actual number of bytes of those instructions. Execution time is the next lower priority. Portability is last on the list of priorities.

3.1.3.2 Tradeoffs

Readability

Some level of readability is lost when programming for maximum efficiency. Still, the resulting 'C' will be much more readable than assembly language. Just remember that the most important readability tool is good **comments!**

Portability

Portability is also reduced somewhat as there is a direct relationship between the microprocessor's language and the 'C' code that produces it. However, 'C' is dramatically more portable than assembly language, no matter how it is written.

Debug Display

The optimization process done after the compiler generates assembly code ties assembly lines to their respective 'C' lines, but they tend to drift away as optimization combines them together and eliminates some.

When debugging in 'C' and displaying the assembly language, you may then notice that, even though you are stopped on a subroutine call that uses a variable, the variable does not yet contain what you think it should. This may be because the assembly code that initializes the variable has not yet been reached. This can cause confusion in debugging, which once understood, is tolerable.

3.1.3.3 Basic Rules of Behavior

Check Assembly Output

Examine the assembly output of the compiler as you develop your code. This step cannot be omitted or left out of your schedule. Fortunately, as you become familiar with the compiler and efficient coding techniques, this will be required less often.

Understand the Microprocessor

You need to know at least as much about the processor as an assembly programmer because you are constantly optimizing for the processor, trying to predict what the compiler is going to do with your 'C' code.

Design Stingy Algorithms

Sometimes, when writing in 'C', it's tempting to design 'extra flexible algorithms' or other cool stuff just because of the language before you. As a result, you may look at the individual lines of code you've created and see that they themselves are as code efficient as possible, yet you've just designed an algorithm that, in itself, is very costly. If you wouldn't design it in assembly, don't design it in 'C'.

Watch for Library Routines

Library routines linked into your code package can add invisible overhead. Library routines can be re-coded and replaced with custom functions at the end of the project.

Don't Duplicate Code

Code duplication is more common and harder to recognize than you think. If you're an assembly programmer you may be in the habit of in-lining variable stores and other code fragments due to their efficiency. However, in 'C', this could be costly due to register set-ups and other overhead factors. In 'C', it is also tempting to define multiple subroutines that do nearly the same thing. One way to see if you are duplicating code is to 'grep' the occurrences of your variables. If a variable appears in a lot of places then chances are you're duplicating code.

Plan for the Future

Plan each algorithm in terms of what happens when its data structures change. For instance, if you design an array of structures that starts out with a single byte per structure then you can write algorithms that access the elements with array subscripting variables (rather than pointers) and that won't cost you too much. However, if someone adds two bytes to the structure then the subroutine that you have written will re-compile with a complicated multiply algorithm to access each of the array elements and no one will know it until someone notices that the code doesn't fit into ROM anymore.

Don't let the code deteriorate

The degree of deterioration that occurs as a result of maintenance engineer's uncertainty determines the useful lifespan of the code. For this reason, the degree of maintainability in the initial release of your code determines the lifespan of the code package. Also, maintenance engineers must be aware of all the code efficiency requirements in order to minimize this deterioration.

3.1.3.4 Coding Techniques

The following coding techniques are outlined in no particular order.

Declare Variables Signed Whenever Possible

This is crucial! ATLAS is a signed processor. If you declare a byte or halfword as unsigned, the compiler will generate code to mask off the (potential) sign-extended bits every time it is loaded from memory. The potential amount of excess code that can get generated by variables declared unsigned is huge. Declaring the variable signed will prevent this.

However, there is a tradeoff. A signed byte has only seven useful bits for math and compares (unless you are really interested in its sign). A signed halfword has only 15. So the general rule is to declare variables based on its real range of values. See 'Elementary Data Types' in the glossary for those ranges. But this is only important if math or compares are done on the variable. See 'Safe Coding Practices' for some examples of math that can fail with signed variables.

Often, you can get away with declaring a byte or halfword signed, even though you need all 8 bits of its value. If you will never compare or do math on the contents then this is true. This is most often the case with ASIC registers. Here are the things you can and cannot do with signed bytes and halfwords:

Can Do:

Assign an 8-bit value to a signed byte:

```
byte A;
A = 0xff;
```

A is really (-1). However, you cannot compare it to 0xff later because 0xff is really +255.

Store a signed byte as an 8-bit value.
All 8 bits will be stored in memory.

Retrieve a signed byte as an 8-bit value.
All 8 bits will be retrieved from memory. No extra code will be generated.

Can't Do:

Compare an 8-bit value to a signed byte:

```
byte A;
A = 0xff;
if (A == 0xff)
{
     statement;
}
```

The if-statement will fail because A is really -1, not +255
(the real value of 0xff)

Declare Local Variables as Words (Do for all locals!)

Do not declare local as bytes or half-words. If you do this, the compiler will generate extra assembly code to truncate the counter values to bytes or half-words each time you do math on them, even if your algorithm would never generate values greater than the byte or half-word in size.

Can Do:

Use a word as a loop counter:

```
word    workWord = 1;

do
{
     workWord++;
}while (workWord < 100);
```

Can't Do:

Don't use bytes or half words as counters:

```
byte    workByte = 1;

do
{
     workByte++;
}while(workByte < 100);
```

Do Not Define Large Data Structures or Arrays Locally

Large data structures and arrays, when declared local to functions, use up stack space. Do not do this. Instead, define them globally. They will go into external DRAM.

You may want to consider developing some memory manager code if you need to temporarily allocate DRAM for functions to use.

Reducing Compound Logical if-expressions

Use DeMorgan equivalents to reduce the complexity of logical expressions. For example:

Wrong:

```
if(!A||(A&&!B))
```

Right:

```
if(!A||!B)
```

Bitwise OR boolean variables instead of logically ORing them

Logically ORing boolean variables causes multiple compares to get generated. Bitwise ORing them together will cause them to be arithmetically combined and then a single compare to be generated, which is more efficient.

However, bit fields are NOT boolean variables and must not be bitwise OR'd. A boolean variable is TRUE if non-zero and FALSE if zero.

Note: You cannot bitwise AND boolean variables instead of logically ANDing them!

Wrong:

```
Tboolean A;
Tboolean B;

if ( A || B)
{
        (more code...)
}
```

Right:

```
if (A | B)
{
        (more code...)
}
```

This will fail completely:

```
if (A&B)
{
        (more code...)
}
```

Bit Fields

Bitwise manipulation forces the compiler to generate a lot of excessive shifting, ORing, and ANDing instructions to combine the bit fields. Do not assign one bit to another. Setting, clearing, and testing bit conditions are OK. Assigning bit values to other bit values generates excessive code.

Wrong:

```
if(A.Bit5|B.Bit3)
```

Right:

```
if (A.Bit5 || B.Bit3)
```

Wrong:

```
A.Bit2 = B.Bit4
```

Right:

```
A.Bit2 = 0;

if (B.Bit4)
{
        A.Bit2 = 1;
}
```

Do Not Repeat Logical Tests

If you need to make the same decision twice in a subroutine and the code can't be structured to avoid doing the same test in more than one place then the duplicate tests will generate double the code. It is better to assign the results of the first test to a variable and then test the variable later.

Wrong:

```
if(B==C)
{
        D=E;
}

(...MoreCode...)

if(B==C)
{
        F=G;
}
```

Right:

```
BequalsC=(B==C);

if(BequalsC)
{
        D=E;
}
        (...MoreCode...)
if(BequalsC)
{
        F=G;
}
```

Do Not Repeat Logical Test Values

Even if the logical tests are different, don't re-use the same literal value in multiple logical tests. A Green Hills optimization should minimize the repeated assignment of the same literal to a register used in repeated tests but there will be circumstances in which optimization won't be able to catch repeated use of the same literal. It's better to create a variable assigned to the literal and compare that in each test.

Wrong:

```
if(a==0xff)
{
        if(b==0xff)
        {
                if(c==0xff)
                {
                        d=10;
                }
        }
}
```

Right:

```
e=0xff;

if(a==e)
{
        if(b==e)
        {
                if(c==e)
                {
                        d=10;
                }
        }
}
```

Do Not Post-Increment/Decrement Variables inside 'if' expressions

Doing this forces the compiler to store the original value of the variable in a register, increment it, then test the variable and jump on it. This forces the compiler to work with two copies of the variable instead of one.

Wrong:

```
if(A++)
{
        C=B;
}
```

Right:

```
if(A)
{
        A++;
        C=B;
}
```

Minimize Stack Usage

When stack is in internal RAM, performance is fast; however, when stack is in external RAM, performance is slow. (See Figure 3.3, Memory Regions Occupied by Task RAM Structures, on page 3-61.)

Try To Keep the Number of Function Parameters Under 4

We have four parameter registers (R6 - R9). Passing more than four parameters uses four bytes of stack per additional parameter.

Never Pass Structures by Value (instead, pass pointers to structures)

(If you pass a structure (rather than a pointer to the structure) to a subroutine, the compiler all compilers do this) will 'invisibly' pass the entire structure ONE WORD AT A TIME! Large local structures or arrays should be global.

Set Local Variables As Late As Possible

Don't set local variables where they are defined (at the top of the subroutine). Whenever you set a variable to a value and subsequently call another function the compiler assigns the variable to a permanent register to ensure that its value is preserved across the call. Assigning a variable to a permanent register requires that it be saved on the stack at the start of the function and restored from the stack at the end, totaling 8 bytes per variable. This quickly adds up into large prologues and epilogues costing code space, performance, and stack utilization. You may even write some inefficient looking 'C' code to defer the setting of variables until calls are done which may generate better assembly. In general, look at the assembly output of your 'C' code and see how the compiler is using permanent registers. Try to coerce the compiler into using less permanent registers.

Avoid Using Array Subscript Variables

When you use constants for array subscripts the compiler determines their value so this kind of indexing is free. However when you use variables for array subscripts then the indexing is done at run time, generating lots of assembly instructions. When the array elements are only one byte each this is not too expensive but when they are more than one byte then a multiply must be done to arrive at the effective address. This is even worse when the number of bytes per element is not a power of two because the index can't use a shift operation.

Use Pointers to Structures Instead of Array Subscripts

When you increment/decrement a pointer to a structure, the compiler adds the right value to it automatically. Due to the indirect register oriented nature of this microprocessor, it is very efficient to use pointers to structures in initialization loops and other operations. It is even worth allocating such a pointer as a local variable.

Pass Pointers To Structures to Subroutines

When you pass a pointer to a structure into a subroutine, subsequent references within the subroutine via the indirection operator (->) are just as efficient as referencing the members directly (.).

Design Data Groups in Powers of 2

Multiplies and divides are much more efficient if they are based on powers of 2. This allows you to explicitly do shift operations instead of multiplies, divides, and modulos.

Avoid 'for' Loops Controlled By Counter Variables

One of the costly things you can do is design a loop to initialize all the elements of a structure into a 'for' loop controlled by some counter.

Wrong:

```
for(i=0;i<MAX_STRUCTURE;i++)
{
      jiveStruct[i].count=0;
}
```

This forces the compiler to assemble two essentially parallel operating functions. One is the incrementing and control of the variable 'i', and the other is the initialization of 'JiveStruct'. The 'i' subscript probably generates an ugly multiply operation to access the appropriate structure element. As simple as this 'C' looks, it can generate a LOT of assembly instructions.

Right:

```
struct jiveStruct *pJiveStruct;

pJiveStruct = jiveStruct;

do
{
      *pJiveStruct->count = 0;
      pJiveStruct++;
}while (pJiveStruct < &jiveStruct[MAX_STRUCTURE]);
```

Replace 'for' and 'while' loops with do-while Loops

'For' loops are the most expensive generally, followed by 'while' loops. 'For' loops tend to be oriented around counter variables but are the same as 'while' loops if they're organized around pointers. In either case, 'for' loops and 'while' loops start with a jump to the bottom of the loop where the range is tested. This jump is a three cycle operation. In cases where you know you will always execute the first pass of the loop, use a do-while construct instead. This does not cause a jump to the bottom.

Use 'for(;;)' loops with break tests

The least expensive construct seems to be as follows:

```
for (;;)
{
      if (pJiveStruct < &jiveStruct[MAX_STRUCTURE])
      {
            break;
      }
}
```

Other kinds of loops like this could be do-while(TRUE) loops, but these create a compiler warning 'condition always true', which we discourage. We don't want any warnings being generated as no warnings should be ignored.

Explicitly Cast Multiply/Divide Variables

Although the microprocessor provides multiply and divide opcodes, if you're not careful you can cause the compiler to use a library subroutine to do multiplies and divides. This is because there is only one combination of variable types that the divide opcodes can handle.

e.g.
```
(word) MUL = (hword) operand1 * operand2;

(word) DIV = (word) num / (hword) denom;
```

Note. Multiply and Divide are different.

Use Integer Arithmetic for Modulus

Doing a modulus operation (x = y % 5) causes the compiler to generate a call to a library subroutine. Instead, you should use an integer operation to get the same result (x = y - ((int) (y / 5) *5)). This generates the appropriate arithmetic inline and gives the compiler the opportunity to optimize for any partial results it may have on hand in existing registers. Also, if you have any partial results in existing variables you should use them in the calculation.

The Conditional Expression/Assembly Language Trick

A common trick that assembly language programmers do is to test a condition, make one or more default assignments, then execute a conditional jump, which if not taken, will cause a non-default set of assignments to be made. This saves a jump.

To take advantage of a custom compiler optimization produced for us, code conditional expressions as follows:

```
result = (condition1 == condition2) ? result1 : result2;
```

Don't do this when assigning bits!

Do:
```
result = result2;
      if (condition)
      {
            result = result1;
      }
```

3.1.3.5 Coding Examples

Program 3-1. Code efficiency test module.

```
/*************************************************************************************
 * FILE NAME: codeeff.c

 * CONTENTS: Code efficiency test module.

 * HOW TO USE THIS MODULE:
 * This module contains code for the purpose of determining coding efficiency only.

 * It is not used in the product at all. It should contain a minumum amount of
 * includes, etc.
*************************************************************************************/

        /*lint -e46 */

        /** include files **/
        #include "r_global.h"

        /** public data **/

        #pragma ghs startzda
```

Program 3-2. Code efficiency test variables.

```
/*************************************************************************************
 * CODE EFFICIENCY TEST VARIABLES:
 * This is variable space for my tests for code efficiency.
*************************************************************************************/

        ubyte          globalUbyte1;
        ubyte          globalUbyte2;
        ubyte          globalUbyte3;
        ubyte          globalUbyte4;
        volatile ubyte globalVolatileUbyte1;
        volatile ubyte globalVolatileUbyte2;
        volatile ubyte globalVolatileUbyte3;
        volatile ubyte globalVolatileUbyte4;
        uhword          globalUhword1;
        uhword          globalUhword2;
        uhword          globalUhword3;
        uhword          globalUhword4;
        volatile uhword globalVolatileUhword1;
        volatile uhword globalVolatileUhword2;
        volatile uhword globalVolatileUhword3;
        volatile uhword globalVolatileUhword4;
        word          globalWord1;
        word          globalWord2;
        word          globalWord3;
        word          globalWord4;
        volatile word globalVolatileWord1;
        volatile word globalVolatileWord2;
        volatile word globalVolatileWord3;
        volatile word globalVolatileWord4;
        volatile word globalVolatileWord5;
```

```
struct
{
      uword bitField1a  :15;
      uword bitField1b  :1;
      uword bitField1c  :16;
}globalBitWord1;

struct
{
      uword bitField2a  :15;
      uword bitField2b  :1;
      uword bitField2c  :16;
}globalBitWord2;

struct threeByteStructType
{
      ubyte byteOne;
      ubyte byteTwo;
      ubyte byteThree;
};

struct threeByteStructType globalThreeByteStruct[20];
struct threeByteStructType globalSingleThreeByteStruct;

struct twoDimensionalThreeByteStructType
{
      ubyte byteOne;
      ubyte byteTwo;
      ubyte byteThree;
}globalTwoDimensionalThreeByteStruct[20][100];


#pragma ghs endzda
```

Program 3-3. Multiple stores into same global variable.

```
/*********************************************************************************
 * MULTIPLE STORES INTO SAME GLOBAL VARIABLE.
 * The compiler should keep temporary copies of 1 and 12 in temporary registers
 * in this instance. It should then only store the 1 and 12 into globalUbyte's 1
 * and 2 upon the second set of stores.

 * This only occurs when the volatile assignments are inserted between the non-
 * volatile assignments. You could consider this a compiler efficiency problem
 * and this has been reported to Green Hills.
 *
 * Even though we have seen a compiler inefficiency here, we believe that it is
 * unreasonable to expect the compiler to catch all cases so this kind of 'C'
 * coding inefficiency will be reported.
 ********************************************************************************/

        void SameGlobalVariableBad (void);

        void SameGlobalVariableBad (void)
        {
              globalUbyte1 = 1;
              globalVolatileUbyte1 = globalUbyte1;
              globalUbyte2 = 12;
              globalVolatileUbyte1 = globalUbyte2;

              globalUbyte1 = 1;
              globalVolatileUbyte1 = globalUbyte1;
              globalUbyte2 = 12;
              globalVolatileUbyte1 = globalUbyte2;

        }
```

Program 3-4. OK example.

```
/*********************************************************************************
 * This is the OK example. The compiler figures out that the first two
 * assignments are unnecessary.
 ********************************************************************************/

        void SameGlobalVariableOk (void);

        void SameGlobalVariableOk (void)
        {
              globalUbyte1 = 1;
              globalUbyte2 = 12;

              globalUbyte1 = 1;
              globalVolatileUbyte1 = globalUbyte1;
              globalUbyte2 = 12;
              globalVolatileUbyte1 = globalUbyte2;

        }
```

Program 3-5. Non-word function parameters.

```
/**************************************************************************************
 * NON-WORD FUNCTION PARAMETERS.
 * This caused an unexpected result. Regardless of whether the parameters are
 * words or bytes, the caller converts the data to the parameter type. However, if
 * the parameter is a word type, the callee does not re-convert the data type for its
 * target. It must assume that the conversion is alread done. This is the opposite of
 * what I would expect. In any case, defining function parameters as words
 * eliminates the callee's conversion, saving code.
**************************************************************************************/

        void NonWordParmCaller1 (void);
        void NonWordParmCaller2 (void);
        void NonWordParmCallee1 (ubyte inputParm);

        void NonWordParmCaller1 (void)
        {

                NonWordParmCallee1 (globalVolatileUbyte1);

        }

        void NonWordParmCaller2 (void)
        {

                NonWordParmCallee1 (globalVolatileUbyte2);

        }


        void NonWordParmCallee1 (ubyte inputParm)
        {

                globalVolatileUbyte3 = inputParm;

        }
```

```
/**************************************************************************************
 * If the final destination of an operation is a memory location defined to be a byte
 * or ubyte, the compiler will emit a ST.B instruction which merely stores the low order 8 bits
 * of the register. This obviously performs a truncation. Thus it is not necessary to have
 * the compiler do truncation on intermediate results. The same holds true for hwords and uhwords
**************************************************************************************/
```

Do this:

```
ubyte  ASICREG;

void  F ( word A, word B);
{
        Word   C;
        C  = A + B;
        ASICREG = C;
}
```

Do Not do this:

```
void G (ubyte A, ubyte B)
{
        Ubyte  C;
        C  = A + B;
        ASICREG = C;
}
```

```
/*******************************************************************************
 * Example showing why you use words instead of hwords or bytes
 *******************************************************************************/

        #define ubyte    unsigned char
        #define uhword   unsigned short
        #define uword    unsigned long

        #define byte     signed char
        #define hword    signed short
        #define word     signed long

        #pragma ghs startzda
        word    a;

        void good1(word b, word c)
        {
            a = b + c;
        //    add r6,r7
        //    st.w    r7,zdaoff(_a)[zero]
        //    jmp [lp]
        }

        void good2(uword b, uword c)
        {
            a = b + c;
        //    add   r6,r7
        //    st.w  r7,zdaoff(_a)[zero]
        //    jmp   [lp]
        }

        void bad1(hword b, hword c)
        {
            a = b + c;
        //    shl    16,r6
        //    sar    16,r6
        //    shl    16,r7
        //    sar    16,r7
        //    add    r6,r7
        //    st.w   r7,zdaoff(_a)[zero]
        //    jmp    [lp]
        }

        void bad2(uhword b, uhword c)
        {
            a = b + c;
        //    and    r21,r6
        //    and    r21,r7
        //    add    r6,r7
        //    st.w   r7,zdaoff(_a)[zero]
        //    jmp    [lp]
        }

        void bad3(byte b, byte c)
        {
            a = b + c;
        //    shl    24,r6
        //    sar    24,r6
        //    shl    24,r7
        //    sar    24,r7
        //    add    r6,r7
        //    st.w   r7,zdaoff(_a)[zero]
        //    jmp    [lp]
        }

        void bad4(ubyte b, ubyte c)
        {
```

```
    a = b + c;
//    and    r20,r6
//    and    r20,r7
//    add    r6,r7
//    st.w   r7,zdaoff(_a)[zero]
//    jmp    [lp]
}
```

Program 3-6. Word function parameters.

```
/*******************************************************************************
 * WORD FUNCTION PARAMETERS. When function parameters are words,
 * the data is not converted by the callee.
 *******************************************************************************/
```

```
void WordParmCaller1 (void);
void WordParmCaller2 (void);
void WordParmCallee1 (word inputParm);

void WordParmCaller1 (void)
{

      WordParmCallee1 ((word) globalVolatileUbyte1);

}

void WordParmCaller2 (void)
{

      WordParmCallee1 ((word) globalVolatileUbyte2);

}


void WordParmCallee1 (word inputParm)

{

      globalVolatileUbyte3 = inputParm;

}
```

Program 3-7. Repeated 'if' expressions.

```
/***********************************************************************************
 * REPEATED 'IF' EXPRESSIONS. Whether at the same nesting level or not, repeatedly
 * testing the same condition raises an eyebrow.
 * It may be possible for the author to re-design the logic.  We see that if there is a test
 * on a local variable, the compiler sees that it is the same test and optimizes it.
 * If we test a global variable, even though it is not volatile, the compiler repeats the test.
 * Even though we have seen a compiler inefficiency here, we believe that it is
 * unreasonable to expect the compiler to catch all cases.
 ***********************************************************************************/

        void RepeatedIfExpression (void);


        void RepeatedIfExpression (void)

        {

              if (globalUbyte1 == 1)
              {
                    globalUbyte2 = 12;
              }

              if (globalUbyte1 == 1)
              {
                    globalUbyte3 = 13;
              }
        }
```

Program 3-8. Multiple code fragments that do the same thing.

```
/*******************************************************************************************
 * MULTIPLE CODE FRAGMENTS THAT DO THE SAME THING.
 * Typically, the compiler will be unable to see code fragments that repeat
 * assignments (etc) unless they are very close together.
 * Even though we have seen a compiler inefficiency here, we believe that it is
 * unreasonable to expect the compiler to catch all cases so this kind of 'C' coding
 * inefficiency will be reported.
 *******************************************************************************************/

        void RepeatedFragment1 (void);
        void RepeatedFragment2 (void);

        void RepeatedFragment1 (void)

        {

              globalUbyte1 = 1;
              globalVolatileUbyte1 = 2;
              globalUhword1 = 3;
              globalVolatileUhword1 = 4;
              globalWord1 = 5;
              globalVolatileWord4 = 6;



              /* Now repeat the whole thing within the same function. */

              globalUbyte1 = 1;
              globalVolatileUbyte1 = 2;
              globalUhword1 = 3;
              globalVolatileUhword1 = 4;
              globalWord1 = 5;
              globalVolatileWord4 = 6;

        }

        void RepeatedFragment2 (void)

        {

              globalUbyte1 = 1;
              globalVolatileUbyte1 = 2;
              globalUhword1 = 3;
              globalVolatileUhword1 = 4;
              globalWord1 = 5;
              globalVolatileWord4 = 6;


        }
```

Program 3-9. Repeated constants.

```
/*******************************************************************************
 * REPEATED CONSTANTS. When a constant is used more than once it should be
 * stored in a temporary register. The compiler remembers a single repeated constant.
 * However, it forgets as soon as you introduce a second constant. You should assign
 * repeated constantsb to local variables to ensure that they are remembered.
 *
 * NOTE: The compiler is really bad on this one. Even when we try to assign repeated
 * constants to temporary variables the compiler overrides them and repeatedly sets
 * registers over and over again!!!
*******************************************************************************/

        void RepeatedConstantsBad (void);
        void RepeatedConstantsNoGoodTry (void);

        void RepeatedConstantsBad (void)

        {

              globalVolatileWord1     = 1;
              globalVolatileWord1     = 2;
              globalVolatileWord2     = 1;
              globalVolatileWord2     = 2;
              globalVolatileWord3     = 1;
              globalVolatileWord3     = 2;
              globalVolatileWord4     = 1;
              globalVolatileWord4     = 2;

        }

        void RepeatedConstantsNoGoodTry (void)

        {

              word valueOne = 1;
              word valueTwo = 2;

              globalVolatileWord1     = valueOne;
              globalVolatileWord1     = valueTwo;
              globalVolatileWord2     = valueOne;
              globalVolatileWord2     = valueTwo;
              globalVolatileWord3     = valueOne;
              globalVolatileWord3     = valueTwo;
              globalVolatileWord4     = valueOne;
              globalVolatileWord4     = valueTwo;

        }
```

Program 3-10. Bitwise or'd bit fields.

```
/**********************************************************************************
 * BITWISE OR'd BIT FIELDS. In an 'if' expression, you must not OR together two
 * bit fields BITWISE and then test the resulting condition as you would whole word
 * variables. This is because the resulting shifting ANDing and ORing is huge.
 * This is a nasty mistake that I haven't seen happen very often but it is a big
 * mistake if it does occur.
 * It will be a low priority to find this case because of its unlikelihood.
**********************************************************************************/

        void BitwiseOredFieldsBad (void);
        void BitwiseOredFieldsGood (void);


        void BitwiseOredFieldsBad (void)
        {

              if (globalBitWord1.bitField1b |
        globalBitWord2.bitField2b)
              {

                 globalVolatileWord4 = 1;
              }
        }



        void BitwiseOredFieldsGood (void)
        {

              if (globalBitWord1.bitField1b ||
        globalBitWord2.bitField2b)
              {

                 globalVolatileWord4 = 1;
              }
        }
```

Program 3-11. Post-incrementing a variable within a 'while' (or other conditional) statement

```
/*******************************************************************************
 * POST-INCREMENTING A VARIABLE WITHIN A 'WHILE'
 * (or other conditional) STATEMENT:
 * Post increments force the compiler to save a temporary copy of the register
 * assigned to the associated variable so that the pre-incremented version can be
 * tested. This takes an extra two bytes per statement.
 * We should find lots of these.
*******************************************************************************/

void PostIncrementIfExpressionBad (void);
void PostIncrementIfExpressionGood (void);

void PostIncrementIfExpressionBad (void)

{

      word  postIncrementLoopControl = 0;

      while ( postIncrementLoopControl++ < 100 )
      {

         globalVolatileWord4 = 1;

      }

}

void PostIncrementIfExpressionGood (void)

{

      word  postIncrementLoopControl = 0;

      while ( postIncrementLoopControl < 100 )
      {

         globalVolatileWord4 = 1;
         postIncrementLoopControl++;

      }

}
```

Program 3-12. Subroutine with more than 4 parameters.

```
/*******************************************************************************
 * SUBROUTINE WITH MORE THAN 4 PARAMETERS:
 * There are only four registers (R6 - R9) dedicated to parameter passing. Functions with more
 * than 4 parameters force the stack to be used for the additional parameters. This
 * uses more code than functions with four or less parameters. Note that in my
 * example I execute the same number of instructions regardless of the number
 * of parameters.
*******************************************************************************/

        void FiveParameterFunction (word parm1, word parm2, word
        parm3, word parm4, word parm5)
        {

              globalVolatileWord1 = parm1;
              globalVolatileWord2 = parm2;
              globalVolatileWord3 = parm3;
              globalVolatileWord4 = parm4;
              globalVolatileWord5 = parm5;

        }


        void FourParameterFunction (word parm1, word parm2, word
        parm3, word parm4)
        {

              globalVolatileWord1 = parm1;
              globalVolatileWord2 = parm2;
              globalVolatileWord3 = parm3;
              globalVolatileWord4 = parm4;
              globalVolatileWord5 = parm4;

        }
```

Program 3-13. Arrays using subscript variables.

```
/*******************************************************************************
 * ARRAYS USING SUBSCRIPT VARIABLES: This forces the compiler to use
 * arithmetic for access to the elements. Pointer arithmetic generates less assembly,
 * even with the counter shown below included.
*******************************************************************************/

        void ArrayWithSubscripts (void);
        void ArrayWithoutSubscripts (void);

        void ArrayWithSubscripts (void)

        {

        hword arrayIndex = 0;

              do
              {

                    globalThreeByteStruct[arrayIndex].byteOne = 1;
                    globalThreeByteStruct[arrayIndex].byteTwo = 2;
                    globalThreeByteStruct[arrayIndex].byteThree = 3;
                    arrayIndex++;
              }while (arrayIndex < 20);

        }

        void ArrayWithoutSubscripts (void)

        {

        word  arrayCount = 0;
        struct  threeByteStructType *structPointer =
        &globalThreeByteStruct[0];
              do
              {

                    structPointer->byteOne = 1;
                    structPointer->byteTwo = 2;
                    structPointer->byteThree = 3;
                    arrayCount++;
                    structPointer++;
              }while (arrayCount < 20);

        }
```

Program 3-14. Arrays using two subscript variables.

```
/*******************************************************************************
 * ARRAYS USING TWO SUBSCRIPT VARIABLES:
 * This forces the compiler to use even more complex arithmetic for access to
 * the elements. Pointer arithmetic generates less assembly, even with the counter
 * shown below included.
 * NOTE: The code size difference here is dramatic.
 ******************************************************************************/

void ArrayWithTwoSubscripts (void);
void ArrayWithoutTwoSubscripts (void);

void ArrayWithTwoSubscripts (void)

{

hword arrayIndex1 = 0;
hword arrayIndex2 = 0;

      do
      {
            do
            {
            globalTwoDimensionalThreeByteStruct
                  [arrayIndex1][arrayIndex2].byteOne = 1;

                  globalTwoDimensionalThreeByteStruct
                  [arrayIndex1][arrayIndex2].byteTwo = 2;

                  globalTwoDimensionalThreeByteStruct
                  [arrayIndex1][arrayIndex2].byteThree = 3;

                  arrayIndex1++;

            }while (arrayIndex2 < 100);
      }while (arrayIndex1 < 20);

}

void ArrayWithoutTwoSubscripts (void)

{

word  arrayCount1 = 0;
word  arrayCount2 = 0;

struct  twoDimensionalThreeByteStructType *structPointer1 =
&globalTwoDimensionalThreeByteStruct[0][0];
struct  twoDimensionalThreeByteStructType *structPointer2 =
&globalTwoDimensionalThreeByteStruct[0][0];

      do
      {
            structPointer2 = structPointer1;

            do
            {
                  structPointer2->byteOne = 1;
                  structPointer2->byteTwo = 2;
                  structPointer2->byteThree = 3;
                  arrayCount1++;
```

```
                        structPointer2++;
                }while (arrayCount2 < 100);

                structPointer1 +=100;
                // Note that this adds 100 * size of structure

        }while (arrayCount1 < 20);
}
```

Program 3-15. Arrays or structures as function parameters.

```
/*********************************************************************************
 * ARRAYS OR STRUCTURES AS FUNCTION PARAMETERS: This forces
 * the compiler to push the entire array or structure onto the stack.
 * Instead, you should use a pointer to the array or structure. The example below uses
 * a structure.
*********************************************************************************/
```

```
void ArrayParamCaller (void);
void ArrayParamCallee (struct threeByteStructType
testThreeByteStruct);
void ArrayAddressParamCallee (struct threeByteStructType
*testThreeByteStruct);

void ArrayParamCaller (void)

{

        ArrayParamCallee (globalSingleThreeByteStruct);

}

void ArrayParamCallee (struct threeByteStructType
testThreeByteStruct)

{

        globalUbyte1 = testThreeByteStruct.byteOne;
        globalUbyte2 = testThreeByteStruct.byteTwo;
        globalUbyte3 = testThreeByteStruct.byteThree;

}

void ArrayAddressParamCallee (struct threeByteStructType
*testThreeByteStruct)

{

        globalUbyte1 = testThreeByteStruct->byteOne;
        globalUbyte2 = testThreeByteStruct->byteTwo;
        globalUbyte3 = testThreeByteStruct->byteThree;

}
```

Program 3-16. For and while loops vs do-while loops.

```
/*******************************************************************************
* FOR AND WHILE LOOPS VS DO-WHILE LOOPS: For loops and while loops
* (with the 'while' statement at the top) each generate two more bytes of code than
* do-while loops. The reason is that the for and while loops execute an initial jump
* to the bottom of the routine to test the range. Of course, the do-while method only
* works if you know that the function must execute the first pass of the loop.
*******************************************************************************/

        void ForLoopExample (void);
        void DoWhileExample (void);
        void WhileLoopExample (void);

        void ForLoopExample (void)

        {
              word  exampleCount;

              for   (exampleCount = 0; exampleCount < 10;
              exampleCount++)
              {
                    globalVolatileUbyte1 = 1;
              }
        }


        void WhileLoopExample (void)

        {
              word  exampleCount = 0;

              while (exampleCount < 10)
              {
                    globalVolatileUbyte1 = 1;
                    exampleCount++;
              };
        }


        void DoWhileExample (void)

        {
              word  exampleCount = 0;

              do
              {
                    globalVolatileUbyte1 = 1;
                    exampleCount++;

              }while (exampleCount < 10);
        }
```

Program 3-17. Byte and half-word local variables.

```
/*************************************************************************************
 * BYTE AND HALF-WORD LOCAL VARIABLES: Local numeric variables should
 * always be WORDs unless specifically required to be smaller.
 * The reason for this is that the V851 is a 'signed' processor, meaning that whenever
 * it loads a byte or halfword into a register it 'sign extends' the high-order bit.
 * If you unnecessarily declare a local counter variable as a byte (for instance), then
 * whenever any math is done on it, extra assembly instructions will be generated
 * to mask off any overflow bits.
 *************************************************************************************/

        void LocalByteExample (void)
        {

        byte localByte1;

              localByte1 = 1;

              do
              {

                    globalVolatileUbyte1 = 1;
                    localByte1++;
                    // Extra code generated to mask off the upper 24
                    (overflow) bits.

              }while (localByte1 < 10);

        }

        void LocalHwordExample (void)
        {

        hword localHword1;

              localHword1 = 1;

              do
              {

                    globalVolatileUbyte1 = 1;
                    localHword1++;
                    // Extra code generated to mask off the upper 16
                    (overflow) bits.

              }while (localHword1 < 10);

        }

        void LocalWordExample (void)
        {

        word localWord1;

              localWord1 = 1;

              do


              {
                  globalVolatileUbyte1 = 1;
                  localWord1++;
                  // No extra code generated.

            }while (localWord1 < 10);

      }
```

(This page was intentionally left blank.)

3.2 NEC V851 Microprocessor

3.2.1 Architecture

This document describes the 'Atlas' V851 Microprocessor architecture as it affects firmware development. For further documentation on the processor, you can refer to the appropriate NEC publications.

3.2.1.1 Instruction Set

The following summary shows you two important values for each V851 machine instruction; the size in bytes and the number of cycles required to execute the instruction. As you develop 'C' and assembly code, both code size and performance can be revealed by studying the resulting assembly language using this chart.

This table is most commonly used to perform manual 'pipeline' optimization of assembly code by determining the latency associated with each instruction. For instance, if an instruction is executed with an issue value of 1, but a latency value of 2, then the results of the instruction can't be used for 2 cycles but another instruction can be executed within 1 cycle. So, you will insert some unrelated instruction (such as initializing a register for some future operation) after the current one to make use of the machine cycle sacrificed by the latency.

The compiler automatically performs pipeline optimization, but the assembler does not. Latency can account for about a 20% performance loss unless optimization is done.

The size and performance values are as follows:

- Bytes: The size in bytes of the instruction
- Issue: The number of machine cycles required for the instruction.
- Repeat: The number of machine cycles required until the same instruction can be repeated.
- Latency: The number of machine cycles required until the result is available.

Bytes	Issue	Repeat	Latency	Description
2	1	1	1	mov reg1,reg2
2	1	1	1	mov imm5,reg2
4	1	1	1	movea imm16,reg1,reg2
4	1	1	1	movhi imm16,reg1,reg2
4	1	1	2	ld.b disp16[reg1],reg2
4	1	1	2	ld.h disp16[reg1],reg2
4	1	1	2	ld.w disp16[reg1],reg2
4	1	1	1	st.b reg2,disp16[reg1]
4	1	1	1	st.h reg2,disp16[reg1]
4	1	1	1	st.w reg2,disp16[reg1]
2	1	1	2	sld.b disp7[ep],reg2
2	1	1	2	sld.h disp8[ep],reg2
2	1	1	2	sld.w disp8[ep],reg2
2	1	1	1	sst.b reg2,disp7[ep]
2	1	1	1	sst.h reg2,disp8[ep]
2	1	1	1	sst.w reg2,disp8[ep]
2	1	1	1	add reg1,reg2

Bytes	Issue	Repeat	Latency	Description
2	1	1	1	add imm5,reg2
4	1	1	1	addi imm16,reg1,reg2
2	1	1	1	sub reg1,reg2
2	1	1	1	subr reg1,reg2
2	1	1	1	cmp imm5,reg2
2	1	1	1	cmp reg1,reg2
2	1	1	1	shl imm5,reg2
4	1	1	1	shl reg1,reg2
2	1	1	1	shr imm5,reg2
4	1	1	1	shr reg1,reg2
2	1	1	1	sar imm5,reg2
4	1	1	1	sar reg1,reg2
2	1	1	1	and reg1,reg2
4	1	1	1	andi imm16,reg1,reg2
2	1	1	1	or reg1,reg2
4	1	1	1	ori imm16,reg1,reg2
2	1	1	1	xor reg1,reg2
4	1	1	1	xori imm16,reg1,reg2
2	1	1	1	not reg1,reg2
2	1	1	1	tst reg1,reg2
4	4	4	4	set1 bitnum,disp16[reg1]
4	4	4	4	clr1 bitnum,disp16[reg1]
4	4	4	4	not1 bitnum,disp16[reg1]
4	3	3	3	tst1 bitnum,disp16[reg1]
2	1	1	2	mulh reg1,reg2
2	1	1	2	mulh imm5,reg2
4	1	1	2	mulhi imm16,reg1,reg2
2	36	36	36	divh reg1,reg2
2	1	1	1	satadd reg1,reg2
2	1	1	1	satadd imm5,reg2
4	1	1	1	satsubi imm16,reg2
2	1	1	1	satsub reg1,reg2
2	1	1	1	satsubr reg1,reg2
2	*	*	*	bcond disp9
2	3	3	3	jmp [reg1]
4	3	3	3	jr disp22[pc]
4	3	3	3	jarl disp22[pc],reg2
4	1	1	1	setf cccc,reg2
4	1	1	1	halt
4	4	4	4	trap vector
4	4	4	4	reti
4	1	1	*	ldsr reg2,regID
4	1	1	1	stsr regID,reg2
4	1	1	1	di
4	1	1	1	ei
2	1	1	1	nop

Table 3-1. Instruction Set Size and Latency Summary

* See pp 217 - 220 of the *NEC: V851TM User's Manual Sept., 1996*

3.2.1.2 Registers

The V851 provides thirty-two 32-bit registers. In order to create efficient 'C' code and to write assembly code that works, you must understand how these registers are used. Registers are used in the following ways:

- Dedicated by the microprocessor architecture R0, R30 (ep), PSW
- Reserved by special purposes by the compiler or assembler, R6 - R9, R10, R1
- Reserved by servo code as agreed upon by Quantum development teams
- Temporary registers useable by 'C' and assembly code
- Permanent registers useable by 'C' and assembly code

Register	Usage	Explanation
r0	Dedicated by Microprocessor	The Zero Register. Always contains zero. Bit bucket if written to.
r1	Reserved	
r2	Reserved	Will become available with version 3.0 emulator
r3	Dedicated by compiler	Stack Pointer (sp)
r4	Reserved by compiler	Global Pointer for SDA (Small Data Area Access) (gp)
r5	Reserved	Will become available with version 3.0 emulator
r6	Temporary Register	Parameter register for function calls
r7	Temporary Register	Parameter register for function calls
r8	Temporary Register	Parameter register for function calls
r9	Temporary Register	Parameter register for function calls
r10	Temporary Register	Return value from function calls.
r11	Temporary Register	General purpose temporary register
r12	Temporary Register	General purpose temporary register
r13	Temporary Register	General purpose temporary register
r14	Temporary Register	General purpose temporary register
r15	Temporary Register	General purpose temporary register
r16	Temporary Register	General purpose temporary register
r17	Dedicated to Servo	Not used by other firmware
r18	Dedicated to Servo	Not used by other firmware
r19	Dedicated to Servo	Not used by other firmware
r20	Reserved	Always contains 0xFF for masking and compares
r21	Reserved	Always contains 0xFFFF for masking and compares
r22	Dedicated to Servo	Not used by other firmware
r23	Permanent Register	General purpose permanent register
r24	Permanent Register	General purpose permanent register
r25	Permanent Register	General purpose permanent register
r26	Permanent Register	General purpose permanent register
r27	Permanent Register	General purpose permanent register
r28	Permanent Register	General purpose permanent register
r29	Permanent Register	General purpose permanent register
r30	Dedicated by Microprocessor	Element Pointer for 2-byte load/stores (ep)
r31	Dedicated by compiler	Link Pointer(return address)for function calls (lp)

Table 3-2. V851 Registers - Eclipse Configuration (26 register mode)
Except for r0 and r30, these registers are based from GHS usage.

Twenty-Six Register Mode

As a compiler option, this model of operation specifies that only 26 of the 32 registers are available to the compiler. By selecting 26 register mode we allow some registers to be reserved by servo code. This would normally allow servo to use six registers without saving them on the stack each time it executes. However, we have selected an option that allows the compiler to use r20 and r21 as mask registers, forcing servo to reset them to their mask values after using them. This reduces the number of dedicated servo registers to four.

Register Zero

This register is hard-wired by the V851 to always contain the value zero (zero, or r0). This saves code by providing an immediate value for initialization, compare, and data access. Writing to it is writing to the bit bucket.

Small Data Area Access

This data access method uses the global pointer register (gp or r4) as a base register. It provides 32K in either direction from the global pointer. Although this is not currently used by the Eclipse code, it is an available option.

Temporary Registers

The registers r6 - r16 are used most freely by the compiler. It assumes that all of them are destroyed by function calls. A function is free to use a temporary register w/o saving it's contents. Therefore a caller of a function can make no assumptions about the value in a temporary register upon return from the called function. The typical use of temporary registers is for values that need not be preserved across calls.

Permanent Registers

The registers r23 - r29 are typically used for values that must be preserved across function calls. The compiler assumes that any function that is called will preserve the contents of any permanent registers that it changes. A function will preserve the original value (save/restore using the stack) stored in any permanent register it uses. Therefore, a caller of a function can assume that the value in all permanent registers will be preserved across the function call.

Function Parameter Registers

The registers r6 - r9 are used for passing parameters to functions. Functions with more than 4 parameters force the compiler to pass the additional ones on the stack, which is inefficient. These registers are also used as temporary registers within functions.

Function Return Value Register

The register r10 is used to provide the return value of functions to their callers. It is also treated as a temporary register since it can be destroyed by any function that does not return a value.

Dedicated Servo Registers

The registers r17, r18, r19, and r23 are used by servo code only. Since only the servo code uses these registers, they need not be saved. This makes the servo code faster.

Mask Registers

We have selected a compiler option to use registers r20 and r21 as mask registers. The value of r20 is always 0xff and the value of r21 is always 0xffff. This allows the compiler to do more code efficient masking operations.

Element Pointer Registers

The register r30 (ep) is hard-wired in the V851. (i.e., the short loads and stores implicitly use ep. It is used by the compiler to generate short (2-byte) loads and stores to RAM. This is typically done for structure access in code fragments that do many accesses to the same structure. As a result, the compiler will use the ep register periodically throughout the code where it can take advantage. The compiler generates object code which initializes the element pointer with the base address of the structure. It can then use ep-relative short loads and stores to access the structure's members. The largest displacement from the element pointer that a structure member can have is 127 bytes when accessed as a byte, or 255 bytes when accessed as a halfword or word.

Link Pointer

The register r31 (lp) is used as a return address when calls to functions are issued. Since there is only one link pointer, any function that performs a call must save the link pointer on the stack.

3.2.1.3 Data Formats

Data for the V851 is stored in three different formats; words, halfwords, and bytes. The following rules show how your data will be affected by these formats:

Words

Words are 32 bits in size. They must be stored on 32-bit boundaries. The compiler enforces this so in order to avoid unused spaces between members your structures should define all the words before halfwords and bytes. This is the preferrable format for arithmetic operators.

Halfwords

Halfwords are 16 bits in size. They must be stored on 16-bit boundaries. The compiler enforces this so in order to avoid unused spaces between members structures should define all the halfwords before bytes.

Bytes

Bytes are 8 bits in size.

3.2.1.4 Signed Operations

The V851 processor is a 'signed' processor. That is, data that is loaded from memory into registers is sign extended. Immediate values in machine instructions are sign extended before being applied to the instruction, and multiply and divide operations are sign-extended. This affects the three data formats in the following ways:

Words — No sign extension is performed when loading words from memory, since this is the largest data format.

Halfwords — Bit 15 of memory is sign extended when loaded into a register.
For instance, if a halfword value of 0x8000 is loaded into a register,
the register will become 0xffff8000

Bytes — Bit 7 of memory is sign extended when loaded into a register.
For instance, if a halfword value of 0x80 is loaded into a register,
the register will become 0xffffff80

3.2.1.5 Saturated Math

Saturated math instructions are special machine instructions which allow registers to reach their full positive or negative values and stay at that value. For example, the maximum positive value of any register is 0x7fffffff. If any saturated math operation would cause the register to exceed 0x7fffffff then the value would be set to 0x7fffffff. The same goes for negative values. The maximum negative value of a register is 0x80000000. Any saturated math operation that would cause the register to exceed 0x80000000 would cause the register to be set to 0x80000000. This generally affects assembly language programming as there is no 'C' language facility to take advantage of saturated math operations.

Note. GHS provides saturated math operations as special inline functions; however, they are not that useful. GHS also provides EI() and DI()as special inline functions. (See page 33 of the *Green Hills Software: Embedded V800 Development Guide, Version 1.8.8*)

3.2.1.6 Hardware Multiply and Divide

The V851 provides multiply and divide instructions which can greatly improve the efficiency of the firmware object code. However, it is sometimes necessary to coerce the compiler into using hardware multiplies and divides by explicitly casting operands. If the operands are not of the appropriate data types, the compiler will use a library function, which is very inefficient. The following data types are required for hardware multiply and divide instructions to be generated.

Multiply — If both operands are variables then both operands must be no greater than half-words in size. If one of the operands is a constant, then less code will be generated if the operand is not greater than 15 or less than -16.

Divide — Divide is word = word/hword. There is no divide instruction which uses a constant.

3.2.1.7 Execution Speed

The V851 processor is very fast. When driven by a 25 Mhz clock, each single-cycle instruction executes in only 40 ns. However, executing data loads and stores to external RAM causes wait states to be used, resulting in approximately 1 microsecond per instruction. So, it is vital to avoid external RAM accesses in performance-critical code sections.

3.2.1.8 Pipeline Stalls

The V851 is generally a RISC processor. In other words, most instructions require only 1 machine cycle to execute. However, some instructions require more than one machine cycle in which to present their data. For instance a load from internal RAM into a register executes in 1 machine cycle but the register contents are not available for another machine cycle. The trick is to insert another unrelated instruction after the load to use the next machine cycle while the previous one completes. This is called pipeline optimization. The compiler performs pipeline optimization automatically for you. However, the assembler does not. So you must code in pipeline optimization by hand when writing in assembly language.

See the instruction set summary in this document for the machine cycle requirements associated with each instruction.

(This page is intentionally left blank.)

3.3 The Multi-Stack Kernel

The multi-stack kernel provides increased drive performance through the *deterministic* (repeatable or consistent response to events) execution of concurrent tasks. Execution of tasks is generally independent; normally the tasks don't need to know anything about each other. However, since they must share resources, some degree of coordination is required.

Each task is provided with its own stack. Each task also has a 'context' structure where the kernel saves its registers when the task is not executing.

Tasks are executed by priority. When the processor is not executing an interrupt service routine, the highest priority active task executes.

Tasks have two basic states; active and suspended. When a task is in the active state, it can execute. However, an active task may not actually be executing because it may be preempted by a higher priority task.

A task becomes suspended by 'pending' on *events* (indications of some occurrence which are 'posted' by ISRs or tasks). A suspended task cannot execute. A task is made active when an event that it had pended on is posted.

The multi-stack kernel consists of three basic components; an ISR handler, a scheduler, and a set of functions for pending, posting, and suspending.

3.3.1 Advantages of a Multi-Stack Kernel

Some of the technical merits of the multi-stack kernel are as follows:

- **Code can be written in a linear manner**
  Each task goes about its business in a straight line. No special code is built into the tasks to pass control to other tasks. This makes the code much more straightforward.

- **Smaller code**
  When compared with the cooperative multi-tasking kernel or code which manages multi-tasking via flags and polling loops, code written for the multi-stack kernel is more compact because it focuses multi-tasking within the kernel itself.

- **Code is easier to understand**
  Because the code is linear and multi-tasking is hidden within the kernel, each task is easier to understand than code written for the cooperative multi-tasking kernel or code which manages multi-tasking via flags and polling loops.

- **Less latency in response to events**
  Because the kernel causes a task that is pending on an event to preempt other tasks immediately upon the posting of an event, the event response latency is consistent and fast.

3.3.2 Disadvantages of a Multi-Stack Kernel

Here is one trade-off associated with the multi-stack kernel:

- **Limited Amount of Internal RAM**
  Because the kernel uses multiple stacks and contexts which must support high performance, it uses internal RAM for stacks and contexts where speed is important. The V851 processor has only 2K of internal RAM. However, we have found in practice that it is possible to implement the multi-stack kernel and still have enough internal RAM to produce a very fast drive.

3.3.3 Context Switch Latency

The average time that it takes to post an event and then switch from one task to another is approximately 6 microseconds.

3.3.4 The Kernel's ISR Handler

When an interrupt occurs and the microprocessor PC jumps to the interrupt vector, special handling occurs. The interrupt's service routine address is captured and the kernel's ISR handler begins execution.

The kernel's ISR handler keeps track of the 'ISR nesting level'. If a task is interrupted then the ISR nesting level is zero. If an interrupt service routine is interrupted by a higher priority interrupt then the ISR nesting level is 1 or greater and the new interrupt's service routine is executed without special processing.

If the ISR nesting level is zero when an interrupt occurs then the kernel's ISR handler saves the current task's registers in the current task's context structure. It then executes the interrupt service routine. Upon return from the service routine the scheduler is executed. The scheduler could then resume execution of a different task than the one that was interrupted if a new event was posted by the interrupt service routine. This would only happen if the posted event activates a higher priority task (preemption).

Interrupt service routines written for the multi-stack kernel are normal functions. They are not coded with the 'interrupt pragma'. However, they will typically enable interrupts almost immediately upon starting. The Kernel's ISR handler takes care of saving and restoring registers.

The servo (NMI) is not included in this scheme. It operates independently of the kernel. Also, the timer overflow and retract interrupt. This is because none of these generate events.

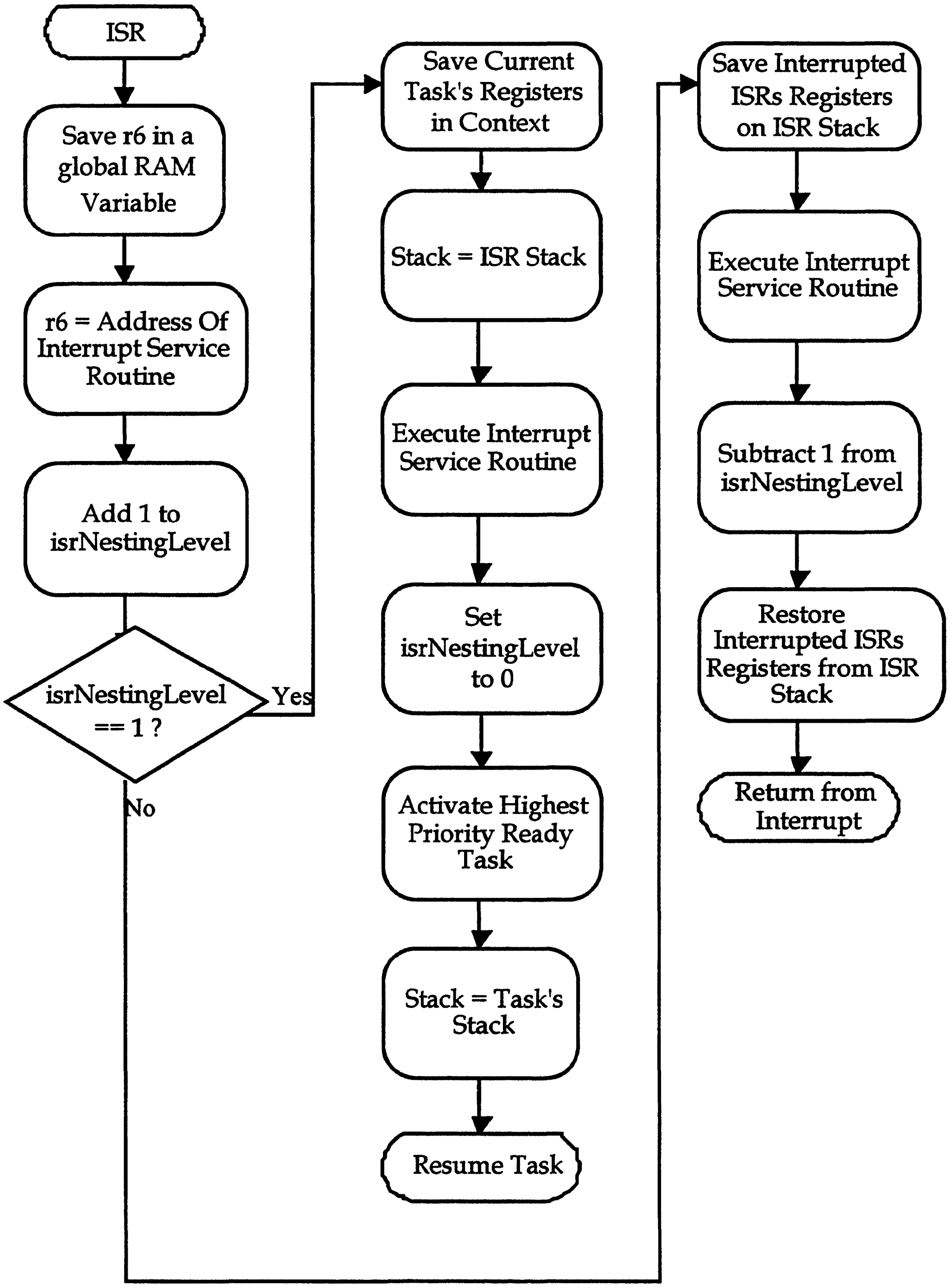

ISR
Save r6 in a global RAM Variable
r6 = Address Of Interrupt Service Routine
Add 1 to isrNestingLevel
isrNestingLevel == 1 ?
Yes
No
Save Current Task's Registers in Context
Stack = ISR Stack
Execute Interrupt Service Routine
Set isrNestingLevel to 0
Activate Highest Priority Ready Task
Stack = Task's Stack
Resume Task
Save Interrupted ISRs Registers on ISR Stack
Execute Interrupt Service Routine
Subtract 1 from isrNestingLevel
Restore Interrupted ISRs Registers from ISR Stack
Return from Interrupt

Figure 3-1. Kernel ISR Handling

3.3.5 The Kernel's Scheduler

The kernel's scheduler is executed under two different circumstances; when the Kernel's ISR handler is managing the return from an interrupt service routine with an ISR nesting level of zero or when a task posts an event which is pended on by some suspended task of higher priority.

A change of the current task from one task to another is called a 'context switch'.

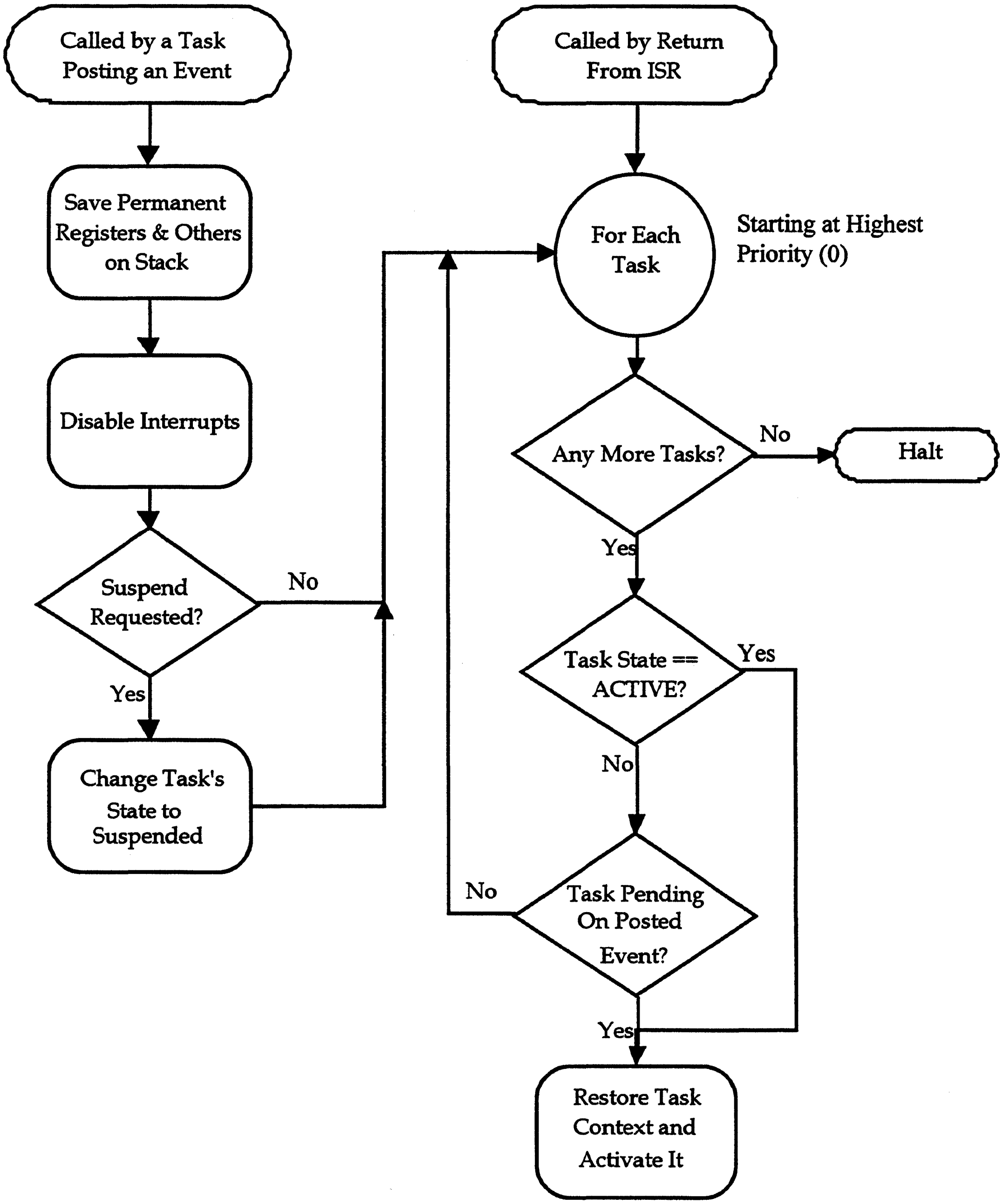

Called by a Task Posting an Event
Save Permanent Registers & Others on Stack
Disable Interrupts
Suspend Requested?
No
Yes
Change Task's State to Suspended
Called by Return From ISR
For Each Task
Starting at Highest Priority (0)
Any More Tasks?
No
Halt
Yes
Task State == ACTIVE?
Yes
No
Task Pending On Posted Event?
No
Yes
Restore Task Context and Activate It

Figure 3-2. Kernel Scheduler

3.3.6 Pending and Posting

The multi-stack kernel provides a set of tools which implement various flavors of these three basic functions:

- Pending
  The calling task provides one or more bits indicating the event(s) which can activate it once it suspends. The caller will suspend immediately if none of the events on which it is pending have been posted. There are thirty-two potential event bits.

- Posting
  The calling task or interrupt service routine provides one or more bits indicating event(s) which have occurred. If there is a higher priority task pending on an event that was posted then the calling task will be immediately preempted by the higher priority task. Interrupt service routines are not preempted in this way. In that case, the pending task is activated when the lowest level interrupt service routine is exited. .

3.3.7 Task States

There are two basic states that a task can be in; active and suspended. However, there's also a transitional state called 'suspended request' which will be discussed. These states are as follows:

- Active
  The task is ready to run although it may not actually be running. An active task could be preempted by a higher priority task. In that case the task remains active but control is given to the higher priority task. When not servicing an interrupt, the highest priority active task always runs. If there are no active tasks then the processor executes a HALT instruction and can only be awakened by an interrupt. A task changes from the suspended to the active state when an event on which it was pending is posted.

- Suspended
  The task is pending on one or more events and has given up microprocessor control to the next lower priority active task until one of the events is posted. When suspended, all the task's registers are saved in its context area. This includes its PSW and PC. When the task is activated again it will resume execution at the instruction immediately following the point at which it became suspended.

- Suspended Request
  This is an internal state that is equivalent to suspended. It is required in order to safely implement the transition from active to suspended.

3.3.8 Tasks in the Eclipse Code

Eclipse has implemented five tasks:

1. Serial Debug (priority 0)
   This task's stack and context are in external, DRAM.

2. Exception Task (priority 1)
   This task is designed for maximum determinism with low speed. Posting an event to the exception task will cause all the other tasks to be preempted immediately. However, once started, this task executes slowly because its stack and context are in external RAM. This task is used for abort command and reset functions.

3. Command Task (priority 2)
   This is the main command execution task. It is responsible for completion of all host commands and inquiries. It is designed for moderate speed, with its context in internal RAM and its stack starting in internal RAM and extending into external RAM.

4. Pre-Processor Task (priority 3)
   This task has two primary responsibilities; handing off commands from the low level interface to the command task and performing as much processing as possible on commands waiting in the queue while the command task waits for events from the disk. This task is designed for greatest speed, with its context and stack in internal RAM.

5. Background Task (priority 4)
   This task performs low priority background operations. During power-up, it performs drive initialization. During normal operations it performs SMART functions. It also performs self-scan operations. Not designed for speed, its context and stack are in external RAM.

3.3.9 Multi-Stack Kernel RAM Structures

Each task has the following data structures associated with it:

- TCB
  This is the Task Control Block. This is the main identifier for the task. It contains the events pended on by the task, a pointer to its context, and its state.

- Context
  This structure contains all the registers that must be saved when a task is interrupted or preempted.

- Stack
  This is a block of memory where the task's stack will be maintained.

3.3.10 TCB, Context, and Stack Locations

Where performance is critical various RAM structure will be located in internal RAM. Where performance is non-critical, the RAM structures will be located in external RAM. TCBs are always located in internal RAM. The following figure illustrates the locations of each task's RAM structures:

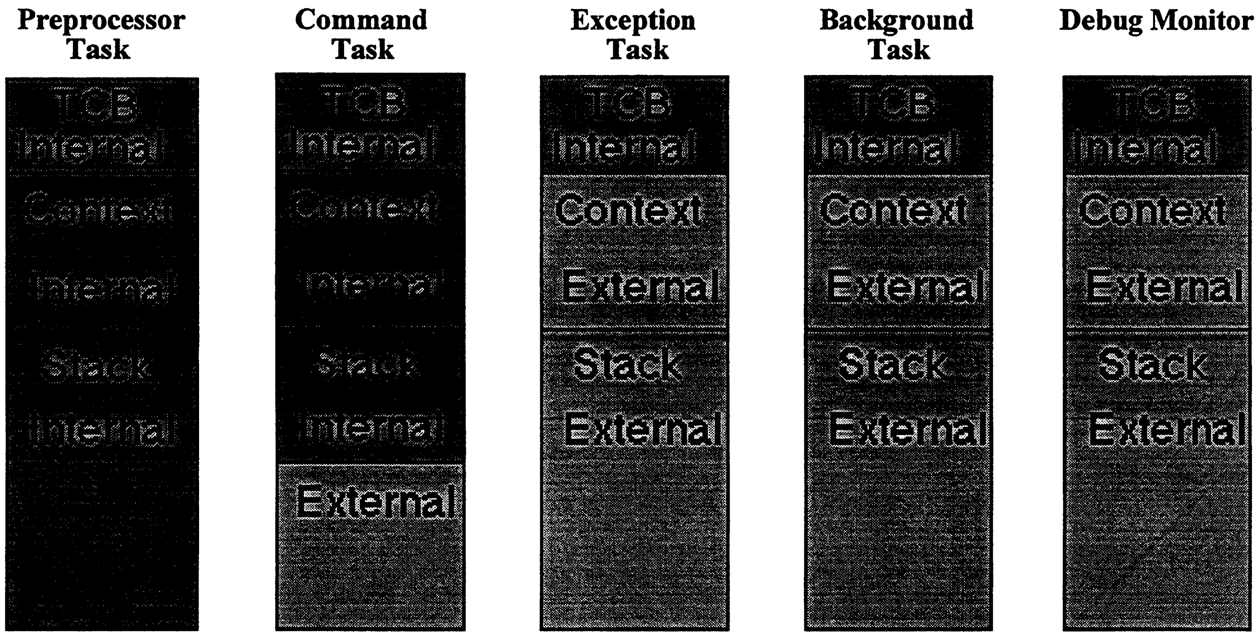


Figure 3-3. Memory Regions Occupied by Task RAM Structures

An example of a multi-tasking scenario

Below is a logic-analyzer like example of write operations on a typical multi-tasking drive:

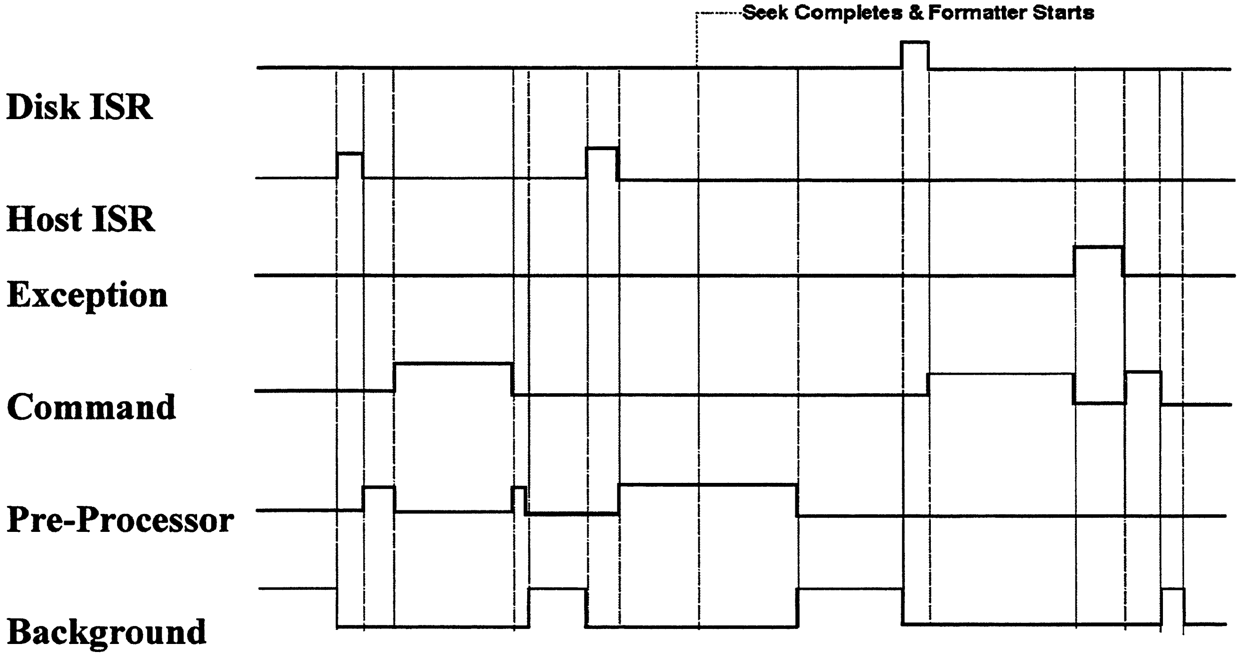


Figure 3-4. A typical hyper-write under multi-tasking control

In the above diagram, background, pre-processor, command, and exception activities are 'tasks'. The host and disk ISRs are not tasks. The active state of the lines of the diagram is high and inactive is low. The following describes the above scenario:

- The background task is active and running with some 'SMART' activity. The command, pre-processor, and exception tasks are all suspended, pending on events.

- A write command is received from the host. The host ISR executes.

- The pre-processor task (which was pending on a command received event) activates and executes, pre-empting the background, and fills in a queue entry for the command. It then posts a command ready event.

- The command task (which was pending on a command ready event) activates and executes, pre-empting the pre-processor task, and performs the write command. When it starts the disk write, it pends on a disk event, suspending until the event occurs.

- The pre-processor executes (because it was pre-empted by the command task and is the next lower priority task) and looks for something to do. It has no work to do so it pends on another command received event, and suspends.

- The background task resumes execution (because it was pre-empted by the pre-processor task and is the next lower priority task) and continues with its SMART activity.

- Another write command comes in from the host. The host ISR executes and posts a command received event.

- The pre-processor task (which was pending on a command received event) activates and executes, pre-empting the background, and fills in a queue entry for the command. It then posts a command ready event. Since the command task is pending on a disk event (not a command ready event), the pre-processor is not pre-empted by it. The pre-processor can go on and decode the command, allocate cache, receive the host data, etc. while the command task is waiting for the disk to finish. When the pre-processor task runs out of work to do it pends on a command received event, and suspends.

- Because it was pre-empted by the pre-processor task, the background task resumes execution and continues its SMART work.

- When the disk completes, the disk ISR posts a disk event, activating and executing the command task, which pre-empts the background task. The command task finishes up the write command. It then pends on another command ready event, which has already been posted by the pre-processor task (so it does not become suspended). The command task continues, executing the new write command.

- The next thing that happens is there for illustration. Some (use your imagination) error occurs (maybe a servo bump something). The command task posts an event indicating this, is pre-empted by the exception task, the exception task activates and executes, handles the error, and then pends again on an exception event, and suspends.

- The command task resumes execution and starts the disk, pending on a disk event.

- Since the pre-processor task is pending on a command received event and there is none, it doesn't activate.

- The background task does resume execution, finishes its SMART work, and then has nothing else to do. It pends on some event and suspends. Since the background task is the lowest priority task, when it suspends the processor halts.

- Eventually, a disk interrupt will post a disk event, activating and executing the command task again (not shown).

3.3.11 Kernel Functions

The following functions are used by tasks to pend, post, suspend, etc:

- **eventControlIdent PendEvents (eventControlIdent eEvents)**
  This function causes the calling task to pend on the events specified in eEvents. The caller suspends if there were no posted events that were pended on for the task. If the task pends on events which were posted, this function will return those events.

  Note. Since this function enables interrupts, it MUST NOT be called from a critical section. Also, this function must not be called from an interrupt.

- **void PostEvents(eventControlIdent eEventId)**
  This function posts events. It can be called from either tasks or interrupts. More than one event ID bit can be provided in eEventId. If a higher priority task is pending on one of the events posted then the posting task will be preempted by that task. Note that this function enables interrupts so it must not be called from a critical section.

- **void ClearPostedEvents(eventControlIdent eEvents)**
  This function un-posts the events indicated in eEvents.

  Note. Since this function enables interrupts, it MUST NOT be called from a critical section.

- **eventControlIdent CheckPostedEvents(eventControlIdent eEvents)**
  This function returns events passed in the parameter if they are currently posted.

Chapter 4 Software Tools (Setup & Configuration)

4A Installation for Win 95

How to use this manual

This manual provides simple, step-by-step instructions that will guide you through the installation of the Atlas development tools.

System Requirements

A Pentium class 90 MHz with 32 Mb of RAM is required as a minimum. You must be running Win 95. Performance will vary depending on CPU speed and amount of RAM. The firmware build process is CPU and RAM limited. A Pentium Pro 200 MHz with 64 Mb of RAM will complete an Eclipse firmware build in sixteen minutes.

Network Requirements

Your system must have loaded Novell VLMs and be logged in to the home server using the Novell Directory Services (NDS) mode before the installation should be attempted. Your login ID must be a member of the Novell group "G-ENG-FWENG.ENG.MILP.QNTM" in order to receive access rights to the install directories. Your manager can add you to this group. Also, your login ID must be a member of your home server PVCS user group. This group will grant access rights and mappings to the INTERSOLV PVCS tools. Your manager can add you to this group.

Server	PVCS User Group Name
APE	Pvcs User.ENG.MILP.QNTM
BLUE LIGHT	G-BLUE-PVCS_USER.BLT.ENG.MILP.QNTM
LETHAL	G-LETHAL-PVCS_USER.FWE.ENG.MILP.QNTM
TITAN	G-TITAN-PVCS_USER.HCS.ENG.MILP.QNTM

Installation Sequence

The installation of the Atlas Firmware Development Environment follows a sequential order:

1. Obtaining access rights to servers
2. Drive Mapping
3. Intersolv Software (PVCS) Installation
4. Green Hills Software Installation
5. Codewright Software Installation
6. Template Icons Installation
7. Path Check
8. Firmware Build
9. Project.rc file Edit

4A.1 Obtaining Access Rights to Servers

Before installing through the network, check if you have access rights to:	Lethal\Vol2\PVCS	
	Ape\Vol1	(for access to Ape\Fwtools)
	Blue_light\Vol1	(for access to Eclipse)

*** If you do not have privileges to Lethal \vol2, check your server's volume 1 for PVCS privileges

*** Call the I/S responce center at **x6470**

*** Make sure I/S upgrades your network access to **Netware Directory Service (NDS)**. Check by viewing the contents of your network. To view your network, select the control panel through your Start\Settings menu then double-click the Network icon.

The Atlas firmware development system uses two of the Intersolv software packages, PVCS Configuration Builder and PVCS Version Manager. The current working PVCS install base is P:\ on your server. In order to use the tools, your manager must add your User ID to the LAF database.

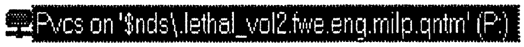
Your manager may add you to the database as follows:

- Run p:\cb\win95\admin\laf.exe
- A window should appear containing an icon labelled "Central"
- Double click the "Central" icon
- A window labelled "LAF Database Central: CENTRAL" should appear with a box labelled "Select Product"
- Click on the line containing "PVCS Configuration Builder" under "Product Name" which also contains "WIN95-WIN" under "Operating System"
- Click the "Licenses" pull-down menu, then select "Assign/Unassign Users"
- Enter the User ID in the box provided
- Click the "Add" button, then click the "OK" button
- In the window labelled "LAF Database Central: CENTRAL", click on the line containing "PVCS Version Manager" under "Product Name" and "WIN95-DOS" under "Operating System"
- Click the "Licenses"pull down menu, then select "Assign/Unassign Users"
- Enter the User ID in the box provided
- Click the "Add" button, then click the "OK" button

4A.2 Drive Mapping

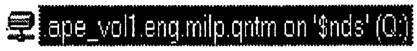
Since the tools for this installation are obtained through network servers, map your computer's network to:

P:\\Lethal\Vol2\PVCS



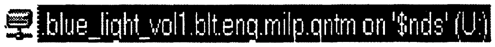
Pvcs on '$nds\.lethal_vol2.fwe.eng.milp.qntm' (P:)

Q:\\Ape\Vol1



.ape_vol1.eng.milp.qntm on '$nds' (Q:)

U:\\Blue_light\Vol1

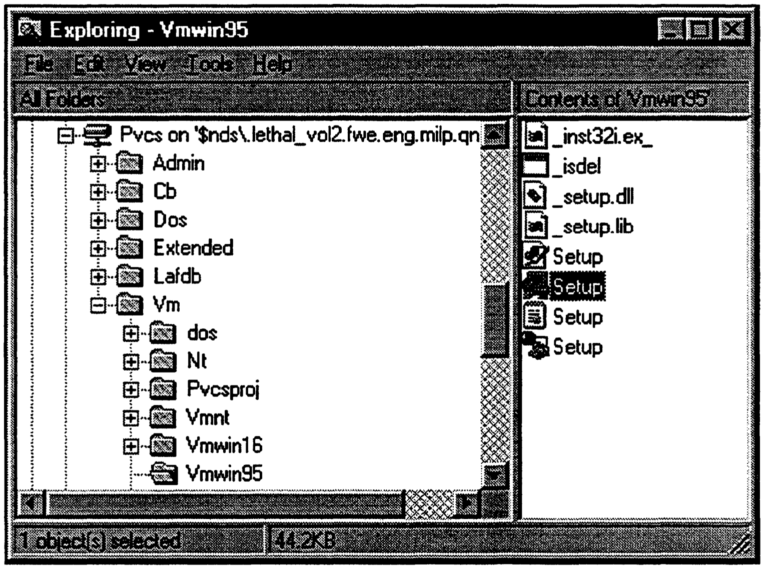


.blue_light_vol1.blt.eng.milp.qntm on '$nds' (U:)

4A.3 Intersolv Software (PVCS) Installation

4.3.1 Installing the Version Manager

Using Win95 Explorer, locate \\Lethal\Vol2\Pvcs\Vm\Vmwin95



Pvcs on '$nds\.lethal_vol2.fwe.eng.milp.qn
Admin
Cb
Dos
Extended
Lafdb
Vm
dos
Nt
Pvcsproj
Vmnt
Vmwin16
Vmwin95

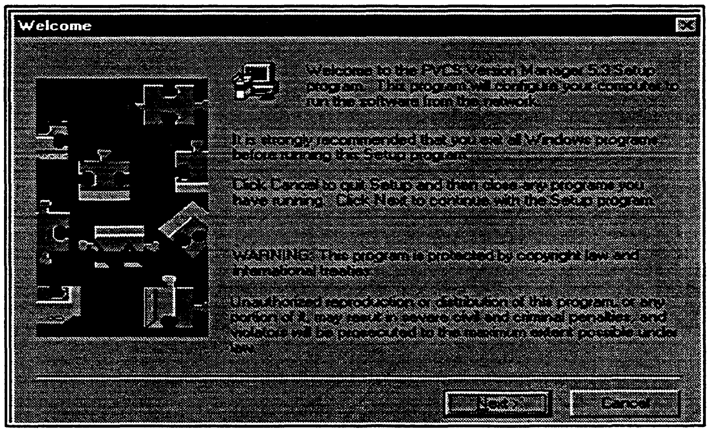
inst32i.ex
_isdel
_setup.dll
_setup.lib
Setup
Setup
Setup
Setup

1 object(s) selected
44.2KB

Double-click the **Setup** icon to open Intersolv Software for Version Manager Installation

On the **Welcome** (to the Version Manager) window,

- Press the **Next** button



Welcome to the PVCS Version Manager 5.3 Setup program. This program will configure your computer to run the software from the network.

It is strongly recommended that you exit all Windows programs before running this Setup program.

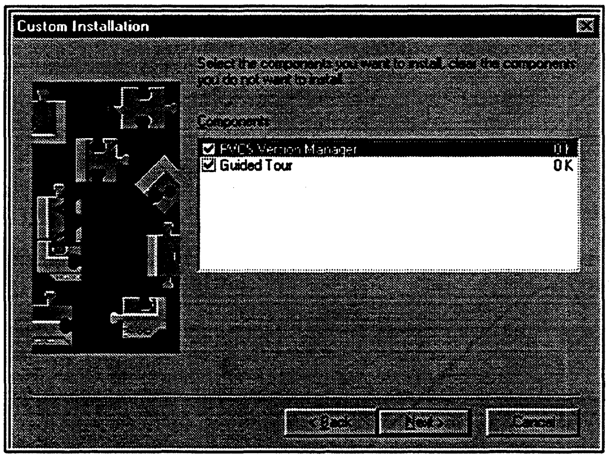
Click Cancel to quit Setup and then close any programs you have running. Click Next to continue with the Setup program.

WARNING: This program is protected by copyright law and international treaties.

Unauthorized reproduction or distribution of this program, or any portion of it, may result in severe civil and criminal penalties, and violators will be prosecuted to the maximum extent possible under law.

When a Custom Installation window prompts you
to Select the Products to Configure,

- Select at least the PVSC Version Manager then
- Press the **Next** button



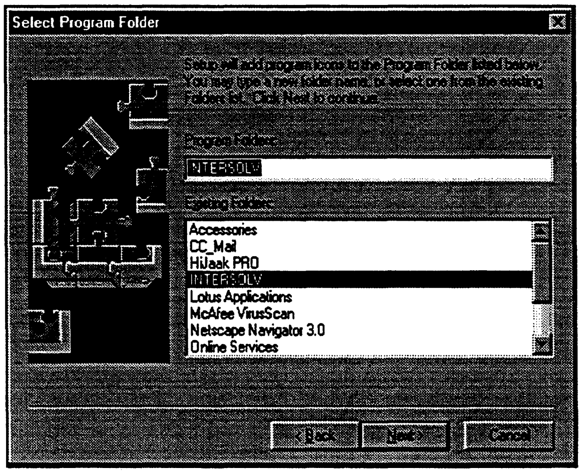
Select the components you want to install, clear the components you do not want to install.

Components:

☑ PVCS Version Manager 0 K
☑ Guided Tour 0 K

When the **Select Program Folder** window prompts you
to Specifiy the Installation Directory for Products,

- Press the **Next** button (this accepts the default directory)



Setup will add program icons to the Program Folder listed below. You may type a new folder name, or select one from the existing Folders list. Click Next to continue.

Program Folder:

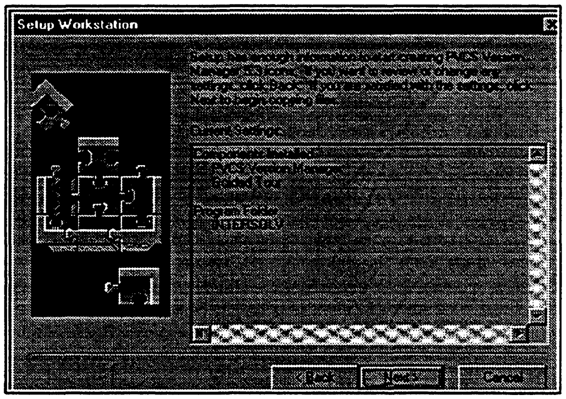
INTERSOLV

Existing Folders:

Accessories
CC_Mail
HiJaak PRO
INTERSOLV
Lotus Applications
McAfee VirusScan
Netscape Navigator 3.0
Online Services

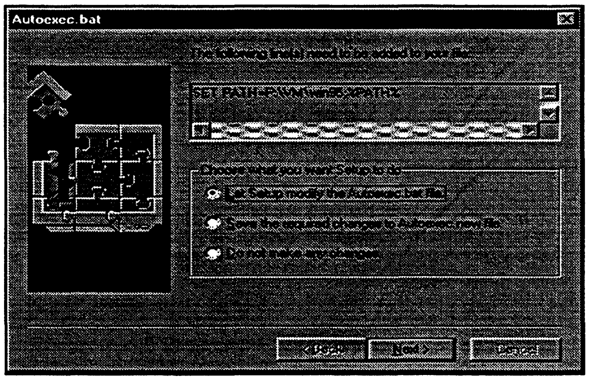
On **Setup Workstation window,**

- Press the **Next** button



On the Autoexe.bat window,

- Press the **Next** button



Choose what you want Setup to do:

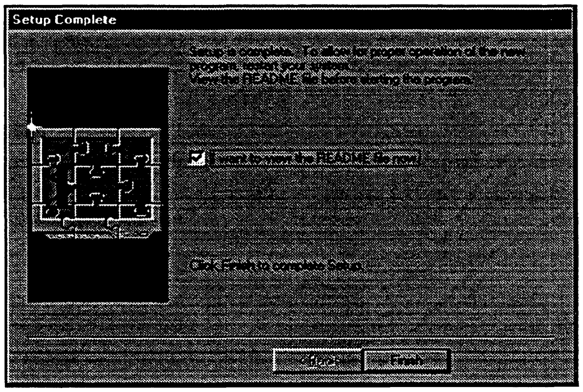
○ Let Setup modify the Autoexec.bat file

○ Save the required changes to Autoexec.new file

○ Do not make any changes.

On Setup Complete window,

- press the **Finish** button to complete Setup.



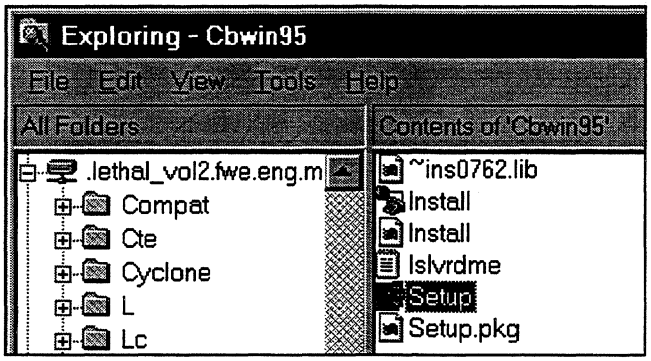
Setup is complete. To allow for proper operation of the new program, restart your system.
View the README file before starting the program.

☑ I want to view the README file now

Click Finish to complete Setup.

4.3.2 Installing the Configuration Builder

Using Win95 Explorer, locate \\Lethal\Vol2\Pvcs\Cb\Cbwin95



Exploring - Cbwin95

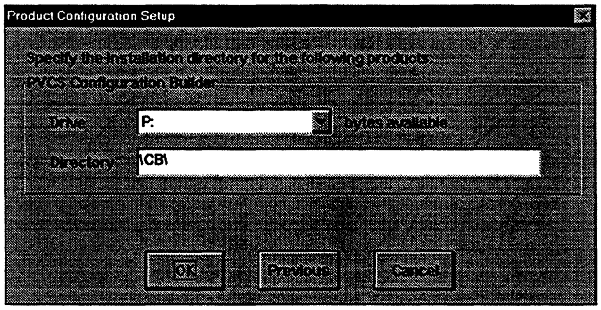
.lethal_vol2.fwe.eng.m
- Compat
- Cte
- Cyclone
- L
- Lc

~ins0762.lib
Install
Install
Islvrdme
Setup
Setup.pkg

Double-click the Setup icon to open Intersolv Software for Configuration Builder Installation

On the **Welcome** (to the Configuration Builder) window,
- Press the **OK** button

When the **Product Configuration Setup** window prompts you to Specify the Installation Directory for Products,
- Press the **OK** button (this accepts the default directory)



Product Configuration Setup

Specify the installation directory for the following products:

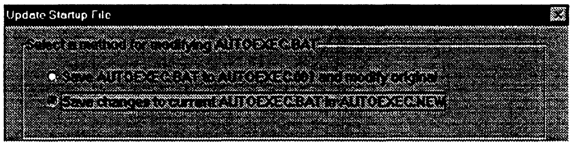
PVCS Configuration Builder

Drive: P: bytes available

Directory: \CB\

On the **Note** (about path statement) window,
- Press the **OK** button

When an **Update Startup File** window prompts you
to Select the method for modifying Autoexec.bat,
- Select "**Save Changes to Current Autoexec.bat in Autoexec.new**"
(This is not a default. You must choose the second option!!!)
then press the OK button



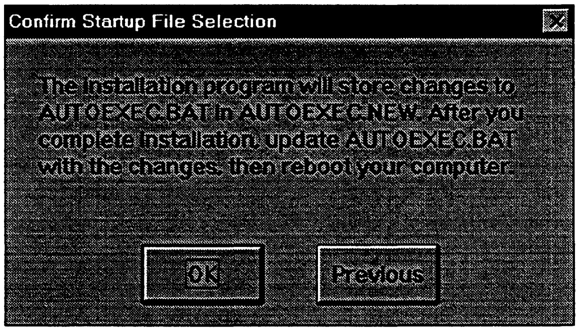
Update Startup File

Select a method for modifying AUTOEXEC.BAT

○ Save AUTOEXEC.BAT to AUTOEXEC.000 and modify original

● Save changes to current AUTOEXEC.BAT in AUTOEXEC.NEW

On the **Confirm Startup File Selection** window
- Press the OK button



Confirm Startup File Selection

The installation program will store changes to AUTOEXEC.BAT in AUTOEXEC.NEW. After you complete installation, update AUTOEXEC.BAT with the changes, then reboot your computer.

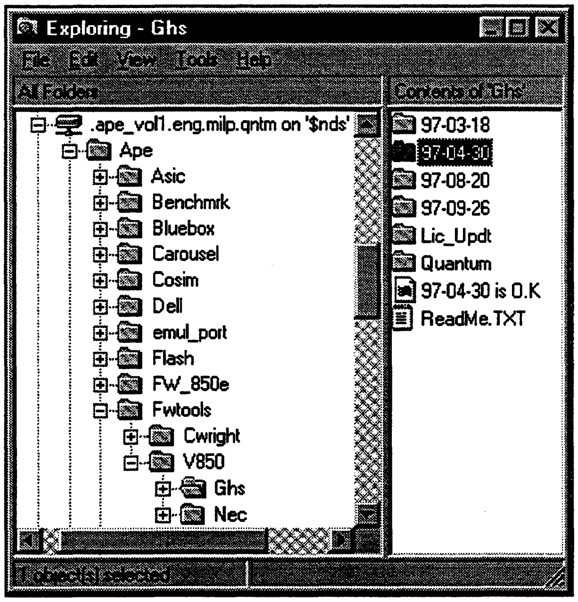
On the **Information** (to inform you that installation is complete) window,
- Press the OK button

On the **Question** window,
- Press the **Yes** button to view the readme file or
press the **No** button to end Intersolv Software Installation program.

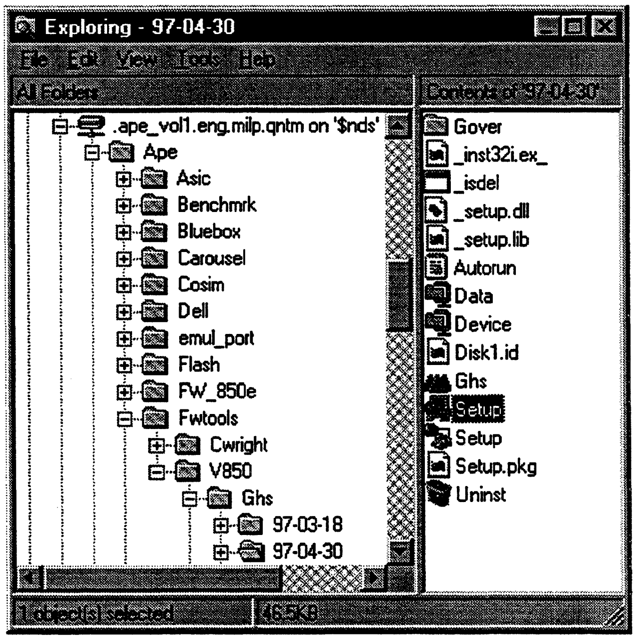
4A.4 Green Hills Software Installation

4.4.1 Installing Multi

Using Win95 Explorer, locate \\Ape\Vol1\Ape\Fwtools\V850\GHS



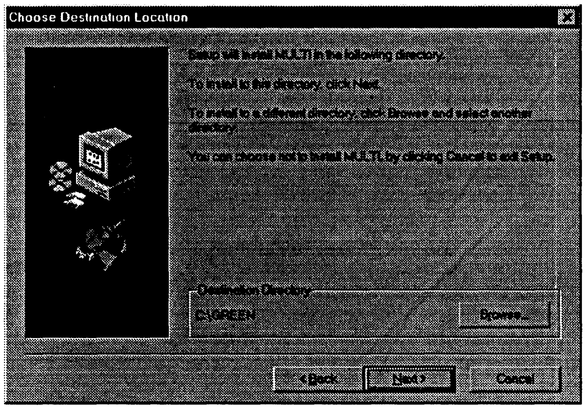
Open the Dated folder declared as 'OK'



Double-click the Setup icon to open the Green Hills Software program for Multi Setup

On the **Welcome** (to the MultiSetup Program) window,
- Press the **Next** button

On the **Choose Destination Location** window,
- Press the **Next** button again (this accepts the default location)



Setup will install MULTI in the following directory.

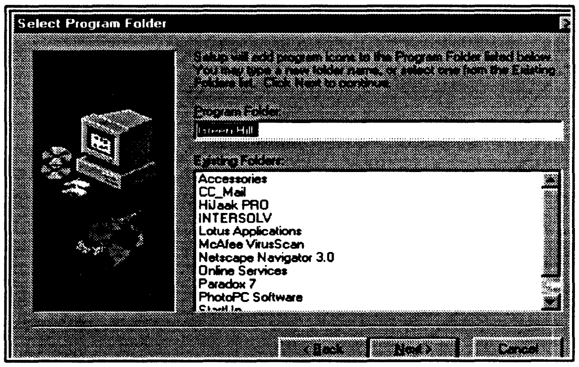
To install to this directory, click Next.

To install to a different directory, click Browse and select another directory.

You can choose not to install MULTI by clicking Cancel to exit Setup.

Destination Directory: C:\GREEN

On the **Select Program Folder** window,
- Press the **Next** button again (this accepts the default location)

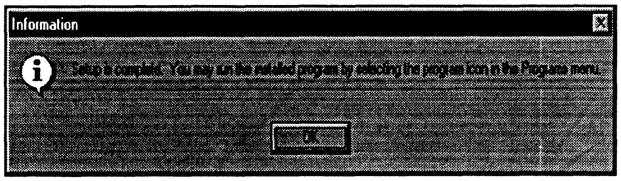


Setup will add program icons to the Program Folder listed below. You may type a new folder name, or select one from the Existing Folders list. Click Next to continue.

Program Folder: Green Hill

Existing Folders:
Accessories
CC_Mail
HiJaak PRO
INTERSOLV
Lotus Applications
McAfee VirusScan
Netscape Navigator 3.0
Online Services
Paradox 7
PhotoPC Software
StartUp

On the **Information** (to inform you that installation is complete) window,
- Press the **OK** button



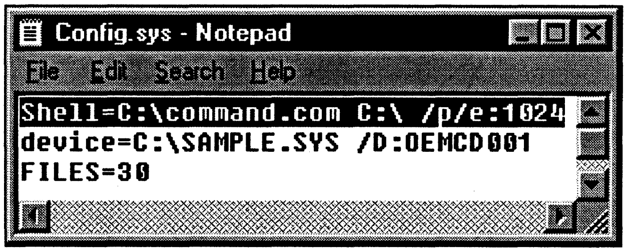
Setup is complete. You may run the installed program by selecting the program icon in the Programs menu.

4.4.2 Installing 850ice

Using an editor, modify the Config.sys file

- Add

  Shell=C:\command.com C:\ /p /e:1024



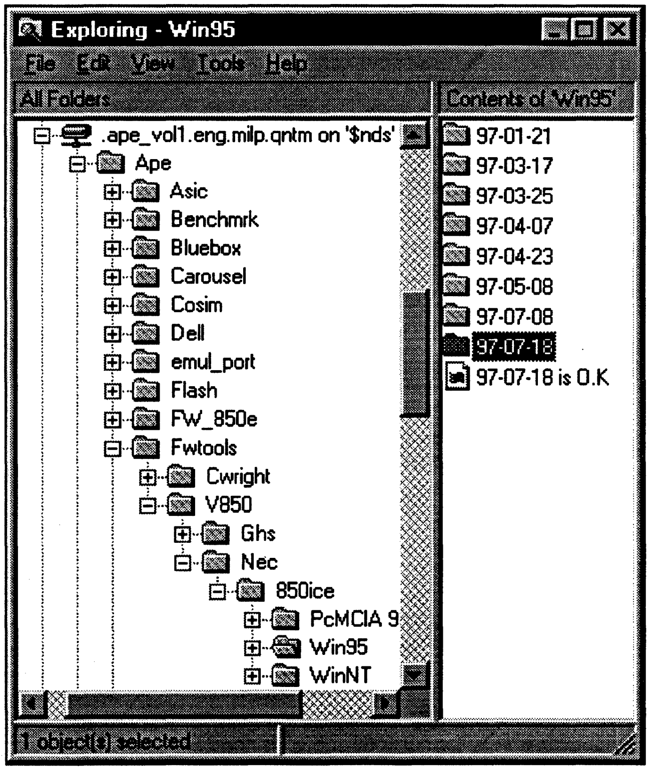
```
Shell=C:\command.com C:\ /p/e:1024
device=C:\SAMPLE.SYS /D:OEMCD001
FILES=30
```

**** Restart your computer**

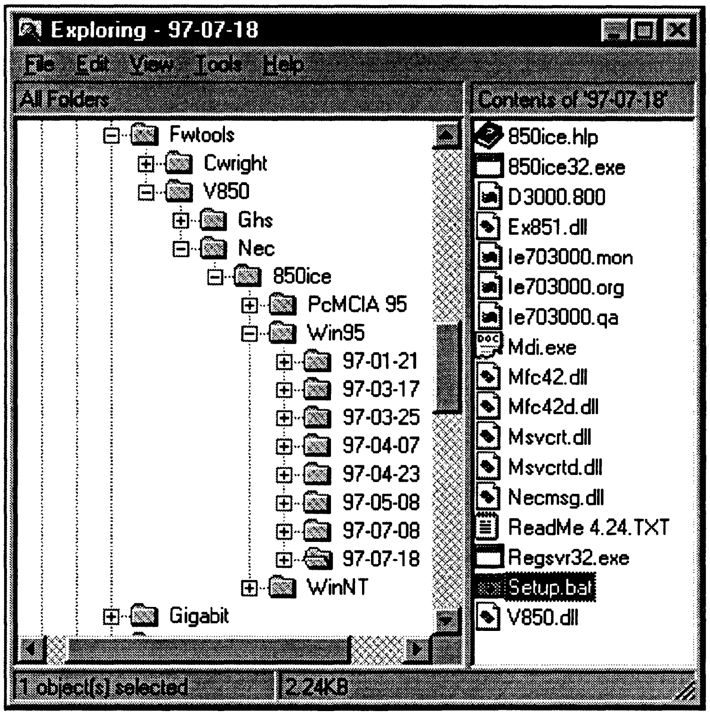
**** Make sure your computer's network is mapped to:**

P:\\Lethal\Vol2\PVCS
Q:\\Ape\Vol1
U:\\Blue_light\Vol1

Using Win95 Explorer, locate \\Ape\Vol1\Ape\Fwtools\V850\Nec\850ice\Win95



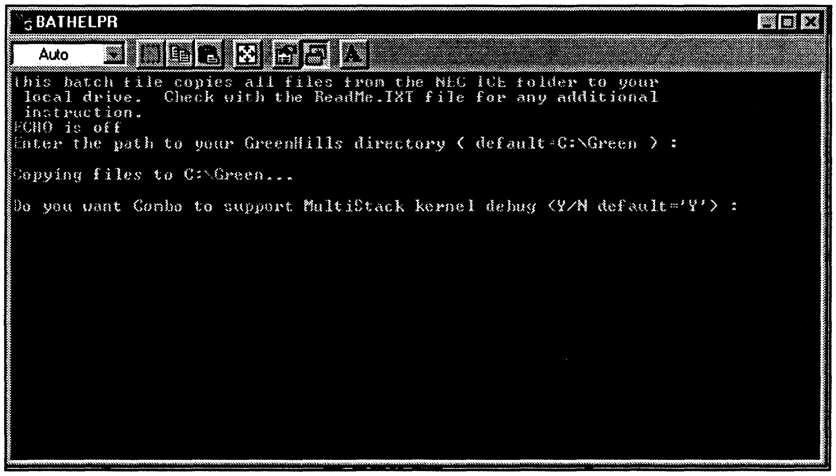
Open the Data folder declared as 'OK'



Double-click the Update icon to run the **Setup.bat** file

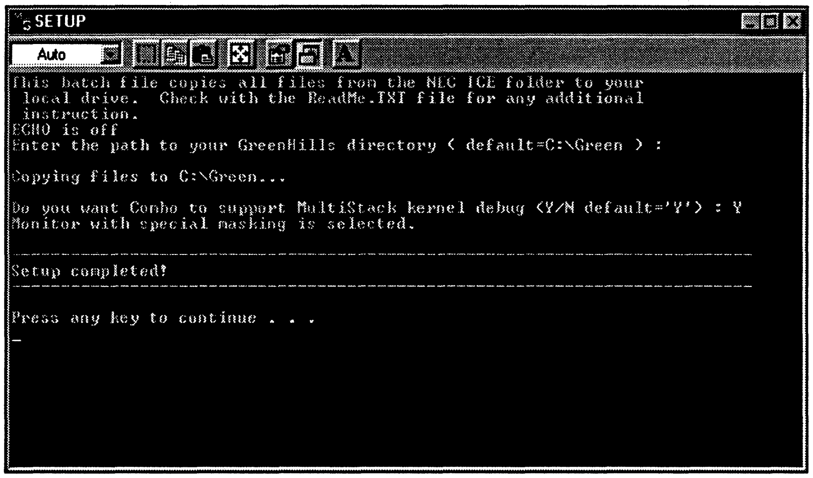
On the **BATHELPR** window,

- Press the **Enter** key to accept the first default path for the GreenHills directory
- Press the **Enter** key again to accept next default to support MultiStack kernal debug



```
This batch file copies all files from the NEC ICE folder to your
 local drive.  Check with the ReadMe.TXT file for any additional
 instruction.
ECHO is off
Enter the path to your GreenHills directory ( default=C:\Green ) :

Copying files to C:\Green...

Do you want Combo to support MultiStack kernel debug (Y/N default='Y') :
```

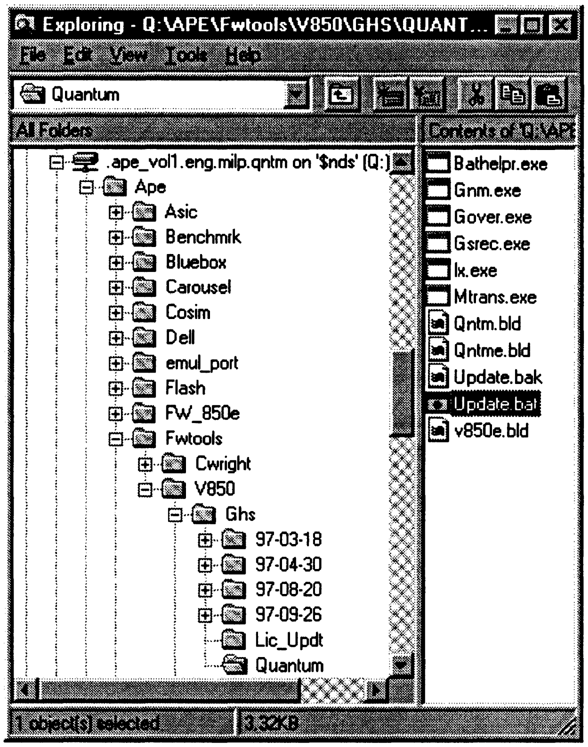
On the **Setup** window,
- Press **any key**
  then close the Setup window



```
This batch file copies all files from the NEC ICE folder to your
 local drive.  Check with the ReadMe.TXT file for any additional
 instruction.
ECHO is off
Enter the path to your GreenHills directory ( default=C:\Green ) :

Copying files to C:\Green...

Do you want Combo to support MultiStack kernel debug (Y/N default='Y') : Y
Monitor with special masking is selected.

--------------------------------------------------------------------------
Setup completed!
--------------------------------------------------------------------------

Press any key to continue . . .
_
```

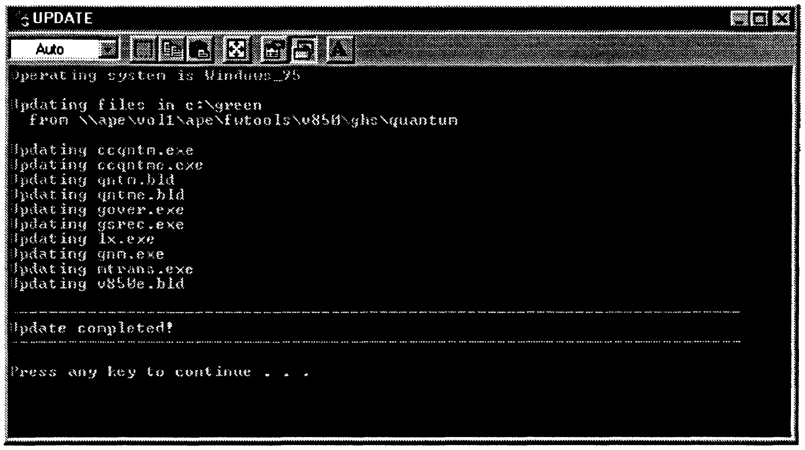
4.4.3 Updating Strings

Using Win 95 Explorer, locate
\\Ape\Vol1\Ape\Fwtools\GHS\V850\Quantum\Update.bat



Double-click the **Update.bat** icon to run the Update.bat file

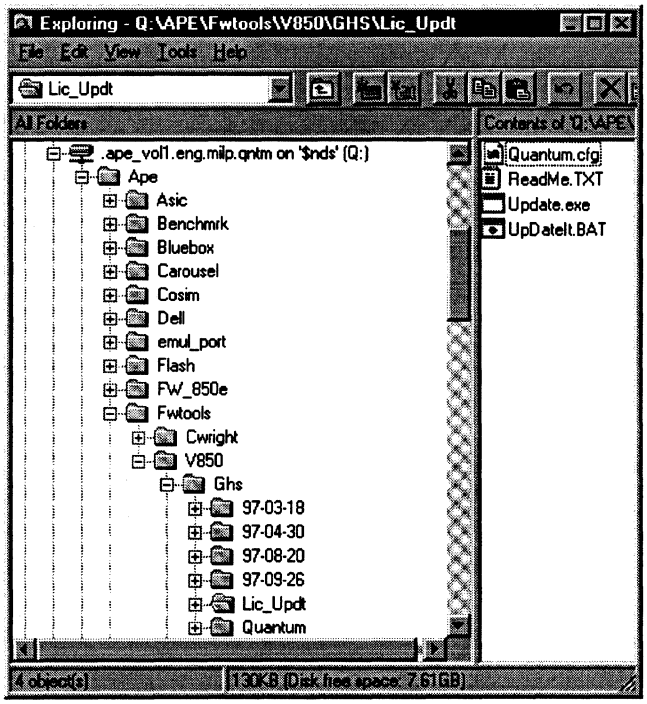
On the **Update** window,
- Press **any key**
then close the Update window



```
Operating system is Windows_95

Updating files in c:\green
  from \\ape\vol1\ape\fwtools\v850\ghs\quantum

Updating ccqntm.exe
Updating ccqntme.exe
Updating qntm.bld
Updating qntme.bld
Updating gover.exe
Updating gsrec.exe
Updating lx.exe
Updating gnm.exe
Updating mtrans.exe
Updating v850e.bld

------------------------------------------------------------------------
Update completed!
------------------------------------------------------------------------

Press any key to continue . . .
```

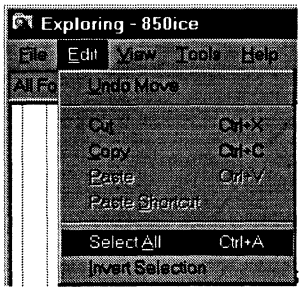
4.4.4 License Extension

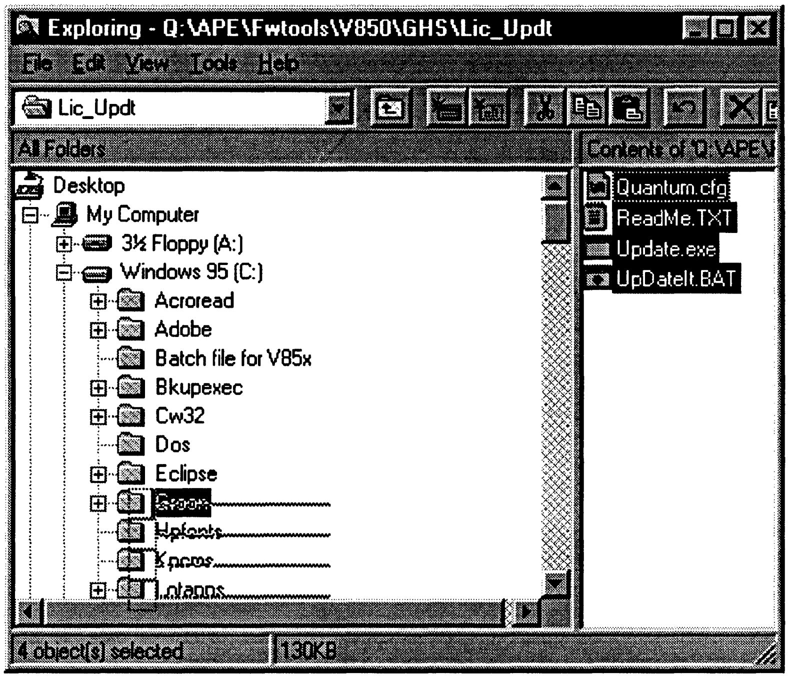
Using Win 95 Explorer, locate \\Ape\Vol1\Ape\Fwtools\V850\GHS\Lic_Updt



Double click on the **Lic_Updt** folder

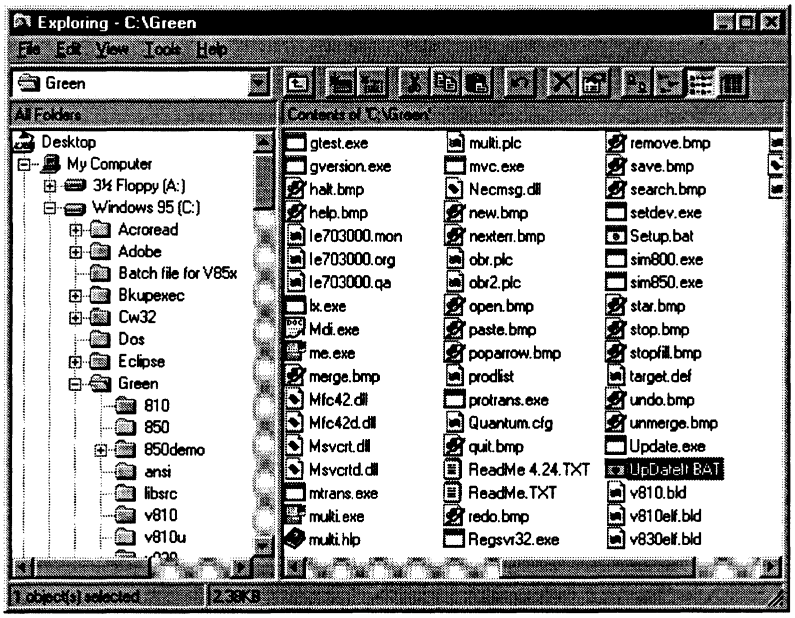
Select the **Edit\Select All** menu



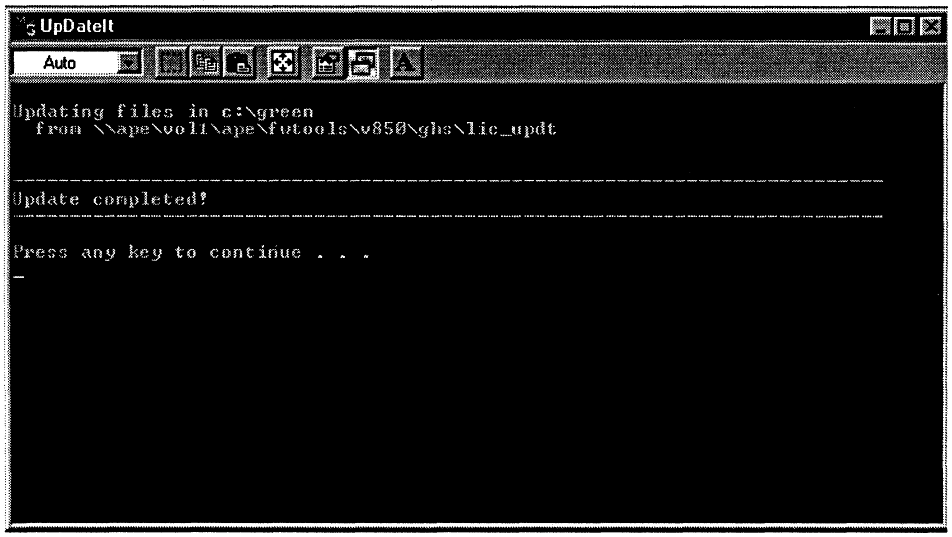


Click and drag the highlighted selection from the **Lic_Updt** folder to the **C:\Green** folder

Using Win 95, locate and open your local drive's **Green** folder



Double click on the **UpdateIt.bat** folder



```
Updating files in c:\green
  from \\ape\vol1\ape\fvtools\v850\ghs\lic_updt

--------------------------------------------------------------------------------
Update completed!
--------------------------------------------------------------------------------

Press any key to continue . . .
_
```

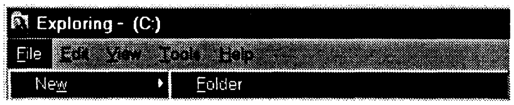
On the **Update** window,
- Press **any key**

then close the **Update** window

4.4.5 Shortcut to Multi

Using Win95 Explorer, create:

(1) an **Eclipse** folder in the root of the C drive,



(2) a **src** folder in the **Eclipse** folder,

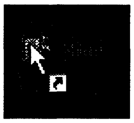
(3) an **at** folder in the **src** folder and/or

(4) a **scsi** folder in the **src** folder

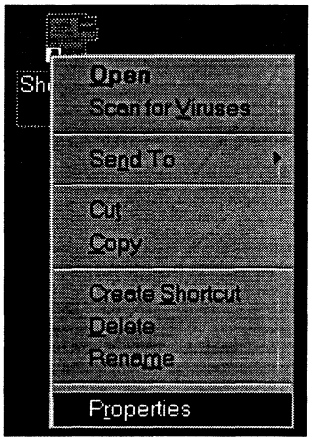
Note. The name you choose for you folders must be specific for your program. You may also want to create build/interface folders (i.e., E1AT, E2SC, ...)

Using Win95 Explorer, locate C:\Green\Multi.exe

Click and drag the Multi.exe file from the Green folder to the desktop



Select **Properties** from the Shortcut to Multi icon's pop-up menu



On the **shortcut to Multi Properties** window,
Click the **Shortcut** tab

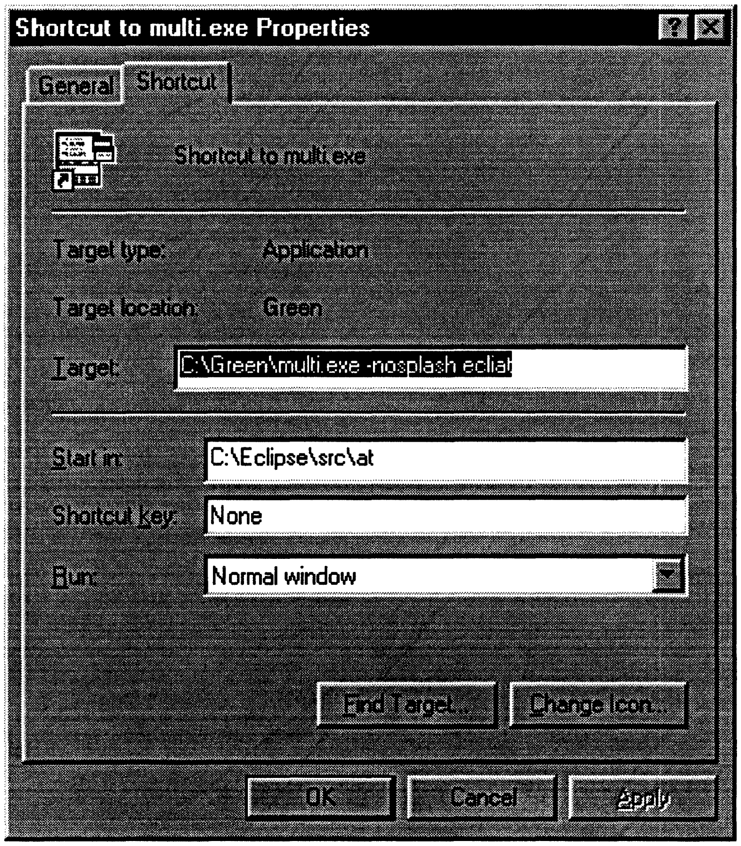
Within **Shortcut** section

On the Target Field
- Type **C:\Green\multi.exe -nosplash ecliat**

On the Start in Field
- Type either **C:\Eclipse\src\at**
or **C:\Eclipse\src\scsi**

(**Note.** Choose the appropriate path for your program.)

Press the **Apply** button and then press **OK**



Shortcut to multi.exe Properties

Target type: Application

Target location: Green

Target: C:\Green\multi.exe -nosplash ecliat

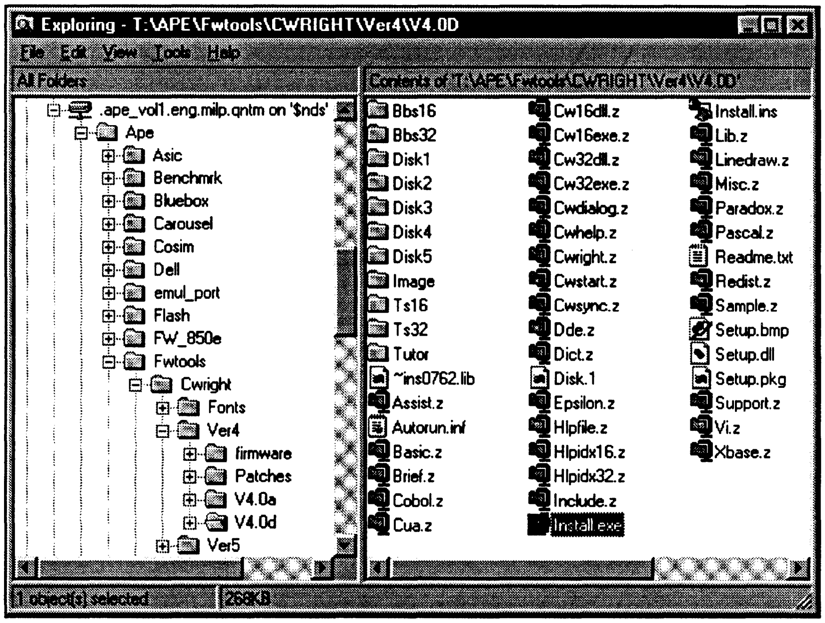
Start in: C:\Eclipse\src\at

Shortcut key: None

Run: Normal window

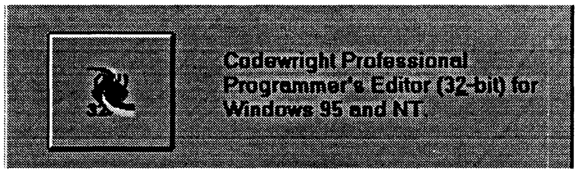
4A.5 Codewright Installation

Using Win95 Explorer, locate \\Ape\Vol1\Ape\Fwtools\Cwright\Ver4\V4.0d

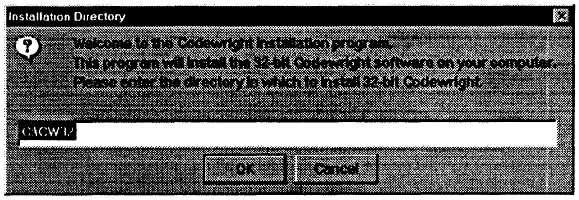


Double-click the Install.exe icon to start the Codewright Installation

When a **Codewright 4.0d Product Selection** window prompts you to select a product
- Press the **32-bit** button


32
Codewright Professional Programmer's Editor (32-bit) for Windows 95 and NT

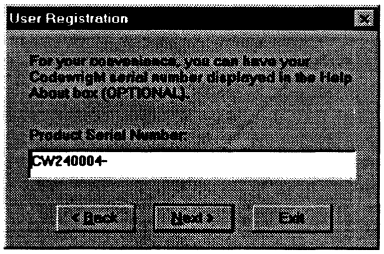
When an **Installation Directory** window prompts you to enter the directory for the installation,
- Press the **OK** button (this accepts the default location)


Installation Directory

Welcome to the Codewright installation program.
This program will install the 32-bit Codewright software on your computer.
Please enter the directory in which to install 32-bit Codewright.

C:\CW32

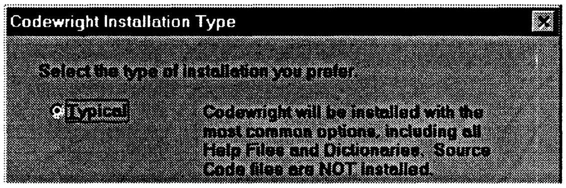
On the **User Registration** window,
- Press the **Next** button (this accepts the default selection)


User Registration

For your convenience, you can have your Codewright serial number displayed in the Help About box (OPTIONAL).

Product Serial Number: CW240004-

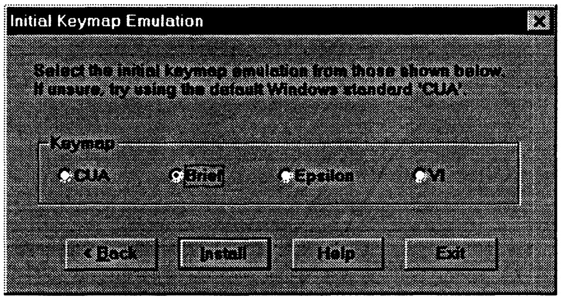
When a **Codewright Installation Type** window prompts you to select the type of installation you prefer.
- Select **Typical**
  then press the **Next** button


Codewright Installation Type

Select the type of installation you prefer.

● Typical — Codewright will be installed with the most common options, including all Help Files and Dictionaries. Source Code files are NOT installed.

When **Initial Keymap Emulation** window prompts you to select a keymap
- Select **Brief** or **CUA** (your choice)
  then press the **Install** button

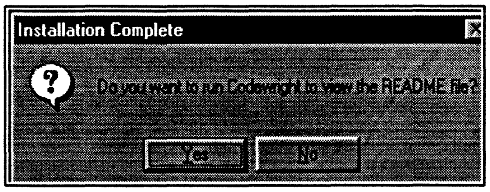

Initial Keymap Emulation

Select the initial keymap emulation from those shown below. If unsure, try using the default Windows standard 'CUA'.

Keymap: ○ CUA ● Brief ○ Epsilon ○ VI

On the **first Installation Complete** window,
- Press the **OK** button

On the **second Installation Complete** window, either
- Press the **Yes** button to view the readme file or
  press the **No** button to immediately view the installed items.

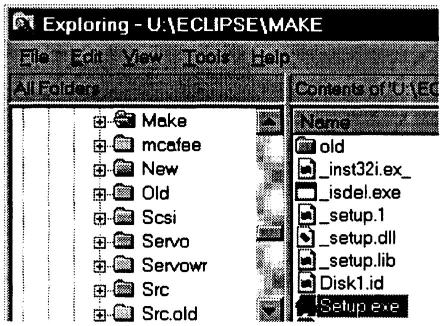

Installation Complete

Do you want to run Codewright to view the README file?

4A.6 Template Icons Installation

4.6.1 Installing Atlas Icons

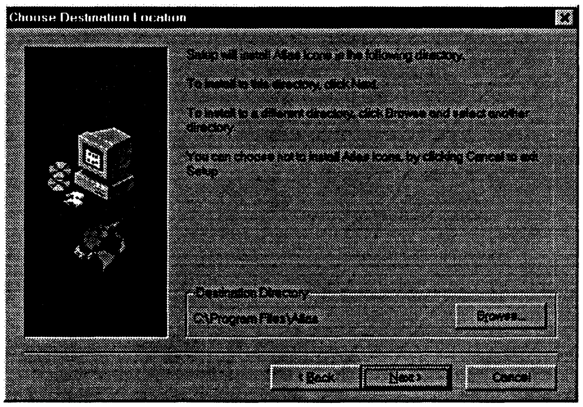
Using Win95 Explorer, locate \\Blue_light\Vol1\Eclipse\Make



Double-click the **Setup.exe** icon to start the Template Icons Installation

On the **Welcome** (to the Atlas Icon Setup) window,
- Press the **Next** button

When a **Choose Destination Location** window prompts you to Select the Products to Configure,
- Press the **Next** button (this accepts the default location)



Setup will install Atlas Icons in the following directory.

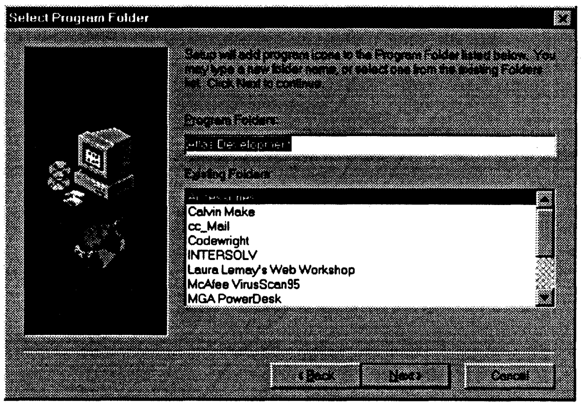
To install to this directory, click Next.

To install to a different directory, click Browse and select another directory.

You can choose not to install Atlas Icons, by clicking Cancel to exit Setup.

Destination Directory
C:\Program Files\Atlas

On the **Select Program Folder** window
- Press the **Next** button (this accepts the default folder)



Setup will add program icons to the Program Folder listed below. You may type a new folder name, or select one from the existing Folders list. Click Next to continue.

Program Folders:

Atlas Development

Existing Folders:

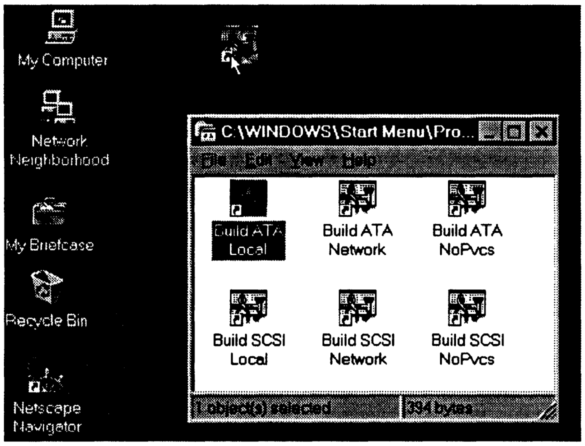
Accessories
Calvin Make
cc_Mail
Codewright
INTERSOLV
Laura Lemay's Web Workshop
McAfee VirusScan95
MGA PowerDesk

On the **Setup Complete** window,
- Press the **Finish** button to view the installed items

4.6.2 Shortcut to icon

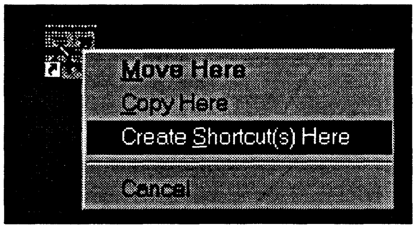
To access the program development icons from your desktop,

Select the **Build ATA Local** icon inside the Atlas Development window
Click and drag the icon out to the desktop by pressing the right button of your mouse



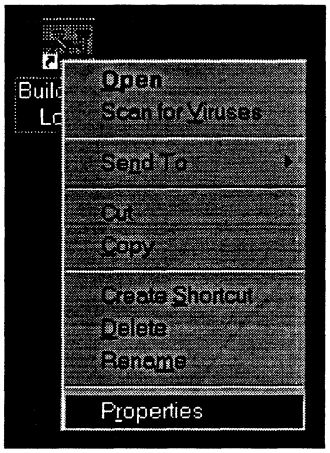
When a pop-up menu appears,

- Drag the mouse down to scroll to **Create Shortcut(s) Here**
  then press the left button of your mouse

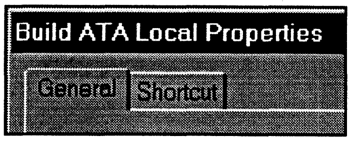


Select the **Build ATA Local** icon on the desktop with the right button of your mouse
When a pop-up menu appears,

- Drag the mouse down to scroll to **Properties**
  then press the left button of your mouse



On the **Build ATA Local Properties** window,
- Click the **Shortcut** tab



Within **Shortcut** section:

(1) Change the **Target** field information

- Replace Win with **Win95**
- Replace"project_ATA"=1 with **E4_ATA=1 NOLINT=1**
  Note. Remember to remove the " ".
  (See Build Options on page 6-3 for definitions of field information.)

- Replace makefile with **Eclipse.mak**

(2) Change the **Start in** field information

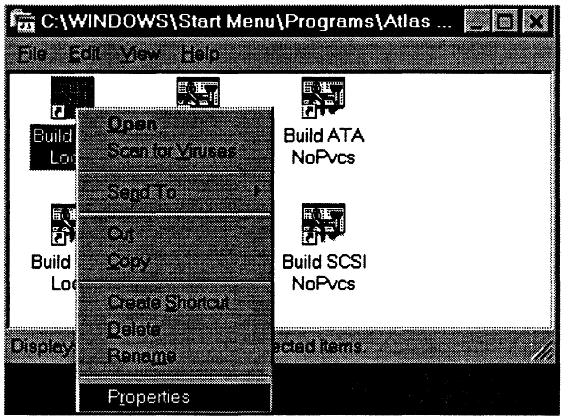
- Replace C:\PROGRA~1\ATLAS with **C:\Eclipse\src**

(3) Press the **OK** button

Note. To set the program development shortcuts within the Start/Programs menu

Select the **Build ATA Local** icon inside the Atlas Development window with the right button of your mouse

When a **pop-up menu** appears,
- Drag the mouse down to scroll to **Properties** then press the left button of your mouse



Follow the instructions in the beginning of this page

*** Apply the instructions in the **Shortcut to Icons** section to all the Atlas development icons.

4A.7 Path Check

*** Make sure your C:\ **Autoexec.bat** file contain the following paths!
If it does not,
- type in the missing paths using an editor.

```
:
:Environment Variables to support Eclipse code builds.
:

set path=u:\eclipse\utils;c:\green;p:\vm\dos;p:\cb\dos
set user=TYPE YOUR USER NAME HERE
set vcsid=%user%
set temp=c:\temp
set tmp=c:\temp

set serversrc=u:\eclipse\src
set vcscfg=%serversrc%\vcs.cfg
set project=eclipse


set workdir=u:\mr\transfer\ %user%

set iepath=c:\green
set islvini=c:\windows
set libpath=p:\vm\win95;p:\cb\win95
set server_path=u:\eclipse
```

*** **Note:** Make sure that you **TYPED YOUR USER NAME !!!!!**

*** **Note:** The workdir path, .../mr/..., used here is specific for the Tsunami program.
Use the workdir path specific for your program.

4A.8 Firmware Build

Before starting a build,

(1) Use the Win 95 explorer to:

(a) Create a Utils folder in the C drive's main folder (root directory)

(b) Drag the tm.exe. icon

from U:\\Blue_light\Vol1\Eclipse\Utils
to C:\\Utils

(2) Create an Eclipse.bat file, that contain:

attach blue_light\ TYPE YOUR USER NAME HERE
map u:=blue_light\vol1:

(3) Save the Eclipse.bat file in the C drive's main folder (root directory)

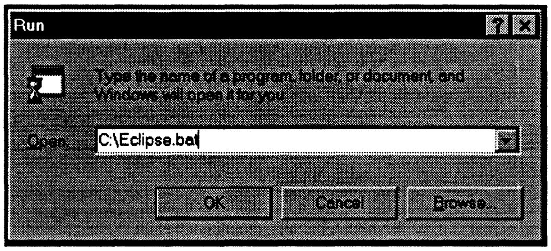
(4) Create a temp folder in the C drive's main folder (root directory))

**** Restart your computer**

**** Make sure your computer's network is mapped to:**

P:\\Lethal\Vol2\PVCS
Q:\\Ape\Vol1

(5) Run C:\Eclipse.bat



Run

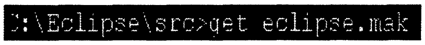
Type the name of a program, folder, or document, and Windows will open it for you.

Open: C:\Eclipse.bat

(6) Using MSDOS,

- Type **cd C:\eclipse\src** then press the Enter key

- Type **get eclipse.mak** then press the Enter key



```
C:\Eclipse\src>get eclipse.mak
```

Double-click one of the Build icons to start a build

4A.9 Project .rc file Edit

Using an editor, view the ".rc" file
(e.g., For the Eclipse AT project this file is c:\eclipse\src\at\ecliat.rc):

Make sure the file contains the following:

remote 850ICE**32**

If the file does not contain a "32" after the "remote 850ICE," use the editor to modify the file.

Note. The ".rc" file is a read-only file. Therefore, switch off the read-only setting of the file. To switch off the read-only setting:

Select the **ecliat.rc** icon with the right button of your mouse
When a pop-up menu appears,
- Drag the mouse down to scroll to **Properties** then press the left button of your mouse

On the **Ecliat.rc Properties** window,
Within **General** section

(1) Click the **Read-only** attribute to remove the check mark

(2) Press the **OK** button

Change a line to your project ".rc" file

Change from : remote 850ICE
to : remote 850ICE**32**

Note. After saving the file switch on the read-only setting of the file.

4A.10 Codewright Browser Installation (Optional)

To install a browser to Codewright , you must:

(1) Install Microsoft Visual C++ (version 2.0 or later)on you hard drive

(2) Delete the contents of your Src folder in C:\Eclipse

(3) Modify the Target field of the build icon that you will use

*Make sure your **Target** field contain the following path!*
If it does not, type in the missing path.

```
P:\CB\WIN95\BUILD.EXE E4_ATA=1 NOLINT=1
SBR=1 -script eclipse.mak makeall
```

(4) Modify the Autoexec.bat file

*Make sure your C:\**Autoexec.bat** file contain the following paths!*
If it does not, type in the missing paths using an editor.

```
set include=c:\msdev\include

set
path=u:\eclipse\utils;c:\green;p:\vm\win95;p:\cb\win95;c:\msdev\bin
```

(5) Change the Browser Database Field Information in Codewright's Project\Properties\Directories

*Make sure your **Browser Database** field contain the following path!*
If it does not, type in the missing path.

```
C:\Eclipse\src\ecliat.bsc
```

(6) Reboot your computer, run Eclipse.bat, get Eclipse.mak, then double-click the build icon that you will use

4B Installation for Windows NT 4.0

How to use this manual

This manual provides simple, step-by-step instructions that will guide you through the installation of the Atlas development tools.

System Requirements

A Pentium class 90 MHz with 32 Mb of RAM is required as a minimum. You must be running **Windows NT 4.0**. Performance will vary depending on CPU speed and amount of RAM. The firmware build process is CPU and RAM limited. A Pentium Pro 200 MHz with 64 Mb of RAM will complete an Eclipse firmware build in sixteen minutes.

Network Requirements

Your system must have loaded Novell VLMs and be logged in to the home server using the Novell Directory Services (NDS) mode before the installation should be attempted. Your login ID must be a member of the Novell group "G-ENG-FWENG.ENG.MILP.QNTM" in order to receive access rights to the install directories. Your manager can add you to this group. Also, your login ID must be a member of your home server PVCS user group. This group will grant access rights and mappings to the INTERSOLV PVCS tools. Your manager can add you to this group.

Server	PVCS User Group Name
APE	PVCS User.ENG.MILP.QNTM
BLUE LIGHT	G-BLUE-PVCS_USER.BLT.ENG.MILP.QNTM
LETHAL	G-LETHAL-PVCS_USER.FWE.ENG.MILP.QNTM
TITAN	G-TITAN-PVCS_USER.HCS.ENG.MILP.QNTM

Installation Sequence

The installation of the Atlas Firmware Development Environment follows a sequential order:

1. Obtaining access rights to servers
2. Drive Mapping
3. Intersolv Software (PVCS) Installation
4. Green Hills Software Installation
5. Codewright Software Installation
6. Template Icons Installation
7. Path Check
8. Firmware Build
9. Project.rc file Edit

4B.1 Obtaining Access Rights to Servers

Before installing through the network,
check if you have access rights to: Lethal\Vol2\PVCS
Ape\Vol1 (for access to Ape\Fwtools)
Blue_light\Vol1 (for access to Eclipse)

*** If you do not have privileges to Lethal \vol2, check your server's volume 1 for PVCS privileges

*** Call the I/S responce center at **x6470**

*** Make sure I/S upgrades your network access to **Netware Directory Service (NDS)**. Check by viewing the contents of your network. To view your network, select the control panel through your Start\Settings menu then double-click the Network icon.

The Atlas firmware development system uses two of the Intersolv software packages, PVCS Configuration Builder and PVCS Version Manager. The current working PVCS install base is P:\ on your server. In order to use the tools, your manager must add your User ID to the LAF database.

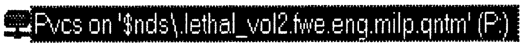
Your manager may add you to the database as follows:

- Run p:\cb\win95\admin\laf.exe
- A window should appear containing an icon labelled "Central"
- Double click the "Central" icon
- A window labelled "LAF Database Central: CENTRAL" should appear with a box labelled "Select Product"
- Click on the line containing "PVCS Configuration Builder" under "Product Name" which also contains "WIN95-WIN" under "Operating System"
- Click the "Licenses" pull-down menu, then select "Assign/Unassign Users"
- Enter the User ID in the box provided
- Click the "Add" button, then click the "OK" button
- In the window labelled "LAF Database Central: CENTRAL", click on the line containing "PVCS Version Manager" under "Product Name" and "WIN95-DOS" under "Operating System"
- Click the "Licenses"pull down menu, then select "Assign/Unassign Users"
- Enter the User ID in the box provided
- Click the "Add" button, then click the "OK" button

4B.2 Drive Mapping

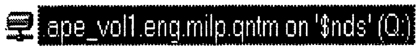
Since the tools for this installation are obtained through network servers,
map your computer's network to:

P:\\Lethal\Vol2\PVCS



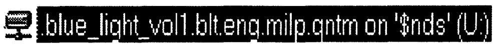
Pvcs on '$nds\.lethal_vol2.fwe.eng.milp.qntm' (P:)

Q:\\Ape\Vol1



.ape_vol1.eng.milp.qntm on '$nds' (Q:)

U:\\Blue_light\Vol1

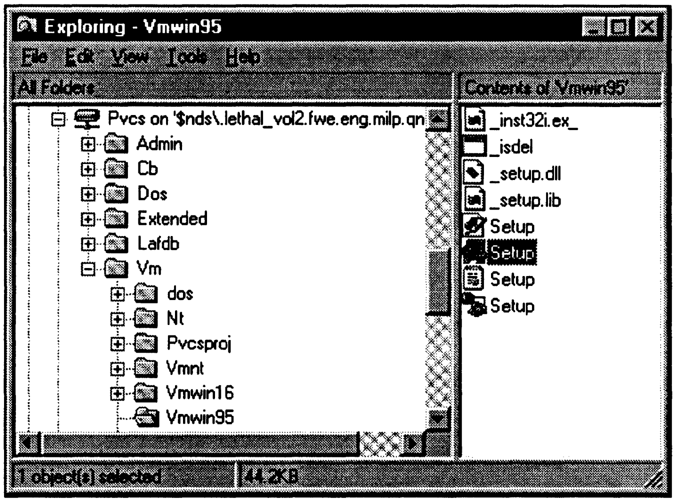


.blue_light_vol1.blt.eng.milp.qntm on '$nds' (U:)

4B.3 Intersolv Software (PVCS) Installation

4.3.1 Installing the Version Manager

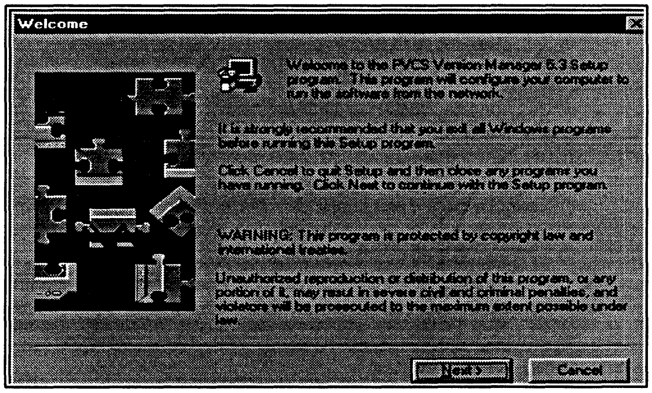
Using the Windows Explorer, locate \\Lethal\Vol2\Pvcs\Vm\Vmnt



Double-click the **Setup** icon to open Intersolv Software for Version Manager Installation

On the **Welcome** (to the Version Manager) window,

- Press the **Next** button



Welcome to the PVCS Version Manager 5.3 Setup program. This program will configure your computer to run the software from the network.

It is strongly recommended that you exit all Windows programs before running this Setup program.

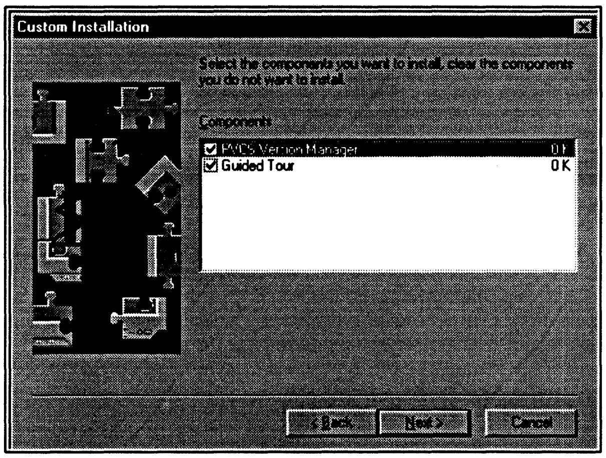
Click Cancel to quit Setup and then close any programs you have running. Click Next to continue with the Setup program.

WARNING: This program is protected by copyright law and international treaties.

Unauthorized reproduction or distribution of this program, or any portion of it, may result in severe civil and criminal penalties, and violators will be prosecuted to the maximum extent possible under law.

When a Custom Installation window prompts you
to Select the Products to Configure,

- Select at least the PVSC Version Manager then
- Press the Next button



Custom Installation

Select the components you want to install, clear the components you do not want to install.

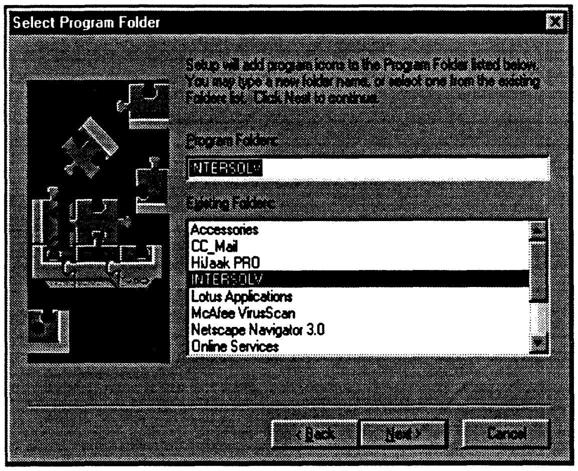
Components:

☑ PVCS Version Manager OK

☑ Guided Tour OK

When the **Select Program Folder** window prompts you
to Specifiy the Installation Directory for Products,

- Press the **Next** button (this accepts the default directory)



Select Program Folder

Setup will add program icons to the Program Folder listed below. You may type a new folder name, or select one from the existing Folders list. Click Next to continue.

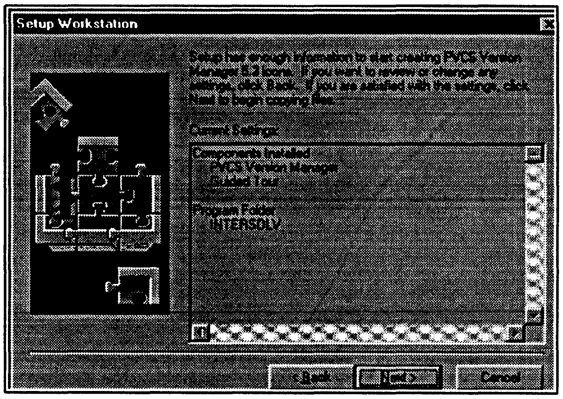
Program Folders:

INTERSOLV

Existing Folders:

Accessories
CC_Mail
HiJaak PRO
INTERSOLV
Lotus Applications
McAfee VirusScan
Netscape Navigator 3.0
Online Services

On **Setup Workstation window,**
- Press the **Next** button



Setup Workstation

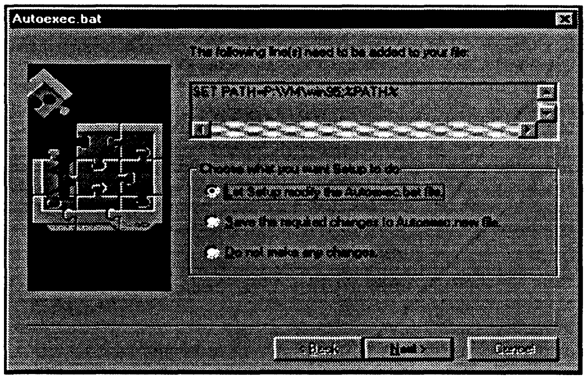
Setup has enough information to start copying PVCS Version Manager 5.3 icons. If you want to review or change any settings, click Back. If you are satisfied with the settings, click Next to begin copying files.

Current Settings:

Components Installed
PVCS Version Manager
Guided Tour

Program Folder
INTERSOLV

On the Autoexe.bat window,
- Press the **Next** button



Autoexec.bat

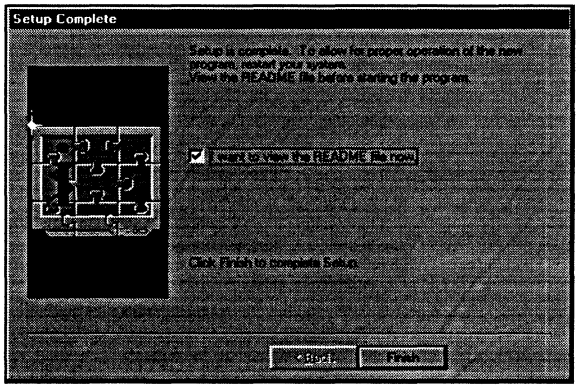
The following line(s) need to be added to your file:

SET PATH=P:\VM\win95;%PATH%

Choose what you want Setup to do

● Let Setup modify the Autoexec.bat file.
○ Save the required changes to Autoexec.new file.
○ Do not make any changes.

On Setup Complete window,
- press the **Finish** button to complete Setup.



Setup Complete

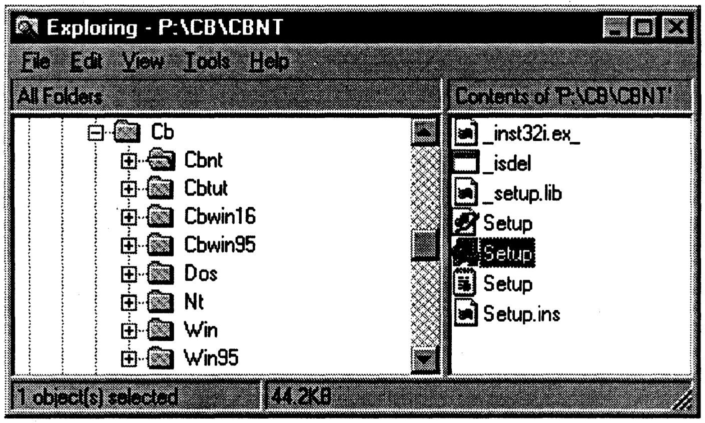
Setup is complete. To allow for proper operation of the new program, restart your system.
View the README file before starting the program.

☑ I want to view the README file now.

Click Finish to complete Setup.

4.3.2 Installing the Configuration Builder

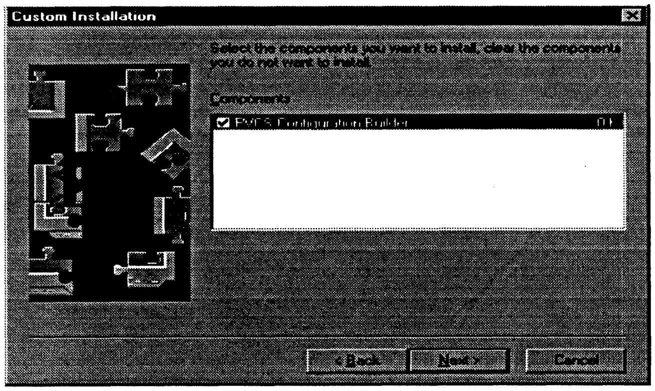
Using the Windows Explorer, locate \\Lethal\Vol2\Pvcs\Cb\Cbnt



Double-click the Setup icon to open Intersolv Software for Configuration Builder Installation

On the **Welcome** (to the Configuration Builder) window,
- Press the **Next** button

When a Custom Installation window prompts you to Select the Products to Configure,
- Select at least the **PVCS Configuration Builder** then Press the **Next** button

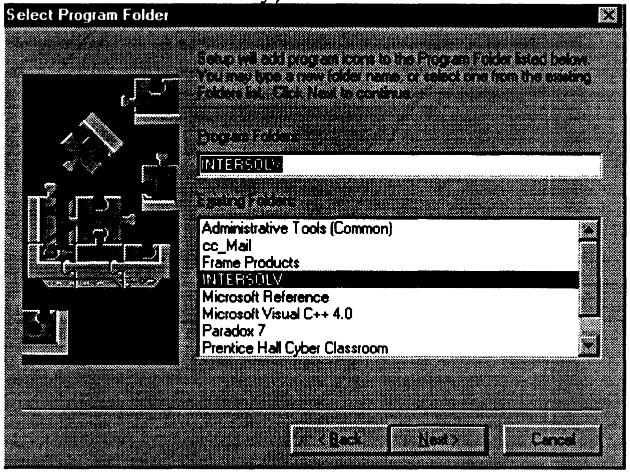

Custom Installation

Select the components you want to install, clear the components you do not want to install.

Components

☑ PVCS Configuration Builder

When the Select Program Folder window prompts you
to Specify the Installation Directory for Products,
- Press the **Next** button (this accepts the default directory)



Select Program Folder

Setup will add program icons to the Program Folder listed below. You may type a new folder name, or select one from the existing Folders list. Click Next to continue.

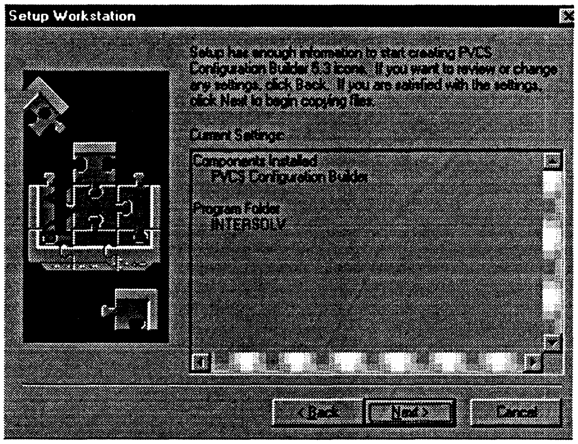
Program Folders:

INTERSOLV

Existing Folders:

Administrative Tools (Common)
cc_Mail
Frame Products
INTERSOLV
Microsoft Reference
Microsoft Visual C++ 4.0
Paradox 7
Prentice Hall Cyber Classroom

On the **Setup Workstation** window ,
- Press the **Next** button



Setup Workstation

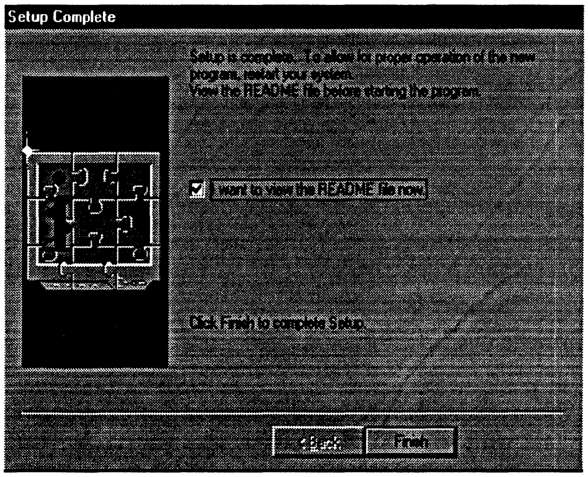
Setup has enough information to start creating PVCS Configuration Builder 5.3 icons. If you want to review or change any settings, click Back. If you are satisfied with the settings, click Next to begin copying files.

Current Settings:

Components Installed
PVCS Configuration Builder

Program Folder
INTERSOLV

On the **Setup Complete** window,
- Press the **Finish** button



Setup Complete

Setup is complete. To allow for proper operation of the new program, restart your system.
View the README file before starting the program.

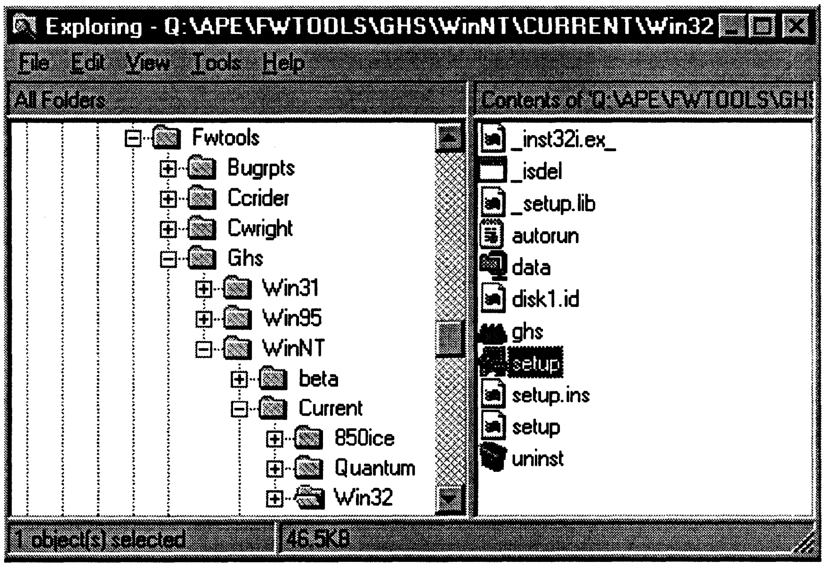
☑ I want to view the README file now.

Click Finish to complete Setup.

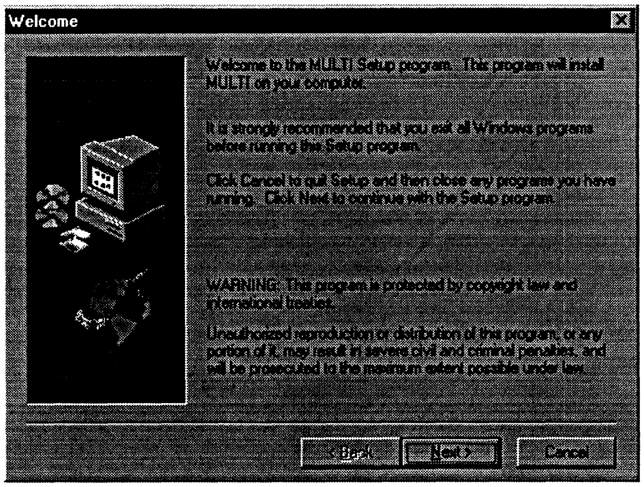
4B.4 Green Hills Software Installation

4.4.1 Installing Multi

Using the Windows Explorer, locate \\Ape\Vol1\Ape\Fwtools\GHS\Winnt\Current\Win32



Double-click the **Setup icon** to open the Green Hills Software program for Multi Setup



Welcome to the MULTI Setup program. This program will install MULTI on your computer.

It is strongly recommended that you exit all Windows programs before running this Setup program.

Click Cancel to quit Setup and then close any programs you have running. Click Next to continue with the Setup program.

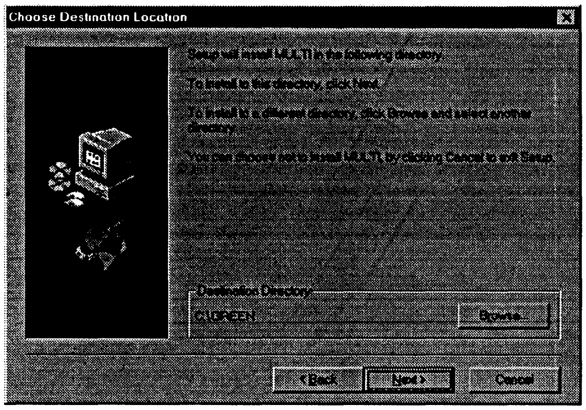
WARNING: This program is protected by copyright law and international treaties.

Unauthorized reproduction or distribution of this program, or any portion of it, may result in severe civil and criminal penalties, and will be prosecuted to the maximum extent possible under law.

On the **Welcome** (to the Multi Setup Program) window,

- Press the Next button

On the **Choose Destination Location** window,
- Press the **Next** button again (this accepts the default location)



Choose Destination Location

Setup will install MULTI in the following directory.

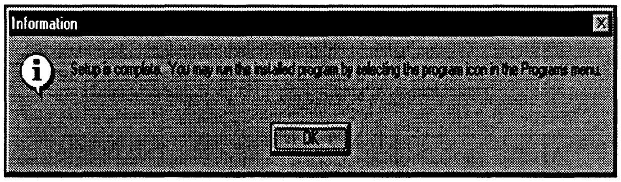
To install to this directory, click Next.

To install to a different directory, click Browse and select another directory.

You can choose not to install MULTI by clicking Cancel to exit Setup.

Destination Directory: C:\GREEN

On the **Information** (to inform you that installation is complete) window,
- Press the **OK** button

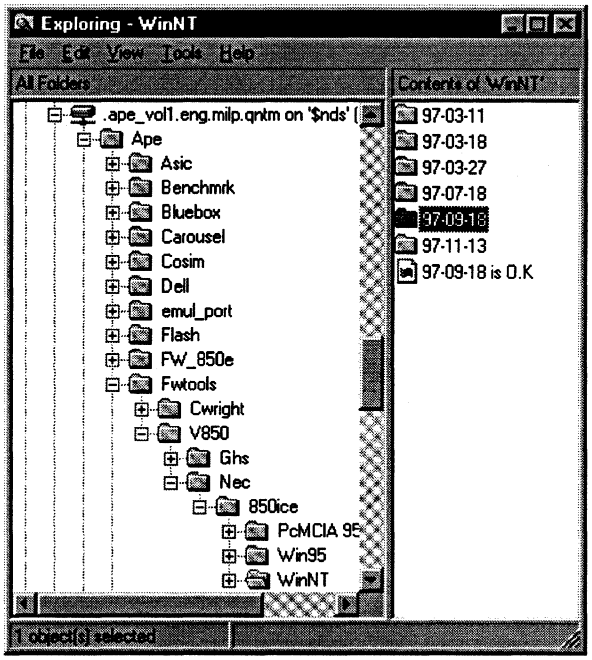


Information

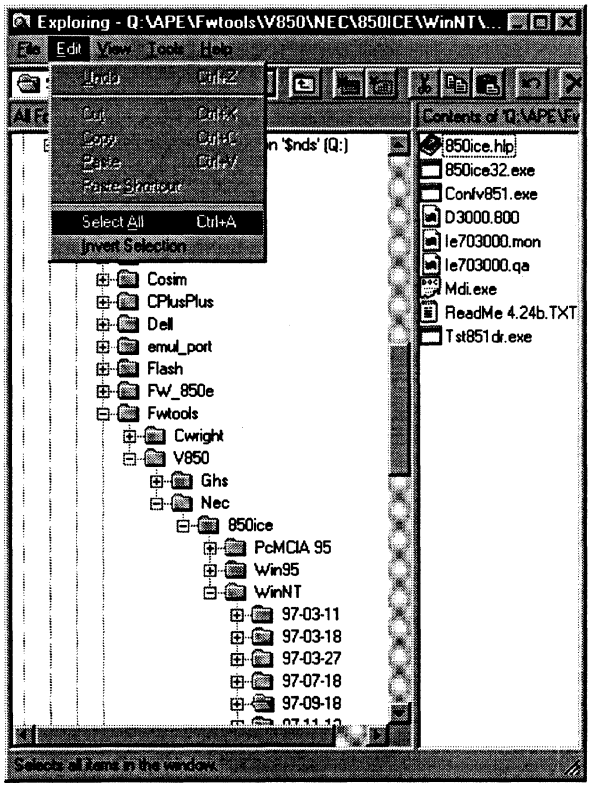
Setup is complete. You may run the installed program by selecting the program icon in the Programs menu.

4.4.2 Installing 850ice

Using Win95 Explorer, locate \\Ape\Vol1\Ape\Fwtools\V850\Nec\850ice\WinNT



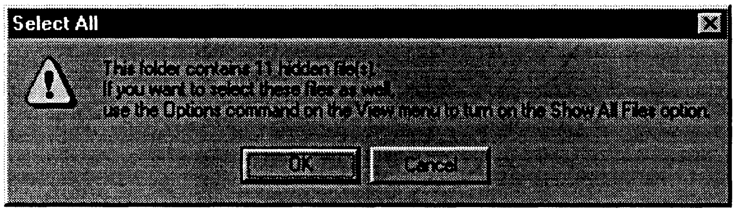
Open the Data folder declared as 'OK'



Select the Edit\SelectAll menu

Note. The **Select all** window appears if the Explorer window's View\Options is not set to **show all files.**

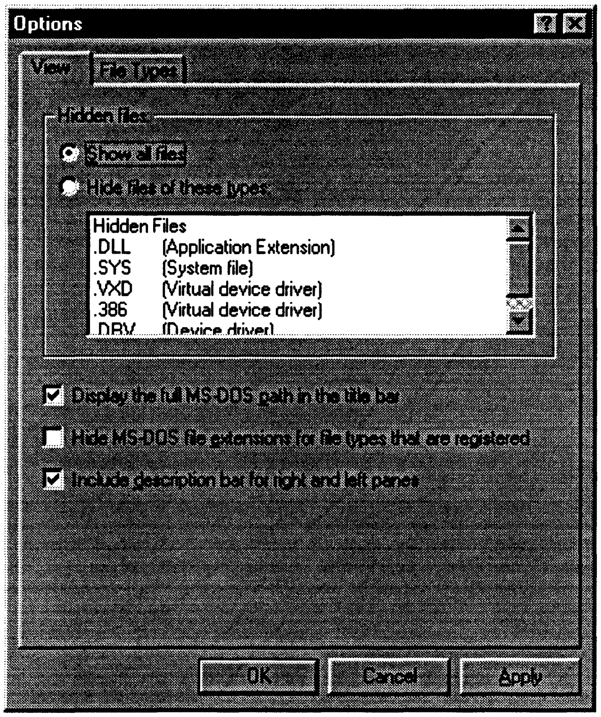
If the **Select All** window appears,
- Press the **OK** button then



Select All

This folder contains 11 hidden file(s).
If you want to select these files as well,
use the Options command on the View menu to turn on the Show All Files option.

On the **Explorer window**
- Select the Views/Options menu



Options

Hidden files:

● Show all files
○ Hide files of these types:

```
Hidden Files
.DLL    (Application Extension)
.SYS    (System file)
.VXD    (Virtual device driver)
.386    (Virtual device driver)
.DRV    (Device driver)
```

☑ Display the full MS-DOS path in the title bar

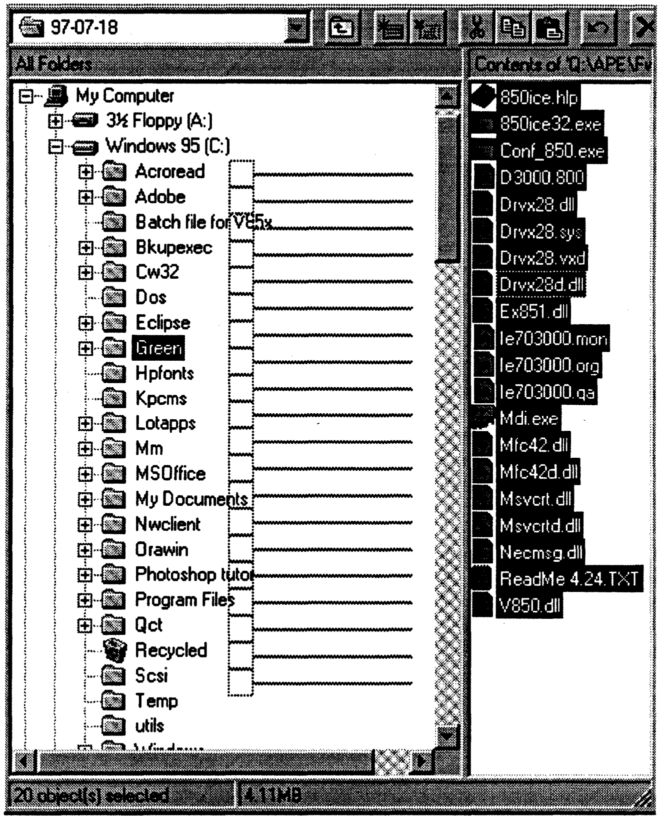
☐ Hide MS-DOS file extensions for file types that are registered

☑ Include description bar for right and left panes

On the **Options** window
- Select the **Show all files** radial button
- Press **Apply** button
  then press the **OK** button

Select the Edit/Select All menu again

If the Select All window does not appear,
Click and drag the highlighted selection from the 850ice folder to the C:\Green folder



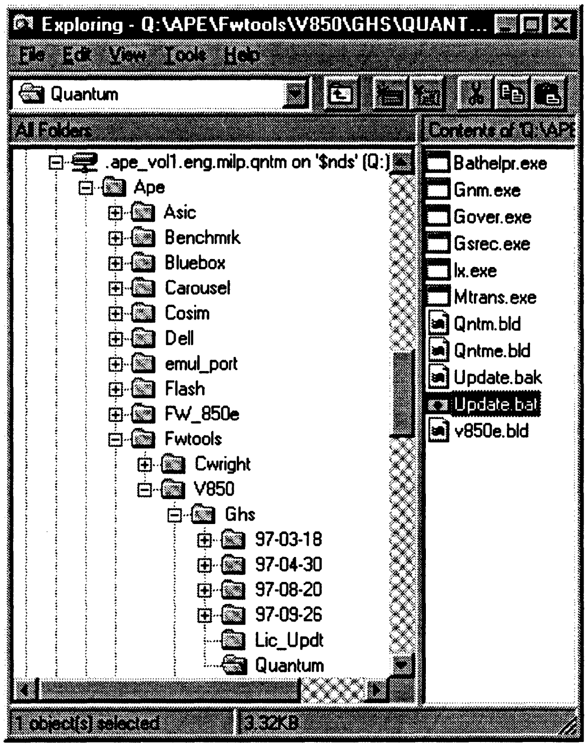
My Computer
3½ Floppy (A:)
Windows 95 (C:)
Acroread
Adobe
Batch file for V85x
Bkupexec
Cw32
Dos
Eclipse
Green
Hpfonts
Kpcms
Lotapps
Mm
MSOffice
My Documents
Nwclient
Orawin
Photoshop tutor
Program Files
Qct
Recycled
Scsi
Temp
utils

850ice.hlp
850ice32.exe
Conf_850.exe
D3000.800
Drvx28.dll
Drvx28.sys
Drvx28.vxd
Drvx28d.dll
Ex851.dll
Ie703000.mon
Ie703000.org
Ie703000.qa
Mdi.exe
Mfc42.dll
Mfc42d.dll
Msvcrt.dll
Msvcrtd.dll
Necmsg.dll
ReadMe 4.24.TXT
V850.dll

20 object(s) selected
4.11MB

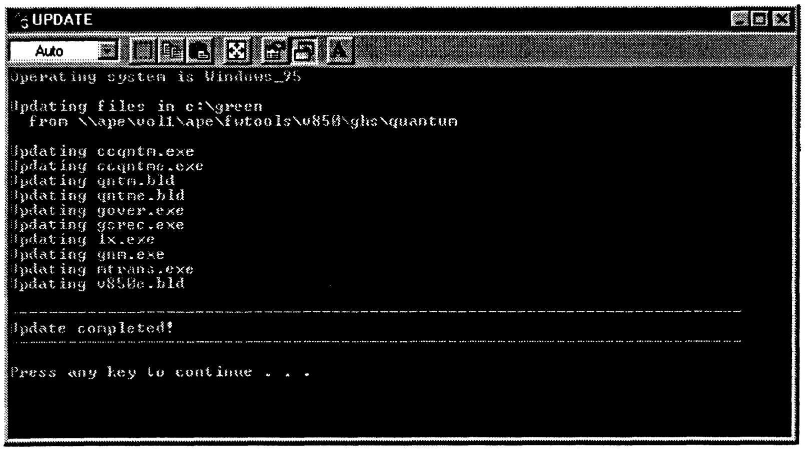
4.4.3 Updating Strings

Using Win 95 Explorer, locate
\\Ape\Vol1\Ape\Fwtools\GHS\V850\Quantum\Update.bat



Double-click the **Update.bat** icon to run the Update.bat file

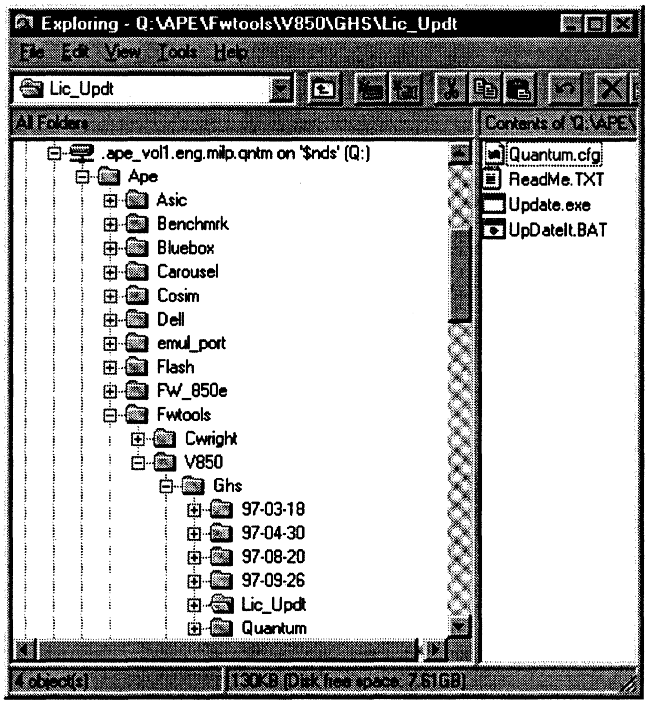
On the **Update** window,
- Press **any key**
  then close the Update window



```
Operating system is Windows_95

Updating files in c:\green
  from \\ape\vol1\ape\fwtools\v850\ghs\quantum

Updating ccqntm.exe
Updating ccqntme.exe
Updating qntm.bld
Updating qntme.bld
Updating gover.exe
Updating gsrec.exe
Updating lx.exe
Updating gnm.exe
Updating mtrans.exe
Updating v850e.bld

-----------------------------------------------------------------------------
Update completed!
-----------------------------------------------------------------------------

Press any key to continue . . .
```

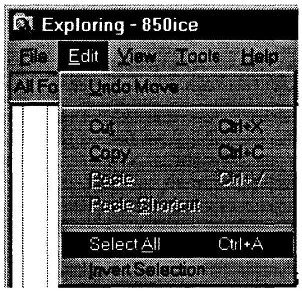
4.4.4 License Extension

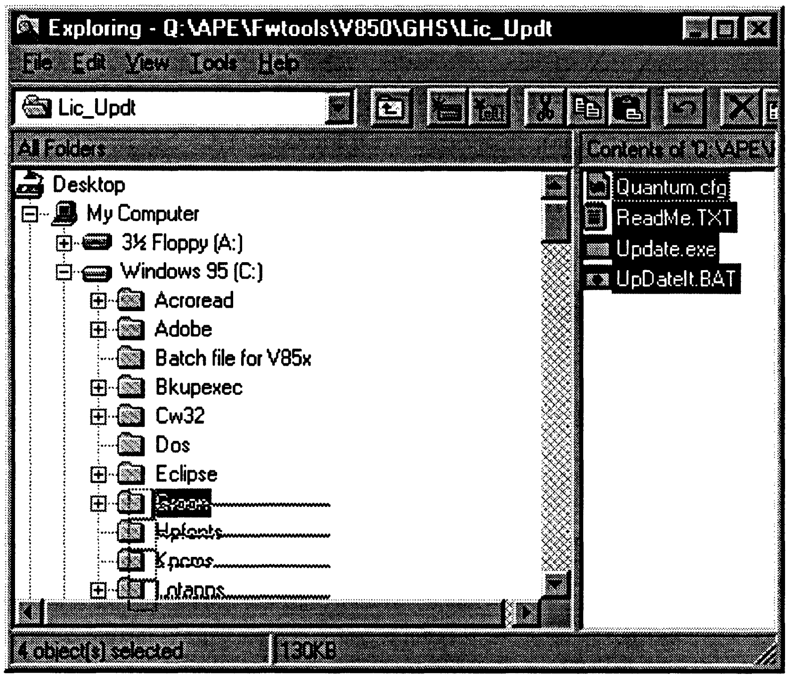
Using Win 95 Explorer, locate \\Ape\Vol1\Ape\Fwtools\V850\GHS\Lic_Updt



Double click on the **Lic_Updt** folder

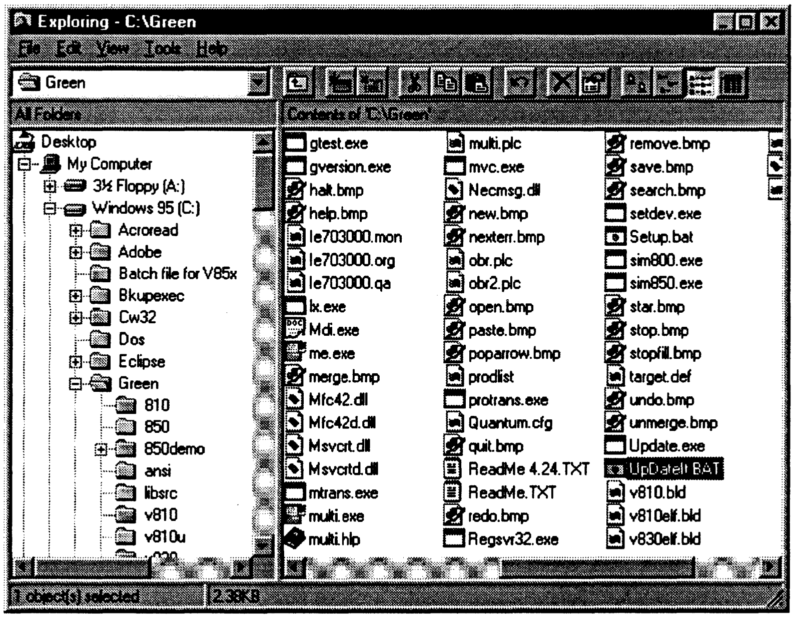
Select the **Edit\Select All** menu



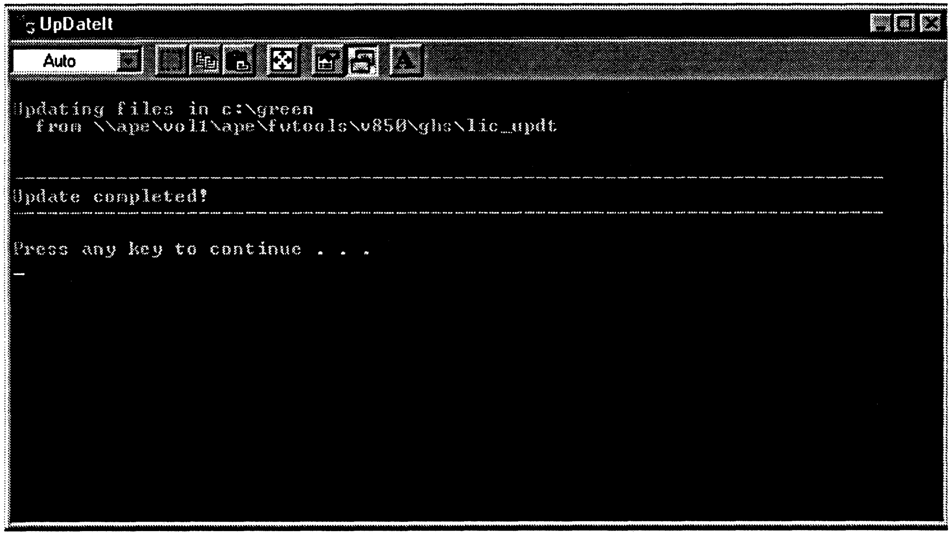


Click and drag the highlighted selection from the **Lic_Updt** folder to the **C:\Green** folder

Using Win 95, locate and open your local drive's **Green** folder



Double click on the **UpdateIt.bat** folder



```
Updating files in c:\green
  from \\ape\vol1\ape\fvtools\v850\ghs\lic_updt


--------------------------------------------------------------------------
Update completed!
--------------------------------------------------------------------------

Press any key to continue . . .
_
```

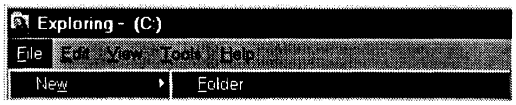
On the **Update** window,
- Press **any key**

then close the **Update** window

4.4.5 Shortcut to Multi

Using the Windows Explorer, create:

(1) an **Eclipse** folder in the root of the C drive,



(2) a **src** folder in the **Eclipse** folder,

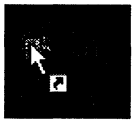
(3) an **at** folder in the src folder and/or

(4) a **scsi** folder in the **src** folder

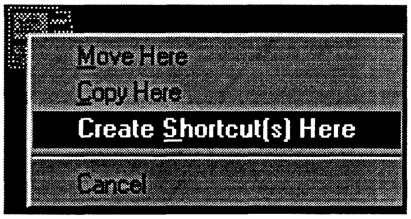
Note. The name you choose for you folders must be specific for your program. You may also want to create build/interface folders (i.e., E1AT, E2SC, ...)

Using the Windows Explorer, locate **C:\Green\Multi.exe**

Click and drag the Multi.exe file from the **Green** folder to the desktop using the right button of your mouse

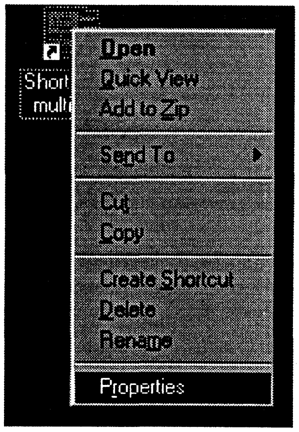


When a pop-up menu appears,
- Drag the mouse down to scroll to **Create Shortcut(s) Here** then press the left button of your mouse



Select the **Shortcut to multi.exe** icon on the desktop with the right button of your mouse.

When a pop-up menu appears,
- Drag the mouse down to scroll to **Properties** then press the left button of your mouse



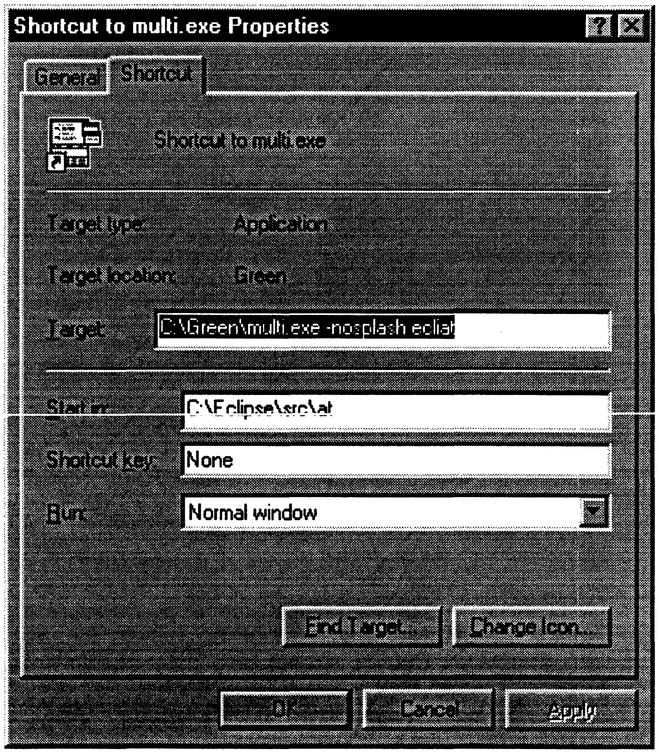
On the **shortcut to Multi Properties** window,
- Click the **Shortcut** tab

Within **Shortcut** section
On the Target Field
- Type **C:\Green\multi.exe -nosplash ecliat**

On the Start in Field
- Type either **C:\Eclipse\src\at** or **C:\Eclipse\src\scsi**

(**Note.** Choose the appropriate path for your program.)

Press the **Apply** button and then press **OK**



Shortcut to multi.exe Properties

Shortcut to multi.exe

Target type: Application

Target location: Green

Target: C:\Green\multi.exe -nosplash ecliat

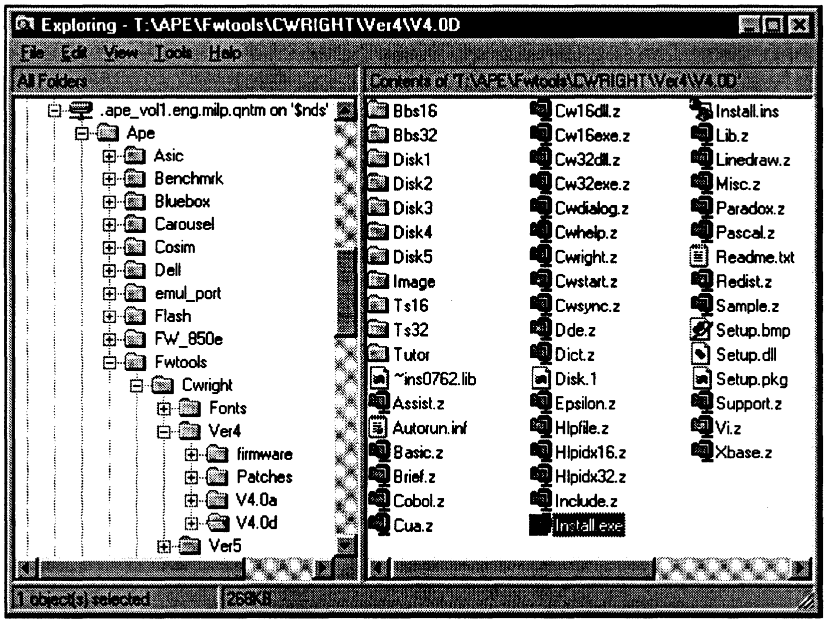
Start in: C:\Eclipse\src\at

Shortcut key: None

Run: Normal window

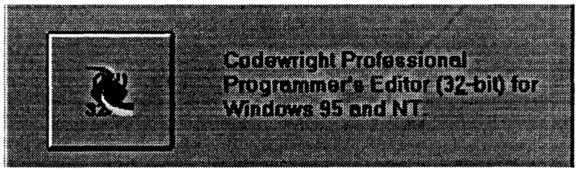
4B.5 Codewright Installation

Using the Windows Explorer, locate \\Ape\Vol1\Ape\Fwtools\Cwright\Ver4\V4.0d

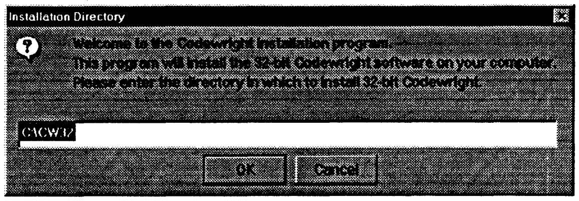


Double-click the **Install.exe** icon to start the Codewright Installation

When a **Codewright 4.0d Product Selection** window prompts you to select a product
- Press the **32-bit** button


Codewright Professional Programmer's Editor (32-bit) for Windows 95 and NT.

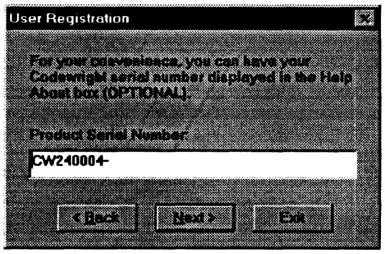
When an **Installation Directory** window prompts you
to enter the directory for the installation,
- Press the **OK** button (this accepts the default location)



Welcome to the Codewright installation program.
This program will install the 32-bit Codewright software on your computer.
Please enter the directory in which to install 32-bit Codewright.

C:\CW32

On the **User Registration** window,
- Press the **Next** button (this accepts the default selection)

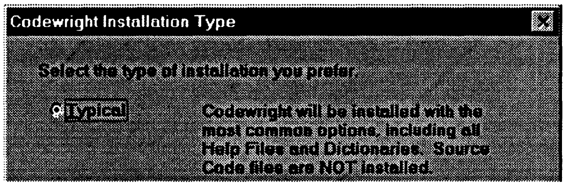


User Registration

For your convenience, you can have your Codewright serial number displayed in the Help About box (OPTIONAL).

Product Serial Number: CW240004-

When a **Codewright Installation Type** window prompts you
to select the type of installation you prefer.
- Select **Typical**
  then press the **Next** button

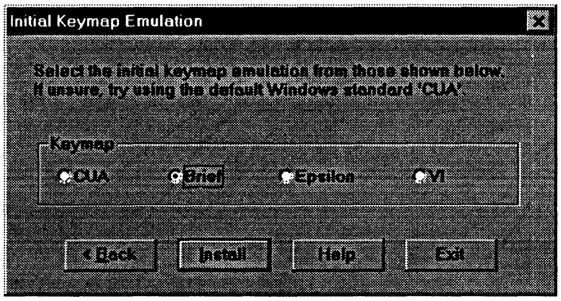


Codewright Installation Type

Select the type of installation you prefer.

● Typical — Codewright will be installed with the most common options, including all Help Files and Dictionaries. Source Code files are NOT installed.

When **Initial Keymap Emulation** window prompts you
to select a keymap
- Select **Brief** or **CUA** (your choice)
  then press the **Install** button



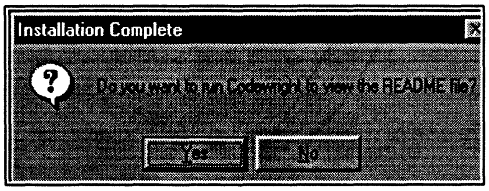
Initial Keymap Emulation

Select the initial keymap emulation from those shown below. If unsure, try using the default Windows standard 'CUA'.

Keymap: ○ CUA ● Brief ○ Epsilon ○ VI

On the **first Installation Complete** window,
- Press the **OK** button

On the **second Installation Complete** window, either
- Press the **Yes** button to view the readme file or
  press the **No** button to immediately view the installed items.



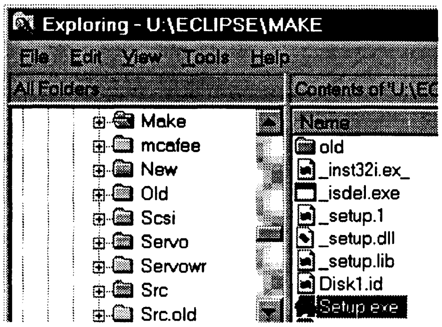
Installation Complete

Do you want to run Codewright to view the README file?

4B.6 Template Icons Installation

4.6.1 Installing Atlas Icons

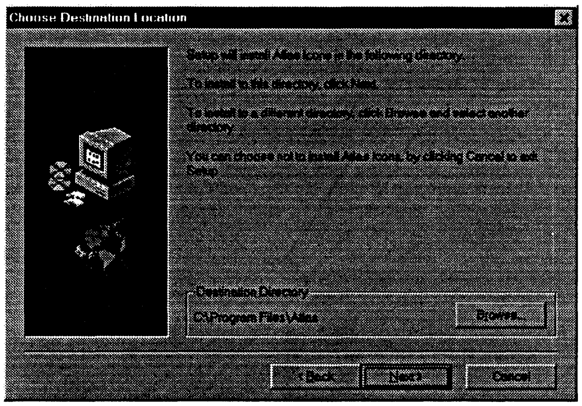
Using the Windows Explorer, locate \\Blue_light\Vol1\Eclipse\Make



Double-click the **Setup.exe** icon to start the Template Icons Installation

On the **Welcome** (to the Atlas Icon Setup) window,
- Press the **Next** button

When a **Choose Destination Location** window prompts you to Select the Products to Configure,
- Press the **Next** button (this accepts the default location)



Setup will install Atlas Icons in the following directory.

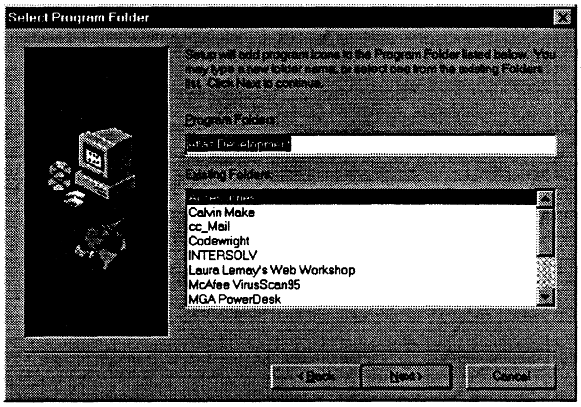
To install to this directory, click Next.

To install to a different directory, click Browse and select another directory.

You can choose not to install Atlas Icons, by clicking Cancel to exit Setup.

Destination Directory: C:\Program Files\Atlas

On the **Select Program Folder** window
- Press the **Next** button (this accepts the default folder)


Select Program Folder

Setup will add program icons to the Program Folder listed below. You may type a new folder name, or select one from the existing Folders list. Click Next to continue.

Program Folders:

[illegible] Development

Existing Folders:

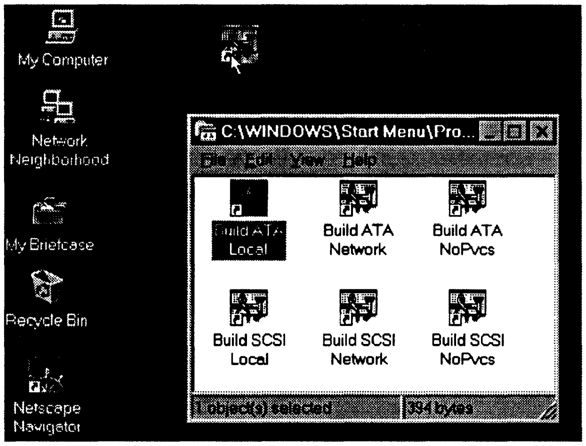
Accessories
Calvin Make
cc_Mail
Codewright
INTERSOLV
Laura Lemay's Web Workshop
McAfee VirusScan95
MGA PowerDesk

On the **Setup Complete** window,
- Press the **Finish** button to view the installed items

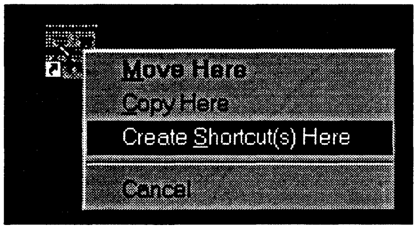
4.6.2 Shortcut to icon

To access the program development icons from your desktop,

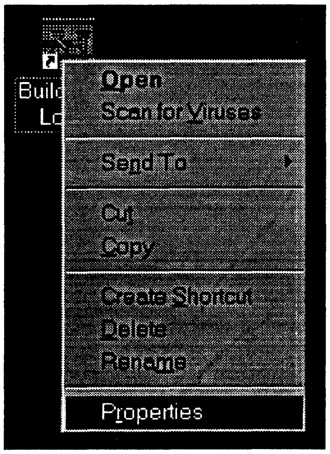
Select the **Build ATA Local** icon inside the Atlas Development window
Click and drag the icon out to the desktop by pressing the right button of your mouse



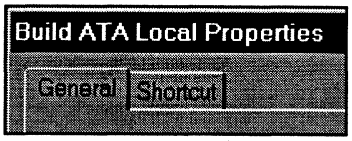
When a pop-up menu appears,
- Drag the mouse down to scroll to **Create Shortcut(s) Here** then press the left button of your mouse



Select the **Build ATA Local** icon on the desktop with the right button of your mouse
When a pop-up menu appears,
- Drag the mouse down to scroll to **Properties** then press the left button of your mouse



On the **Build ATA Local Properties** window,
- Click the **Shortcut** tab



Within **Shortcut** section:

(1) Change the information in the Target field

- The field information must have: **P:\CB\NT\BUILD.EXE E4_ATA=1 NOLINT=1 -script eclipse.mak makeall**

Note. **E4_ATA** is an example of a build stage and type.
Note. See Build Options on page 6-3 for definitions of field information.

(2) Change the **Start in** field information

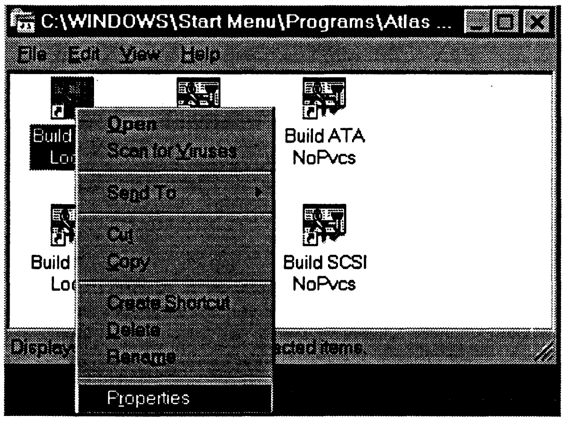
- The field information must have: **C:\Eclipse\src**

(3) Press the **OK** button

Note. To set the **program development shortcuts** within the Start/Programs menu

Select the **Build ATA Local** icon inside the Atlas Development window with the right button of your mouse

When a **pop-up menu** appears,
- Drag the mouse down to scroll to **Properties** then press the left button of your mouse



Follow the instructions in the beginning of this page

*** Apply the instructions in the **Shortcut to Icons** section to all the Atlas development icons.

4B.7 Path Check

Unlike Win 95 operating systems, which use an autoexec.bat file to set paths to the computer, **Windows NT 4.0** operating systems set paths through an environment.

- To access the environment:

  - Select the My Computer icon on the desktop with the right button of your mouse.

    When a pop-up menu appears,
    Drag the mouse down to scroll to **Properties** then press the left button of your mouse

    Select the Environment tab within the System Properties window then set the following in the **User Variables** field to support the Eclipse code builds:

```
set path= u:\eclipse\utils;c:\green;p:\vm\dos;p:\cb\dos

set user=TYPE YOUR USER NAME HERE

set vcsid=%user%

set temp=c:\temp

set tmp=c:\temp

set serversrc=u:\eclipse\src

set vcscfg=%serversrc%\vcs.cfg

set project=eclipse

set workdir=u:\mr\transfer\%user%

set iepath=c:\green

set islvini=c:\windows

set libpath=p:\vm\nt;p:\cb\nt

set server_path=u:\eclipse
```

*****Note.** Make sure that you TYPED YOUR USER NAME !!!!!

*****Note.** The workdir path, .../mr/..., used here is specific for the Tsunami program. Use the workdir path specific for your program.

4.8B Firmware Build

Before starting a build,

(1) Use the Windows explorer to:

(a) Create a Utils folder in the C drive's main folder (root directory)

(b) Drag the tm.exe. icon

from U:\\Blue_light\Vol1\Eclipse\Utils
to C:\\Utils

(2) Create a temp folder in the C drive's main folder (root directory)

**** Restart your computer**
**** Make sure your computer's network is mapped to:**

P:\\Lethal\Vol2\PVCS
Q:\\Ape\Vol1
U:\\Bluelight\Vol1

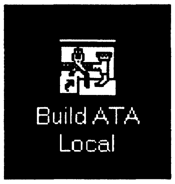
Note. An example of a disconnected computer path to **Bluelight\vol1**
To reconnect simply double click on the **Bluelight Server** icon in the Windows explorer.

(3) Using MSDOS

- Type **cd C:\eclipse\src**
  then press the **Enter** key

- Type **get eclipse.mak**
  then press the **Enter** key

Double-click one of the Build icons to start a build



4B.9 Project .rc file Edit

Using an editor, view the ".rc" file
(e.g., For the Eclipse AT project this file is c:\eclipse\src\at\ecliat.rc):

Make sure the file contains the following:

remote 850ICE**32**

If the file does not contain a "32" after the "remote 850ICE," use the editor to modify the file.

Note. The ".rc" file is a read-only file. Therefore, switch off the read-only setting of the file. To switch off the read-only setting:

Select the **ecliat.rc** icon with the right button of your mouse
When a pop-up menu appears,
- Drag the mouse down to scroll to **Properties** then press the left button of your mouse

On the **Ecliat.rc** **Properties** window,
Within **General** section

(1) Click the **Read-only** attribute to remove the check mark

(2) Press the **OK** button

Change a line to your project ".rc" file

from : remote 850ICE
to : remote 850ICE**32**

Note. After saving the file switch on the read-only setting of the file.

4B.10 Codewright Browser Installation (Optional)

To install a browser to Codewright , you must:

(1) Install Microsoft Visual C++ (version 2.0 or later)on you hard drive

(2) Delete the contents of your Src folder in C:\Eclipse

(3) Modify the Target field of the build icon that you will use

Make sure your ***Target field*** *contain the following path!*
If it does not, type in the missing path.

```
P:\CB\NT\BUILD.EXE E4_ATA=1 NOLINT=1 SBR=1
-script eclipse.mak makeall
```

(4) Modify the User Variables field in My Computer\Properties\Environment

Make sure your ***User Variables*** *field contain the following paths!*
If it does not, type in the missing paths.

set include=c:\msdev\include

set path=u:\eclipse\utils;c:\green;p:\vm\nt;p:\cb\nt;c:\msdev\bin

(5) Change the Browser Database Field Information in Codewright's Project\Properties\Directories

Make sure your ***Browser Database*** *field contain the following path!*
If it does not, type in the missing path.

C:\Eclipse\src\ecliat.bsc

(6) Reboot your computer, run Eclipse.bat, get Eclipse.mak, then double-click the build icon that you will use

Chapter 5 Codewright

5.1 Introduction

At the start of the Atlas firmware development effort, Codewright was choosen as the most advanced Windows based editor available at the time. Throughout this document there will be numerous references to particular dialog boxes and menu selections. These items will be uniquely specified using the following syntax: for each level of menu or dialog box, the entry to select will be displayed seperated by the | character. For example, to select the *fonts* dialog box under Windows 95 you would select the *start menu,* followed by *settings,* followed by *control panel,* followed by *fonts.* The short hand syntax for that in this document would be (Start Menu | Settings | Control Panel | Fonts).

5.1.1 Supported Versions

This document assumes that you are using Microsoft Windows 95, Codewright version 4.0E, GreenHills Multi version 1.8.7, and PVCS version 5.1.2.

5.1.2 File Location Assumptions

For the purposes of this section of this document we assume that the project is named Eclipse, your local project source files are in `c:\eclipse\src`. Codewright as been installed in `c:\cwright`, the GreenHills software is installed in `c:\green`. Windows 95 is installed in `c:\windows` and the network project files are in `u:\eclipse\src`.

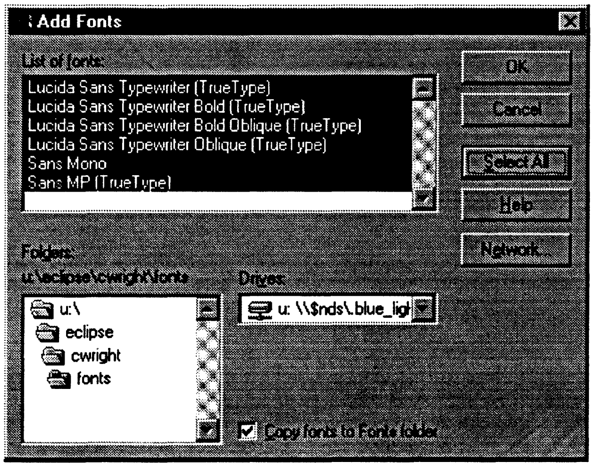
5.1.3 True Type Fonts

Codewright uses Windows true type fonts to display files, and you are free to pick any font you choose. Unfortunately, Windows has a very limited selection of fixed-point true type fonts from which to choose. There are two additional fonts that Premia supplied to address this issue, and you can load them as follows.

Run Win95 and open the fonts dialog box with (Start Menu | Settings | Control Panel | Fonts). Next select (File | Install New Font), and select the path:

`\\blue_light\vol1\eclipse\cwright\fonts`.

Finally, click on the (Select All) button and the (Copy fonts to Fonts folder) entry as indicated below:



List of fonts:

Lucida Sans Typewriter (TrueType)
Lucida Sans Typewriter Bold (TrueType)
Lucida Sans Typewriter Bold Oblique (TrueType)
Lucida Sans Typewriter Oblique (TrueType)
Sans Mono
Sans MP (TrueType)

Folders:
u:\eclipse\cwright\fonts

u:\
eclipse
cwright
fonts

Drives:
u: \\$nds\.blue_ligh

☑ Copy fonts to Fonts folder

5.1.4 Installing Codewright

Version 4.0a of Codewright can be installed from the network by running the following:

```
\\blue_light\vol1\eclipse\cwright\v4\cw4.0a\install.exe
```

The suggested directory for all versions is c:\cwright. Once you have completed this install there are patches that must be applied to bring Codewright up to the latest version. Apply the patches by running the following:

```
\\blue_light\vol1\eclipse\cwright\v4\patches\update.bat
```

To help you get started with Codewright (and the other tools), there is a set of configuration files modifed for eclipse firmware development on the network (this is in lieu of having you execute the "Billions and Billions" of steps required for manual configuration). To load these files execute the following:

```
\\blue_light\vol1\eclipse\cwright\v4\firmware\update.bat
```

This will rename existing files as follows:

c:\cwright\cfile.tpl	c:\cwright\cfile.sav
c:\cwright\hfile.tpl	c:\cwright\hfile.sav
c:\cwright\funct.tpl	c:\cwright\funct.sav
c:\eclipse\src\eclipse.pjt	c:\eclipse\src\pjt.sav

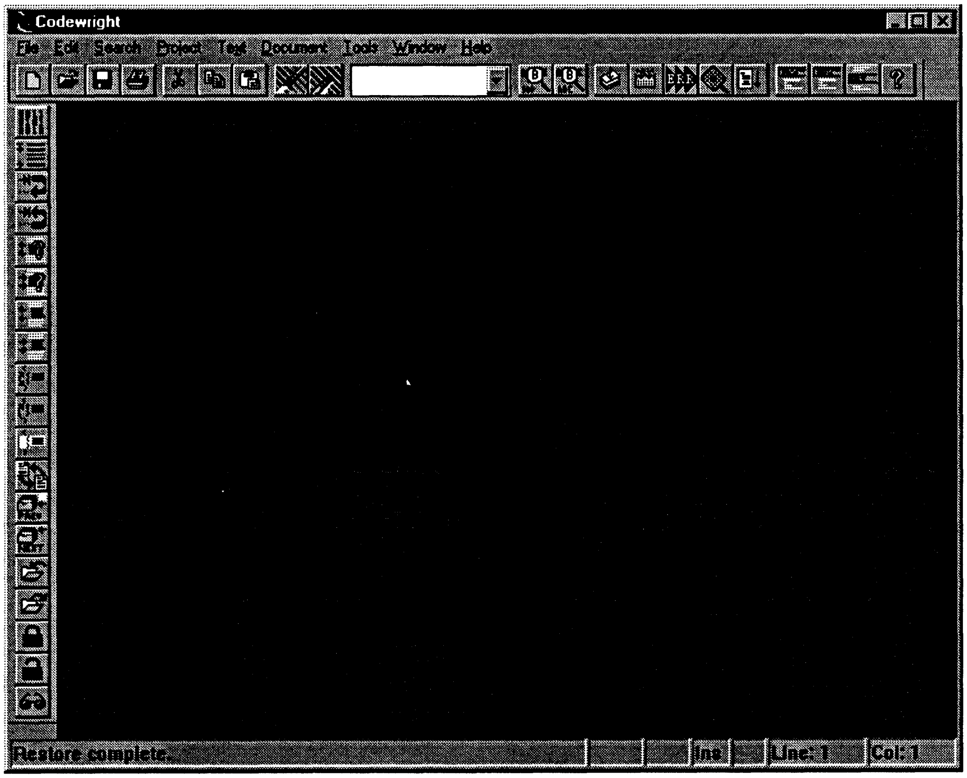
And will copy over the following new files:

```
c:\cwright\cfile.tpl
c:\cwright\hfile.tpl
c:\cwright\funct.tpl
c:\cwright\ini.new
c:\eclipse\src\eclipse.pjt
```

The next step is to merge in the entries in the file new.ini with your cwright.ini file. This is best done with an editor other than Codewright (as Codewrite likes to update the cwright.ini file when you exit). The new.ini file will have several sections that match those in cwright.ini, you will need to cut the information out of these sections and paste them in the corresponding sections in cwright.ini. Using this cut and paste approach allows for updating an existing cwright.ini file that you might have customized with your own preferences.

This merge of files will change the default appearance of Codewright by bringing up a tool bar along the left side containing a variety of usefull icons. There will also be three new icons added to the top tool bar just before the help (?) icon. The first of these is used in a blank file to create the default comment block for ".c" source files. The next new icon is also used in a blank file to create the default comment block for ".h" include files. The last new icon can be used anywhere in a ".c" source file to create a new function comment block.

When you first bring up Codewright after these changes it should look like the following:



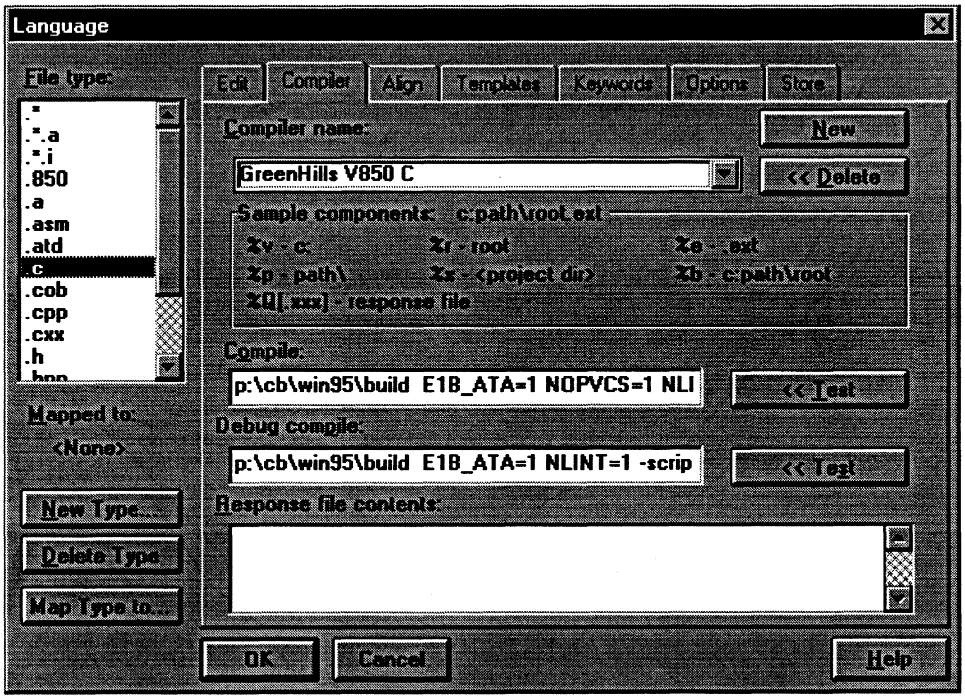
Restore complete.

5.2 Codewright Projects

Projects within Codewright allow you to have general options that apply to all files (with settings in cwright.ini) along with project specific options contained in your various project file. A sample project file was copied over in the previous steps and contains default settings for the Greenhills tools and eclipse source files. With this project file you can launch the compiler to compile the file you are working on, have the error parser examine the results of the compile (in the output window), and place the cursor at the first error in the file. This error parsing feature also works with the results of a full build of the code. To enable these feature you need to load the project file by selecting (Project | Open) and entering the path `c:\eclipse\src\eclipse.pjt`.

5.2.1 Changing Project Settings

The default project file is configured to build code for the E15B ATA platform using the Windows 95 version of the configuration builder. At some point you may which to change the default platform that Codewright uses when it runs the compiler. To make such changes, open a ".c" source file (Codewright makes certain dialog box decisions based on the type of the currently opened file). From the Codewright menu select (Project | Properties), from the Project dialog box select the (Tools) tab and then the (Compiler) button. If all went well you should see the following dialog box:



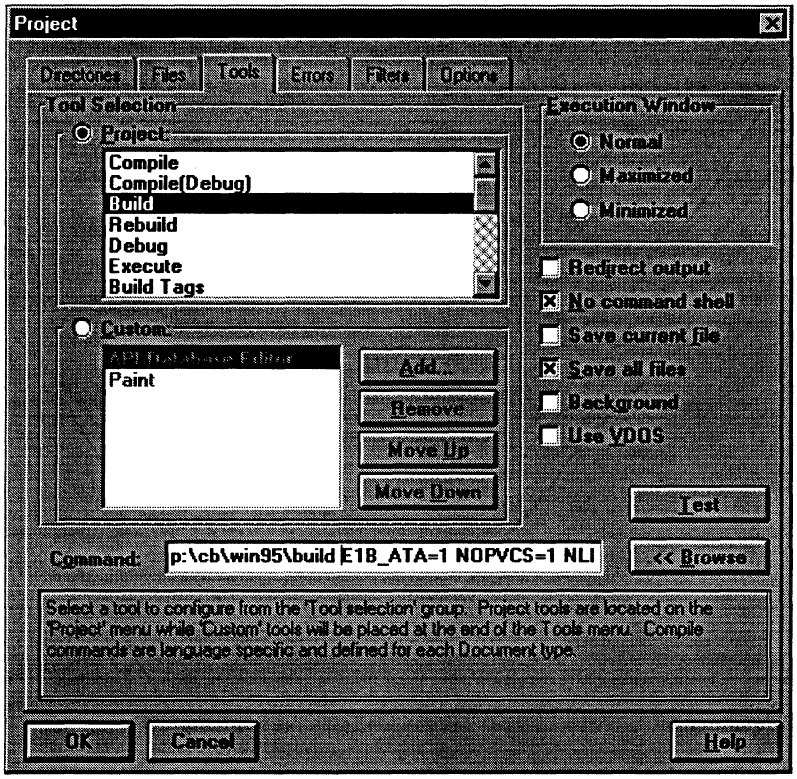
Compiler name: GreenHills V850 C

Compile: p:\cb\win95\build E1B_ATA=1 NOPVCS=1 NLI

Debug compile: p:\cb\win95\build E1B_ATA=1 NLINT=1 -scrip

If you need to switch from the Windows 95 to the Windows 3.1 builder, simply change the \win95\ above to \win\. To change platforms to the E1.5 SCSI (for example), change the E1B_ATA=1 above to E15_SCSI=1. In the above example the "debug compile" entry doesn't have the NOPVCS=1 option so it will fetch newer files from the network if necessary. The "compile" entry is a purely local compile, which is the more likely case for single file compiles from within Codewright.

If you also wish to change the platform for full builds spawned from within Codewrite, open a ".c" source file (Codewright makes certain dialog box decisions based on the type of the currently opened file). From the Codewright menu select (Project | Properties), from the Project dialog box select the (Tools) tab and then select the Build entry in the Tool Selection window. Hopefully you will see the following dialog box:



Tool Selection

● Project:

Compile
Compile(Debug)
Build
Rebuild
Debug
Execute
Build Tags

○ Custom:

API Database Editor
Paint

Execution Window

● Normal
○ Maximized
○ Minimized

□ Redirect output
☒ No command shell
□ Save current file
☒ Save all files
□ Background
□ Use VDOS

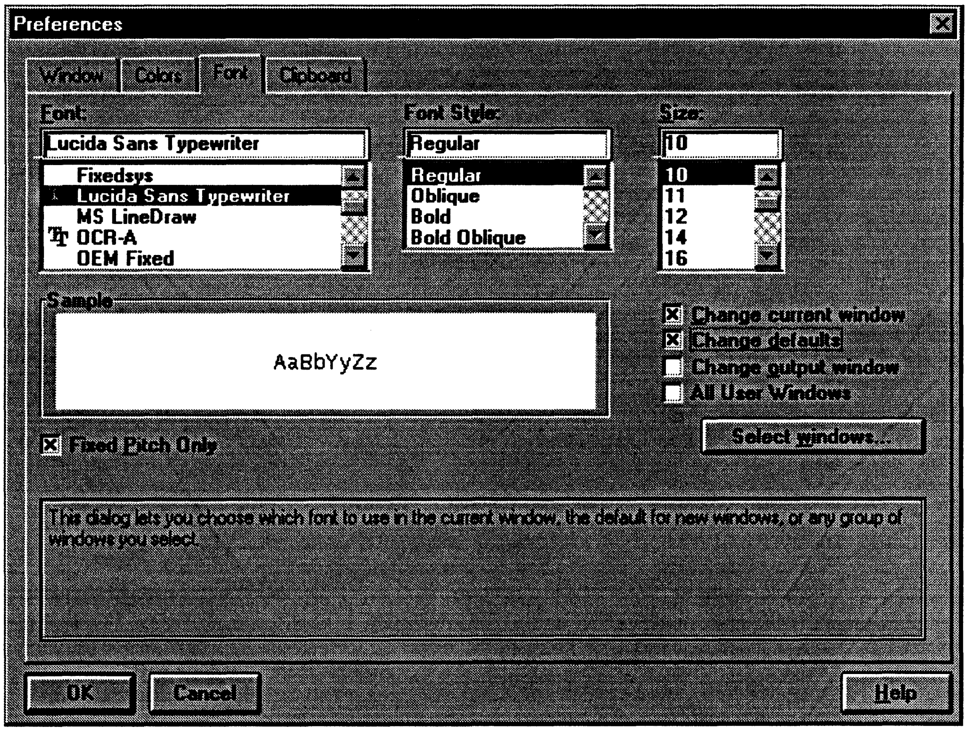
Command: p:\cb\win95\build E1B_ATA=1 NOPVCS=1 NLI

Select a tool to configure from the 'Tool selection' group. Project tools are located on the 'Project' menu while 'Custom' tools will be placed at the end of the Tools menu. Compile commands are language specific and defined for each Document type.

If you need to switch from the Window 95 to the Windows 3.1 builder, simply change the \win95\ above to \win\. To change platforms to the E3.5 SCSI (for example), change the E1B_ATA=1 above to E35_SCSI=1.

5.2.2 Changing Codewright Fonts

If you choose to modify the default Codewright font (a good idea) select (Document | Preferences) from the main Codewright menu. Next select the "Font" tab and once the dialog box is up, make sure that the "Change Defaults" box is selected. Select the font you wish from the ones presented as in the following example:



Font: Lucida Sans Typewriter
Fixedsys
Lucida Sans Typewriter
MS LineDraw
OCR-A
OEM Fixed

Font Style: Regular
Regular
Oblique
Bold
Bold Oblique

Size: 10
10
11
12
14
16

Sample
AaBbYyZz

☒ Change current window
☒ Change defaults
☐ Change output window
☐ All User Windows

☒ Fixed Pitch Only

This dialog lets you choose which font to use in the current window, the default for new windows, or any group of windows you select.

You may limit the list of fonts to only fixed point ones by checking the (Fixed Pitch Only) box. If left unchecked you will see both fixed and proportionally spaced fonts.

Chapter 6 Make

6.1 Introduction

The Eclipse make file was designed to be run by a Windows (95 or NT) based make processor (generally the Intersolv Configuration Builder version 5.1 or greater) and can not be run in a Dos environment. The primary tools (compiler, assembler, linker, and some post processors) invoked by this file are likewise Windows applications from GreenHills Software. There are, however, a few Dos based utilities that are invoked to perform various simple tasks (like deleting files, or checking for differences between files).

Each supported platform that the make file can build is uniquely identified by a single define string. This define string is passed into the make on the command line and is then passed to the compiler for each file compiled. This define string must match an entry in the r_coddef.h file which is included by each module. This necessitates that changes in the make file definition section must be matched by changes in r_coddef.h.

In order to encourage as much commonality as possible, a single source directory can be used to build code for multiple different platforms without conflict. By default, the builder will create an AT or SCSI subdirectory below the source directory depending on the interface specified by the user. These subdirectories will contain the following programmatically generated files: object, list, debug, coff, upd, asmsym, config page binaries, selfscan script binaries, and cpg files. The user can override the default sub directories with command line options allowing for an infinite (ok, a bunch) of different builds all created from a single source directory.

6.2 Invocation

The builder can be launched via the command line, through an icon within Windows, or via a project specific menu entry in Codewright. The Codewright method is the preferred as Codewright can parse the output of this make run and look for errors. The textual output of the tools run by this make file are collected into a file named "proj.err" created in the working directory where the configuration builder was run. The information in this file is parsed by a utility program (projerr.exe) and extraneous output is filtered out, leaving only error information which is displayed at the end of the make run.

As described above the invocation must contain the unique define required to specify the platform and interface, along with any builder options. For example, to build code for an ATA interface at build level E1 the following icon command line would be used:

```
P:\CB\WIN95\BUILD E1_ATA=1 OBJ=E1AT -script eclipse.mak makeall
```

The E1_ATA is the top level define that matches an entry in r_coddef.h, while the "-script eclipse.mak" specifies the make file to use. The "makeall" is the "first target" for the make and specifies which target within the make file to build. This example is for the Windows 95 version of the Intersolv configuration builder as indicated by the path specified. The resulting files would be created in a subdirectory named E1AT under the source directory.

6.3 Dependencies

In the past, it was up to the individual firmware engineers to keep the dependency section of the make file up to date (this section indicates which include files a source file is dependent on). This make file has a separate DEPEND.MAK that is programmatically generated by an Intersolv utility program call Scandeps. This utility is passed a wildcard pattern for the source files in the users directory (generally *.c and *.850 files). From these inputs, Scandeps builds dependency lists of all the .H and .I files used by the sources. These dependency lists are used to determine when specific modules must be rebuilt because of include file changes (nested includes are supported automatically). It is the responsibility of the individual firmware engineers to run Scandeps against the make file whenever they add new source or include files.

6.4 Targets

The make file contains a number of "top level" targets that are passed in via the command line and indicates what part of the code the user wants built.

6.4.1 Makeall

This is the default target used if the user fails to specify one on the command line. This target will build the ROM and the RAM (diskware) code as two separate units and link the results together. This target will detect when a ROM file is being rebuilt during a diskware phase and will generate an error.

6.4.2 Getall

This target will retrieve all the necessary files to build a particular platform. The make file is capable of fetching the files on demand, but this method is much faster.

6.4.3 Rom

This target will compile/assemble only those files that are required for the ROM. The result of this build will be a "library module" that represents the ROM code. This library module can be linked with the objects generated in a RAM build, to create a complete code set.

6.4.4 Ram

This target relies on a previous ROM build to create a complete set code set. Only RAM modules will be recompiled prior to the link phase. This is of course the normal build used after a ROM freeze.

6.4.5 Filename.o

This target is used primarily for single file compiles from within Codewright. In order to insure that the correct build parameters are used, we invoke the build and specify only a single object file be built.

6.4.6 Filename.ler

This target request that the builder run Lint against the specified file name and create the indicated Lint output file. Once again this target is primarily intended for invocation from within Codewright.

6.4.7 Newrel

This target will stamp a version label on all files in the archive and is generally done when a new release of the code is made.

6.5 Build Options

There are a number of options that effect the build behavior and can be applied to any top-level target. These options are passed to the build on the command line.

6.5.1 LOCAL=1

This option affects the behavior of the Intersolv "get " utility in that it prevents the retrieval of any new files from the network. If the file does not exist in the user's directory, then the file will be retrieved, otherwise the user will be notified that a newer file exists on the network.

6.5.2 NOPVCS=1

This option prevents the builder from even looking at the network. This is therefore the fastest possible build, but the user will never know that newer files exist on the network. Since this option prevents the builder from retrieving any files, it is up to the user to insure that all the required files are already on his/her machine.

6.5.3 SBR=1

This option will cause the builder to invoke the Microsoft C compiler in order to generate "Source Browser " files. These files are combined to create a BSC database that is used by Codewright to browse the source files.

6.6.4 NOLINT=1

This option prevents the builder from invoking Gimpel's Lint utility on each "C" source file. The Lint option is left as a "default on" flag in order to encourage the user to cleanup any errors/warnings found by Lint.

6.5.5 OBJ=AE1B

This option overrides the default object directory that will contain the programmatically generated files. Using this option allows the user complete (within the constraints of the Dos file system) freedom to specify subdirectories to contain their particular build files. In the above example, if the users source file was in the directory c:\eclipse\src there would be an object directory created with the path c:\eclipse\src\ae1b that would contain the files necessary for either the debugger or a prom burner.

6.5.6 VERSION = "A01.0000"

The version string determines the file revision retrieved from the archives during a build or "getall" operation, and sets the version label to be associated with the tip during target "newrel". VERSION assignment is mandatory for "newrel", optional for other targets. When undefined, the makefile will retrieve from the tip, depending upon the setting of LOCAL and the source directory's contents. The version string also determines the ROM and RAM version strings that will be built into the resulting code, except that when LOCAL is defined the RAM value will be overwritten with "@@@@". When the user intends to make modification from the current tip, they should use the 'getall' target without a VERSION argument, then build (currently, with "makeall") with an appropriate version string.

6.6 Utilities

The make files requires a number of utility programs to do its job. Some of these utilities are supplied by Greenhills, while the rest were created internally.

6.6.1 Asmsym (assembly symbol generator)

This Quantum-generated utility maintains synchronization between "C" and assembly language files. Normally, when both an assembly module and a C routine need to access a C based data structure, the assembly routine would need manually generated offsets to the individual elements of the data structure. Asmsym parses the compiler-generated list files and builds two include files. The asmsym.i file is used by assembly language routines while the asmsym.h is included by C routines. The asmsym.h file provides symbolic identifiers for individual bits within C defined bit field structures.

6.6.2 Gover

This is a Greenhills-provided utility that extracts user specified sections out of the linker-generated COFF file and creates individual files for the grouped sections. This provides a method of extracting code and/or data from the linker output file for use by post-processors. This approach allows extraction of initialized data from the firmware along with downloadable code fragments. These extracted files are used as input to Makeprom as part of the upd/cpg file generation.

6.6.3 CoffROM

This Greenhills-supplied utility gives us the ability to move sections of memory. This is used to move initialized data from its required position in memory to the end of the ROM. Since the majority of our data is in the ZDA section we have to be able to bootstrap initialized data out of ROM and into its desired position. This bootstrap operation is provided by the _Start function in R_CRT0.C which expects a sequence of initialized data to exist at the end of the ROM after the special section ". shadow".

6.6.4 Blderr

This Quantum-generated utility creates all the internal and external error files using a single data file (errort.dat) as input. This guarantees that the following built files will always be in sync: errort.c, errort.h, r_errort.h, errort.i, and eclipse.err. The first 4 files are compiled as part of the code and the last is the Diag error file.

6.6.5 Bldvec

This Quantum generated utility reads in a data file (vectbl.dat) and builds both the ROM and RAM vector tables. It generates the following files: r_romvec.850, dwvec.850, and vectort.h. The first two files are the actual vector tables while the last is the include file used by any module referencing a vector table entry.

6.6.6 Projerr

This Quantum generated utility reads in the file created during a build run that may contain compiler error information. This file (proj.err) will generally contain the compiler banner line along with information about files retrieved from the network. This utility filters out these extraneous lines and if there is any compiler error information left, it will be displayed. This program is run at the end of the build session, and if it doesn't find any errors it will display nothing. The input file (proj.err) is still available to the user for perusal and is parsed by the Codewright error parser.

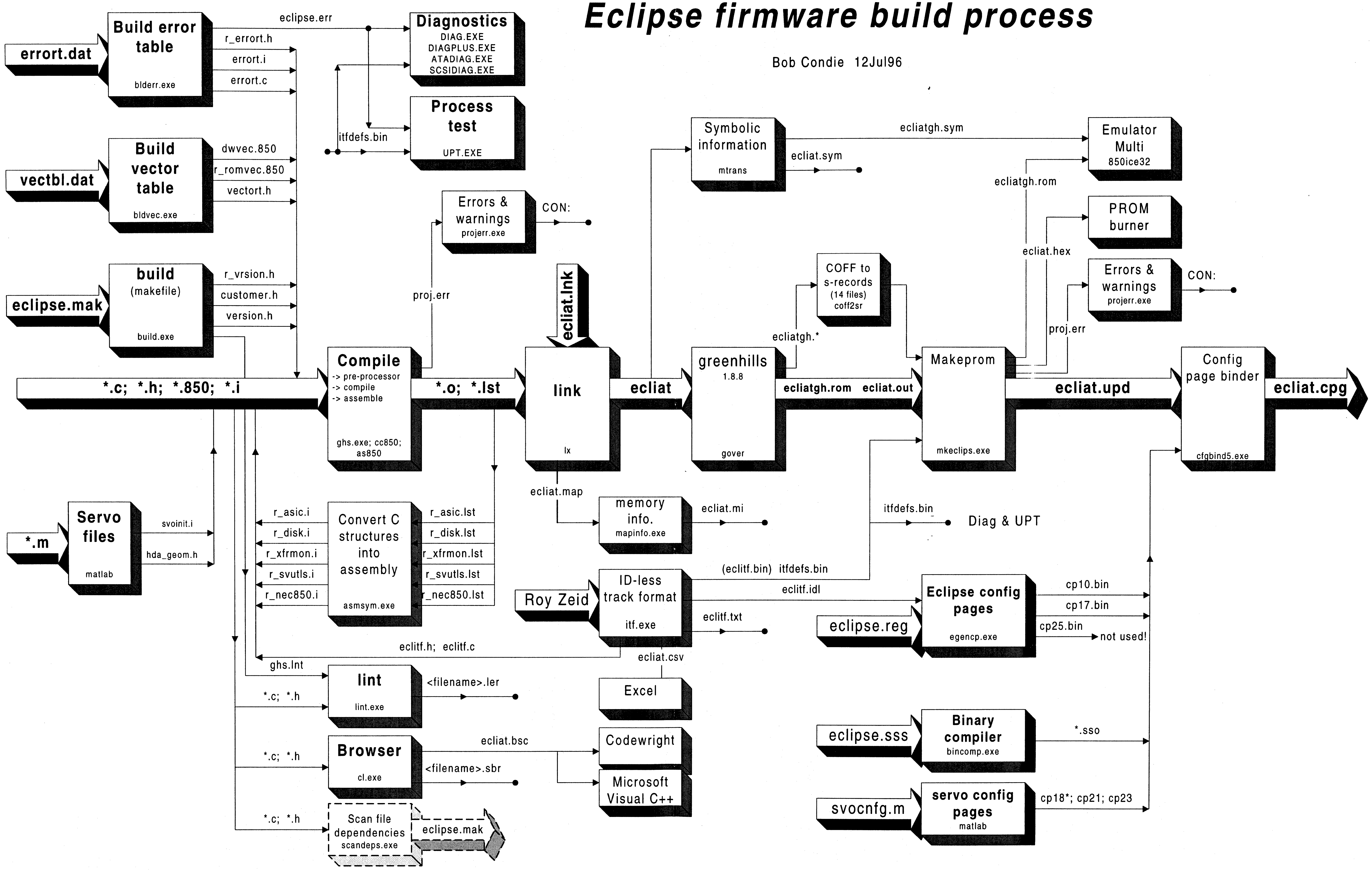
6.6.7 Makeprom

This Quantum generated utility takes files from the Gover utility to create output files that are used on the drive. Makeprom loads the files that Gover outputs, checks the files for validity, updates the headers in each file and checksums each file. Using the validated files, Makeprom builds a .hex file (ROM image file, used to generate the ROM), a .upd file (diskware image file, used to Ramware or Diskware the drive) and several Self Scan files (Self Scan files are used to self test the drive). Makeprom also compresses the ROM vector table and updates the emulator download file with the compressed ROM vector table.

(This page was intentionally left blank.)

Eclipse firmware build process

Bob Condie 12Jul96



Chapter 7 Using the V851 In-Circuit Emulator

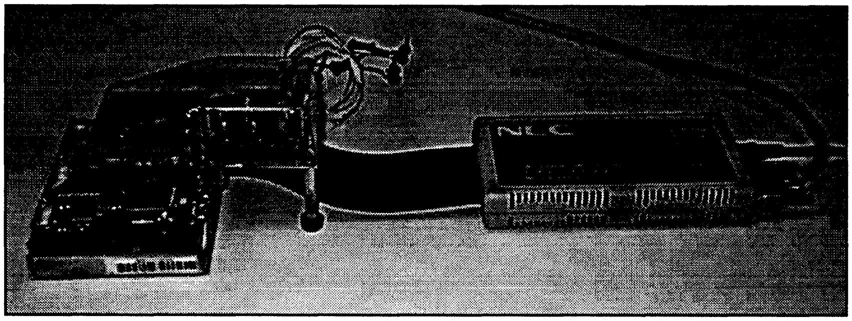


Figure 7-1. Emulator, Pod, and Target Disk Drive

7.1 Introduction

The V851 emulator is a dramatic step forward in firmware debugging. It provides some very powerful features that Quantum has never had before in an emulator:

- Source code level debugging in 'C'.
- Windows GUI Interface.
- Symbolic viewing and editing of all data structures down to the bit level.
- The ability to stop the firmware at a breakpoint while servo interrupts and motor continues to execute.
- Point-and-click setting of breakpoints while the firmware runs (a momentary halt which does not affect servo occurs).
- Advanced trace filtering, dump, section, and search features.
- More hardware bus events (BRAs -- 1-8) and instruction execution events (BRSs -- 1-16).
  Note: The NEC manual tells you that you have 16 BRSs. However, 4 are reserved for various debugger functions.
- Overlay debugging support.
- Very small packaging.

In this section we will explore more than just the emulator itself, although the focus will be on the emulator. We will also briefly touch on some of the other requirements for setting up a debugging environment, such as the computer for running Diag software and building the required object and binary files. However, these subjects will not be covered in full detail. This will leave you with some questions when you get to details such as understanding what files are used by the emulator and where they are retrieved from. Some of this information can be found in the 'build' documentation. In other cases you'll have to dig just a little deeper into the tools themselves.

7.2 Emulator Resources

- 33 MHz Processor
- 512K ROM
- 28K Internal RAM
- 2 MB Emulation Memory

7.3 Hardware Requirements

7.3.1 Emulator Hardware

- NEC V851 Version 3.0 Emulator
- NEC V851 Version 3.0 Emulator 'pod'
- Emulator interface Card for your PC
  (the same card used for the K7)
- NEC V851 Emulator Interface cable
- NEC V851 Emulator Power Supply
- Emulator 'probe' clips (optional).
- Overlay Manager Board and external trigger pin connecting wires (optional).

7.3.2 Target Hardware

- An appropriately socketed target drive.
  Note: Guideposts can be used in the socket to ease the attachment of the pod.
- A computer for running Diag software, connected to the target via its host (e.g., AT or SCSI) interface.

7.3.3 Computer

- Pentium 90 or better (the more power the better)
- 32 Meg RAM or better (the more the better)

7.4 Software Requirements

- Windows 95 OS
- NEC 850 Ice Emulator Software (see "Software Tools Installation - Setup & Configuration")
- Emulator Icon on your desktop
- Diag program
- ITF Program for Your Project
- Resource ("rc") files:
  These files are controlled under PVCS for your project. They are retrieved into your object file directory. They provide various emulator functions.

1. *project*.rc (i.e., ecliat.rc or ecliscsi.rc) -- Green Hills project specific startup script responsible for emulator initialization, download of code and definition of custom buttons for the debugger.
   Note: *project.rc* is executed by 'Multi' on startup based on the command line environment variable that you supply to the debugger icon's command line. For instance, the Eclipse AT project supplies the environment variable ECLIAT, so ecliat.rc is looked up in the 'start in' directory and executed.

2. multi.rc -- Greenhills debugger global startup script file.
3. user.rc -- A personalized configuration file. It is invoked using a 'configure' command in project.rc. It currently specifies the font used in the debugger.
   Note: Configure files are different from other 'rc' files in that you can't execute the same kinds of commands in them, although there is some overlap. They have a different purpose; defining how your emulator windows look.

7.5 Object and Binary File Components

When you do a 'build', the appropriate files are generated for emulation. In addition, other files required to make the drive read and write will be retrieved from PVCS by the build process into your object file directory. The files are as follows:

7.5.1 ROM Object File for Emulator Download

A successful build produces a file named after your project: for Eclipse AT the file is called ECLIATGH.ROM. This file will be downloaded into the emulator's ROM address space when you start the emulator.

7.5.2 Ramware 'UPD' File

A successful build also produces a file named after your project: for Eclipse AT the file is called ECLIAT.UPD. This, of course, is if you have reached the point in your development cycle in which you can download ramware or diskware. It is your responsibility to transfer this file to your Diag station for download.

7.5.3 Channel Parameters File

The channel parameters are in configuration page 17. This is binary data used to program the channel registers for all the zones on the disk. Further refining is done by 'channel training'. The file 'CP17.BIN' contains the channel parameters. It is your responsibility to transfer this file to your Diag station for download.

7.5.4 ITF Sector Descriptor Binary File

At least one person in your project is responsible for the Idless Track Format. If you are working with the 'Dilbert' or earlier ASIC, then 'sector descriptors' are required for read/write to work in your drive. If you are working with the 'Rebel' or later ASIC then there is a 'calculator' which replaces the sector descriptors. However, you still need an ITF binary file for parameters within the Idless formatter. For Eclipse, this file is called 'ITFDEFS.BIN'.

A note on ITF: Using the ITF program and its binary file output allows firmware engineers to tune the read/write parameters of the drive without having to spin-down the drive, re-build the firmware, and restart the emulator with each experiment. Your firmware should have no hard-coded values for programming into the formatter that you code yourself. Your team should work with the tools group (which maintains ITF for your project) to get the required parameters for your drive.

It is your responsibility to transfer this file to your Diag station for download.

7.5.5 Configuration Pages Binary Files

The files 'CP*.BIN', (where '*' is a wild card) contain configuration page data to download into your drive. These will drive many of the algorithms in the firmware and will be needed for initialization of your target. It is your responsibility to transfer these files to your Diag station for download.

7.5.6 Diag Macro File

A file for Diag is usually provided by someone in your team which contains some handy macros for starting up your target. Here are some typical macros:

- **rw** - Loads Ramware (diskware that's not yet written to system sectors).
- **dw** - Loads Diskware (same as ramware except it is written to system sectors, which can be loaded next time the drive spins up).
- **loadcp10** - Loads configuration page 10.
- **loadcp17** - Loads configuration page 17, which are the channel parameters.
- **loadcp23** - Loads configuration page 23.
- **wrddsk** - Writes ITF sector descriptors to system sectors.
- **init** - Writes configuration pages and sector descriptors to the target. This is your full-function initialization.
- **getchb** - Reads all channel registers.
- **putchb** - Writes all channel registers.
- **getch** - Reads one channel register with prompt.
- **putch** - Writes one channel register with prompt.
- **w2w** - writes wedge-to-wedge.
- **rw2w** - Reads Wedge-to-wedge.
- **And more...**

For eclipse AT, this file is called ECLIPSE.ATD.

7.6 Setting Everything Up

You've gone on your scavenger hunt and collected all the required hardware and software to get started. Here is a checklist to help you make sure you've got everything set up.

7.6.1 Emulator Connections

- First connect the emulator 'pod' to the target drive. The drive's processor has presumably been socketed with guide posts to help you attach the pod. Align the notches on the pod and drive processor sockets!
  Note: If you intend to debug in overlays then you need an 'overlay manager' board. Connect this board in the CPU socket between the pod and the target drive. Also, connect the four wires between the overlay manager board and the emulator's external trigger pin inputs.
- Connect the emulator interface cable between the emulator and the interface card in your computer.
- Make sure the emulator's power switch is off. Then connect the emulator power supply.
- Make sure the target drive's power supply switch is off. Then connect the target drive's power supply.
- Connect the interface cable between the Diag station and the target drive.

7.6.2 The Emulator Shortcut (Desktop Icon)

This Icon is also documented in your software tools installation. Here are the settings required to make it work:

- Target: C:\green\multi.exe -nosplash ECLIAT
- C:\green\multi.exe executes the 'Multi' program. 'Multi' starts the 'QBox' server software.
- -nosplash is an option that prevents a silly Green Hills logo from appearing, wasting startup time.
- ECLIAT is an environment variable for the Eclipse AT project. Everywhere you see *'project'* in this document, 'ecliat' is substituted.
- Start in: c:\eclipse\src\at

7.6.3 Safely Applying Power to the Emulator and Target

1. Apply power to the emulator.
2. Apply power to the target drive.

7.6.4 Safely Turning Off Power From the Emulator and Target

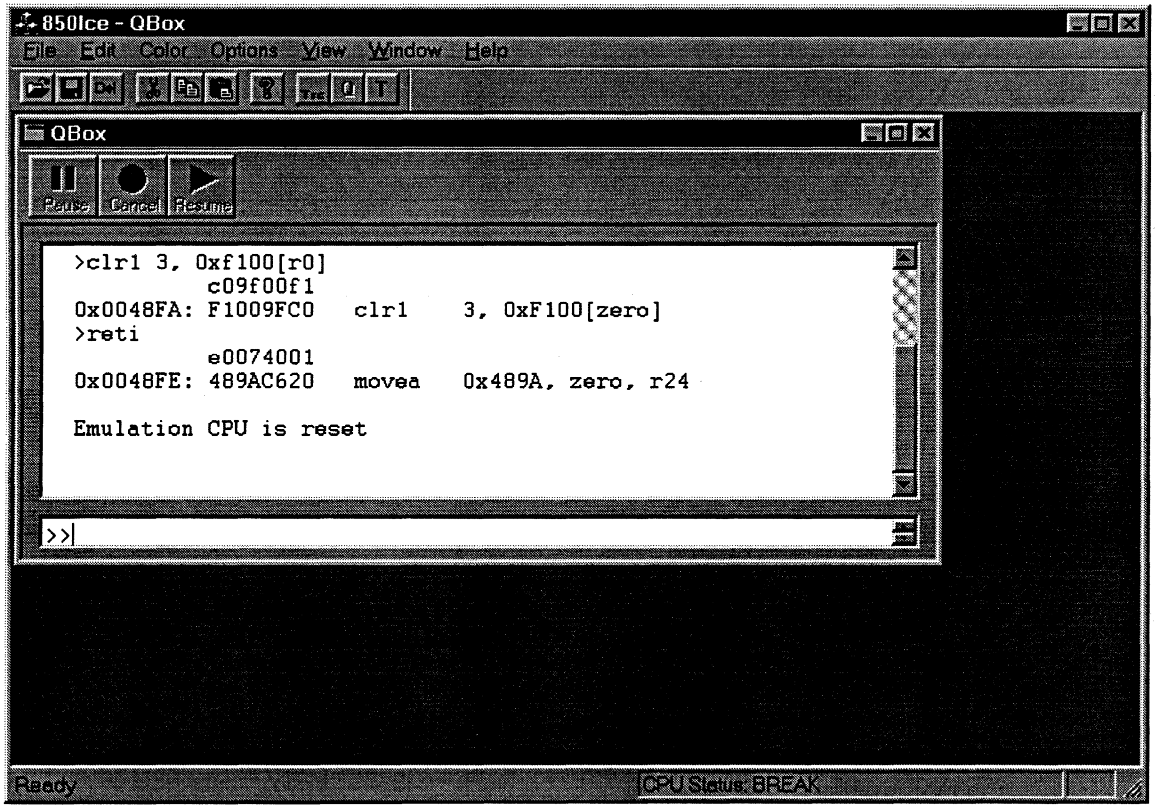
1. Power off the target drive.
2. Power off the emulator.

7.7 Starting the Emulator

Double click on the emulator shortcut to multi.exe (icon) on your desktop. Two emulator software windows will appear on your screen. There are two parts to the emulation package; a 'server' produced by NEC called the 'QBox', and a 'client', which is part of the 'Multi' environment produced by Green Hills.

Note: The user interface to these windows is not 'windows standard'. Many of the normal mouse and keyboard activities you expect will work differently or not at all.

7.7.1 '850ice' - QBox window:



```
>clrl 3, 0xf100[r0]
          c09f00f1
0x0048FA: F1009FC0   clrl    3, 0xF100[zero]
>reti
          e0074001
0x0048FE: 489AC620   movea   0x489A, zero, r24

Emulation CPU is reset
```

```
>>|
```

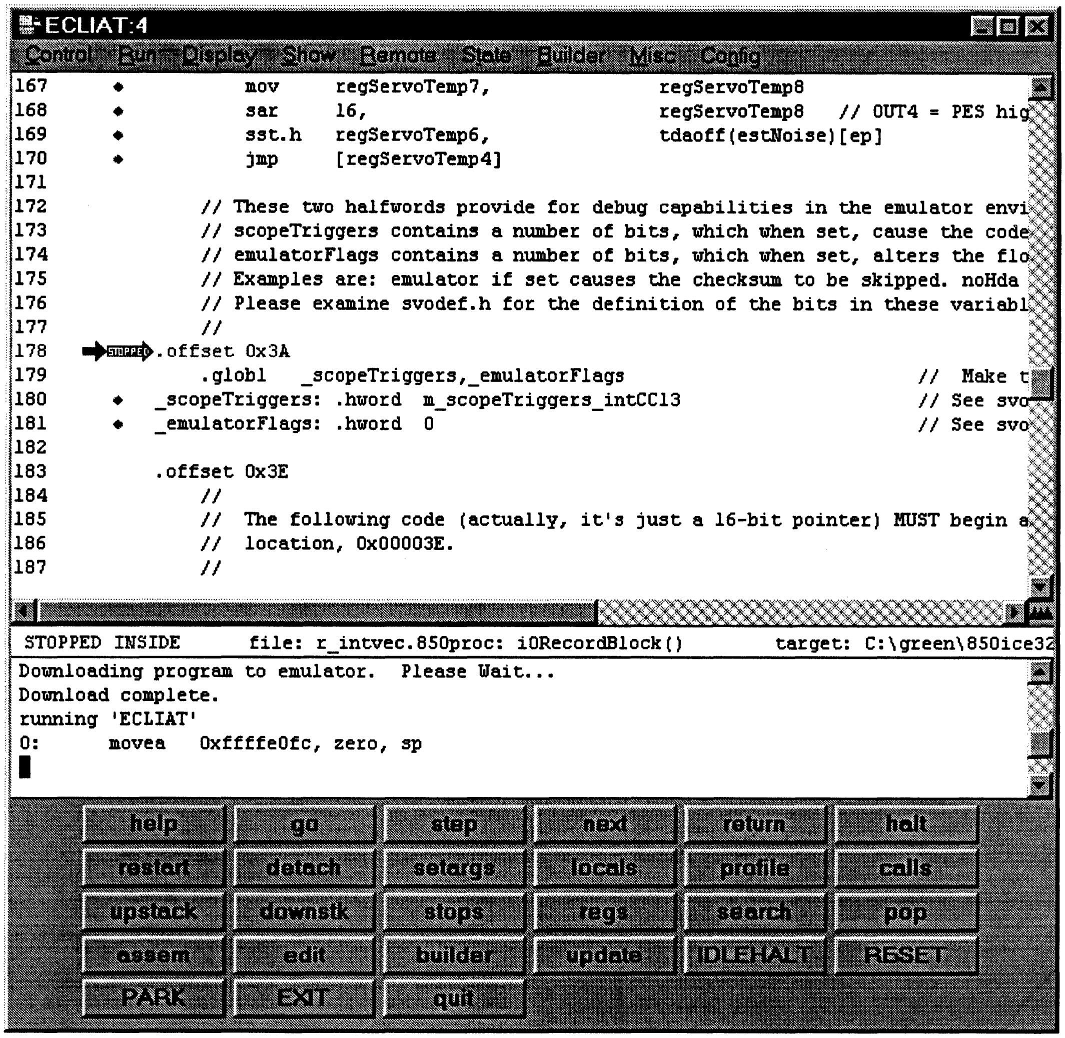
Figure 7-2. 'QBox' Server.

This is the software that communicates directly with the emulator hardware. This is where you will typically do trace captures and dumps, memory dumps and changes, assembly and disassembly, and setting hardware breakpoints. NEC provides a manual on this software called the '850ice Specification'.

The user interface in this window is very similar to the K7 emulator. However, there are some syntax differences for you to get used to. You enter a command line in the command line window at the bottom and see your results in the display window above.
Note: You can press the up and down arrow keys to re-use your command history.

This window displays code and variables in assembly language and memory dump formats. This is different from the 'Multi' window which displays in the source code format.

7.7.2 'Multi' Debugger Window



```
    ◆           mov     regServoTemp7,                  regServoTemp8
    ◆           sar     16,                             regServoTemp8   // OUT4 = PES hig
    ◆           sst.h   regServoTemp6,                  tdaoff(estNoise)[ep]
    ◆           jmp     [regServoTemp4]

            // These two halfwords provide for debug capabilities in the emulator envi
            // scopeTriggers contains a number of bits, which when set, cause the code
            // emulatorFlags contains a number of bits, which when set, alters the flo
            // Examples are: emulator if set causes the checksum to be skipped. noHda
            // Please examine svodef.h for the definition of the bits in these variabl
            //
   STOPPED .offset 0x3A
            .globl   _scopeTriggers,_emulatorFlags                         //  Make t
    ◆   _scopeTriggers: .hword  m_scopeTriggers_intCC13                    // See svo
    ◆   _emulatorFlags: .hword  0                                          // See svo

        .offset 0x3E
            //
            //  The following code (actually, it's just a 16-bit pointer) MUST begin a
            //  location, 0x00003E.
            //
```

```
 STOPPED INSIDE      file: r_intvec.850proc: i0RecordBlock()        target: C:\green\850ice32
```

```
Downloading program to emulator.  Please Wait...
Download complete.
running 'ECLIAT'
0:      movea   0xffffe0fc, zero, sp
```

Figure 7-3. 'Multi' Debugger Window.

The 'Multi' debugger window shown in figure 7-3 allows you point-and-click debugging at the source code level. This tool does its work by communicating with the 'server'. Notice that it is divided into two panels. The upper panel shows the source code at location 0x3A, where the PC register currently sits (the current execution address). We will call this the source code panel. Green Hills provides a manual called the 'MULTI Software Development Environment User's Guide' for this software. This manual also covers their software development tools, such as their 'builder', which we don't use.

The lower panel is the 'input panel. It shows the messages which appeared when the code was downloaded and allows you to enter commands later. The initialization and download occurred when the 'multi' software executed the file 'ecliat.rc' (*project*.rc). This window gives you a command line interface that lets you request the display of source code files, variables, and structures.

The buttons at the bottom of the window give you point-and-click control over execution and breakpoint setting. There are default buttons and customized buttons. The buttons that are customized have all uppercase characters. The default buttons are all lowercase and are documented in the Green Hills manual.

7.7.3 'Custom Buttons

The custom buttons were designed to allow you to safely start and stop the target drive. They do the following:

IDLEHALT This button executes a batch file which temporarily sets a breakpoint at the point in the background task where it halts the processor. When the processor gets there (it normally will when all tasks run out of work) the batch file sets the breakpoint back to its normal place (in the assertion 'trap'). This provides a safe way to stop the task code without halting in the middle of a servo ISR.

RESET This button executes a batch file which performs a hard reset on the target drive. First it stops the emulator. Then it sets various registers to first turn off servo interrupts, sets the VCM in Park mode, disables the spindle, sets the buffer clock to basic (40mhz), and issues a soft reset to the spindle motor. It then resets the emulator, re-enables NMI, and sets 'combo' breaks. **Don't click this button unless you have previously clicked 'PARK'.**

PARK This button executes a batch file which safely parks the read/write head. Because you may not have textured media, if the disk stops with out moving the head over a textured landing zone, the head may stick on the disk or be damaged the next time you spin up. **Always park the heads before resetting or exiting the emulator.** The batch file first stops the emulator, sets the VCM in Park mode, sets the spindle in brake mode, does a seven second delay, disables the spindle, resets the display length to 16 bytes, turns off servo interrupts, and sets some sfr's.

EXIT This button executes a batch file which gracefully exits the emulator. It stops the emulator, disables NMI generation, disables the spindle, sets the buffer clock to basic, issues a soft reset to the spindle motor block, resets the emulation cpu, disconnects from the 'QBox' software, and then quits the debugger. **Don't click this button unless you have previously clicked 'PARK'.**

7.7.4 Defining Your Own Custom Buttons

The *project*.rc file can be modified to create your own custom buttons. Be aware, however, that this file is retrieved via PVCS and could be overwritten if you do a 'get'. To create a button, follow the example in the file and add a line as follows:

button <BUTTONNAME> {target cbatch *filename*.str}

- 'button' is a keyword indicating a button creation.
- <BUTTONNAME> is the text that will appear in your button. Make it uppercase, to distinguish it from default buttons.
- Open brace, '{' indicates a string command follows.
- 'target' indicates to Multi that the text which follows is a command for the 'QBox' server.
- 'cbatch' is an QBox server command to execute a batch file.
- *filename*.str is the name of the batch file to execute.
- Close brace, '}' ends the string command.

An alternative method is as follows:

button <BUTTONNAME> < *filename*.btn

- 'button' is a keyword indicating a button creation.
- <BUTTONNAME> is the text that will appear in your button. Make it uppercase, to distinguish it from default buttons.
- The character '<' indicates that 'multi' will get the command input from a file.
- *filename*.btn is the command filename to execute.

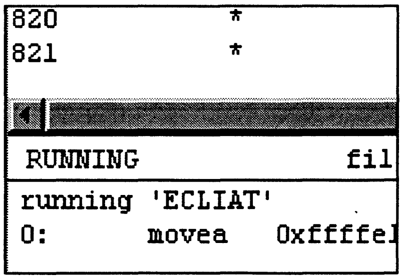
7.7.5 The User Configuration File

At the end of the *project*.rc file, you will see the command 'configure user.rc'. This tells 'Multi' to execute 'user.rc' as a configuration file. This is different from the *project*.rc file in that only configuration commands can be executed. See the 'MULTI Software Development Environment User's Guide' for more information on configuration files.

7.7.6 'Spinning' up the drive

From the initial download point, you can start the code executing by clicking once on the 'go' button. Now is the time to listen for spin-up (if your firmware is ready for that).

Look at the input panel. It should indicate that your firmware is 'running':



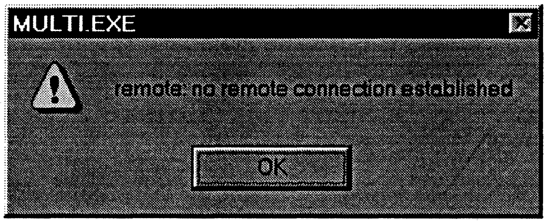
```
820             *
821             *

 RUNNING                 fil
 running 'ECLIAT'
 0:       movea    0xfffe
```

Figure 7-4. Confirmation that Your Firmware is running.

7.8 Troubleshooting

7.8.1 "No Remote Connection Established"

When you try to start the emulator, you may see the following dialog box:


MULTI.EXE

remote: no remote connection established

Figure 7-5. No Remote Connection Message

This is usually caused by one of the following:

- The emulator is not connected to your computer via its interface cable.
- The emulator is not powered up.
- The target drive is not powered up.
- A software problem has caused the 850ice server from starting up. One cause may be failure to configure your *project*.rc file with the following line:

remote 850ice32

The switch from Windows 3.1 tools to Win95 tools required this line to change from '850ice' to '850ice32', so you may not have done this edit.

7.9 'Combo' Mode

In order to allow the firmware to stop at a breakpoint without stalling the servo, NEC has implemented this mode of operation. 'Combo' mode allows any interrupts that are of a higher priority than the code at which the firmware is stopped to execute, maintaining the integrity of the interrupt hierarchy. Combo mode is documented in the NEC 850ice manual. To verify that servo is being serviced while the firmware is stopped, you can see the status line in the 'QBox' window flickering from 'Running' to 'Break'.

One of the features of the emulator, 'software' breakpoints, is not allowed when Combo mode is enabled.

7.10 Special Accommodation for the Multi-Stack Kernel

A special problem for breakpointing was introduced by the multi-stack kernel (see the "Multi-Stack Kernel" documentation). In the multi-stack kernel's method of operation, when an interrupt occurs the kernel is entered, it executes the interrupt service routine which could post an event, and then the scheduler is executed which could switch 'contexts' (all your registers will be changed to the values for a different task), all in one interrupt.

You don't want this happening when you are stopped at a breakpoint because it could change the values of your registers while you are stopped. So, 'Combo' mode was qualified with a special accommodation. The following structure of interrupt masking is implemented (masked means the interrupt will NOT be serviced and not masked means that it WILL be serviced):

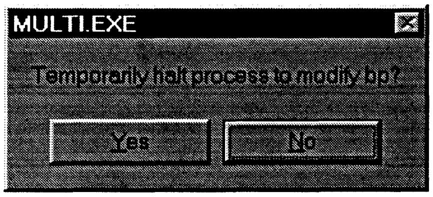
Signal Name	Purpose	Priority	Masking	Events
NMI	Servo interrupt	NMI	NOT Masked	NO Posting Allowed
INTCC11	Retract circuit	1	NOT Masked	NO Posting Allowed
INTCC13	Motor control	2	NOT Masked	NO Posting Allowed
INTSR0	Serial Receive	3	Masked	Posting Allowed
INTP00	Host Reset	5	Masked	Posting Allowed
INTP01	Unused		Masked	Posting Allowed
INTP02	Host interrupt or ECC error	7	Masked	Posting Allowed
INTCM4	Timer Compare	4	Masked	Posting Allowed
INTCSIO	DiskCallback (SW)	6	Masked	Posting Allowed

This allows essential NMI and timer (for the motor) interrupts to execute and keep the disk spinning but not interrupts such as host commands, which could cause a context switch.

Interrupts that are NOT masked must NOT post events and those which are masked may post events.

7.11 Setting Breakpoints

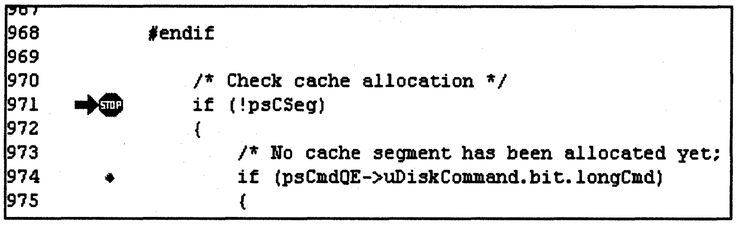
There are two ways to set breakpoints in the firmware; in the 'Multi' window, clicking on a green button, and in the 'QBox' window via event breakpoints.
You can set 'Multi' window breakpoints in the firmware without stopping the drive; sort of. The firmware really stops briefly (it prompts you with a dialog box before it stops the firmware to make sure it's OK), sets the breakpoint, and then resumes the firmware. This is normally OK because 'Combo' mode keeps the servo running the whole time.



Temporarily halt process to modify bp?

Figure 7-6. Setting Breakpoints While Running.

The breakpoint will be seen as a red stop sign over the point where the green button was:



```
    #endif

        /* Check cache allocation */
        if (!psCSeg)
        {
            /* No cache segment has been allocated yet;
            if (psCmdQE->uDiskCommand.bit.longCmd)
            {
```

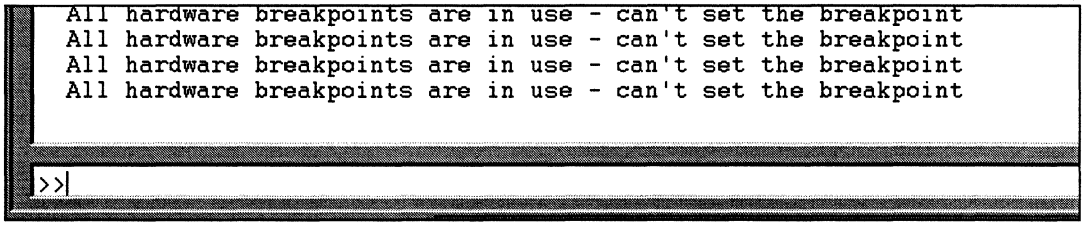
Figure 7-7. A Set Breakpoint in the Multi Debugger Window

7.12 Warning! Setting Too Many Breakpoints!

Each time you set a breakpoint by clicking on a green button in the source code panel, you use one of the instruction execution events (There is a feature for software breakpoints but this is disabled because of the requirement for combo breaks). Since there are only 12 instruction execution events available, you can exhaust them.

The problem is that **the Multi debugger won't tell you when you run out of breakpoints!** Instead, a message will appear in the 'QBox' window indicating that you have used all the hardware breakpoints. This message does not appear until the code **runs,** however, so if you are stopped the message won't appear until you click 'go'.

The red 'stop sign' will still appear in the Multi source code panel.



```
All hardware breakpoints are in use - can't set the breakpoint
All hardware breakpoints are in use - can't set the breakpoint
All hardware breakpoints are in use - can't set the breakpoint
All hardware breakpoints are in use - can't set the breakpoint

>>
```

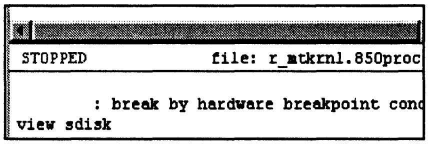
Figure 7-8. Too Many Breakpoints Set Error Messages

7.13 Viewing Data

You can view data structures symbolically using the 'Multi' window. **In order to do this the firmware must be stopped.** There are three methods for viewing data:

7.13.1 'View' Method

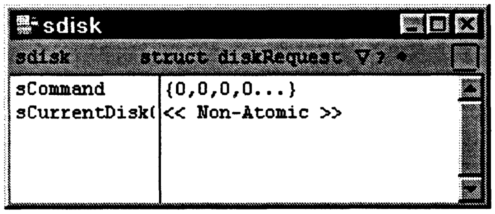
To view the 'sDisk' structure enter 'view sdisk' (not case sensitive) while the firmware is stopped:



```
STOPPED                      file: r_mtkrnl.850proc

           : break by hardware breakpoint cond
view sdisk
```

Figure 7-9. Enter 'view sdisk'

'Multi' will display the following:

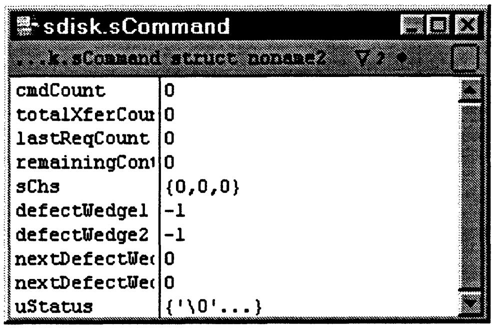


```
sdisk          struct diskRequest
sCommand       {0,0,0,0...}
sCurrentDisk(  << Non-Atomic >>
```

Figure 7-10. Symbolic Display of sDisk

7.13.2 Double Click Method

If you double click on a structure within this window, it will be displayed symbolically:



```
...k.sCommand struct noname2
cmdCount       0
totalXferCou   0
lastReqCount   0
remainingCon   0
sChs           {0,0,0}
defectWedge1   -1
defectWedge2   -1
nextDefectWe(  0
nextDefectWe(  0
uStatus        {'\0'...}
```

Figure 7-11. The sCommand Structure within sDisk.

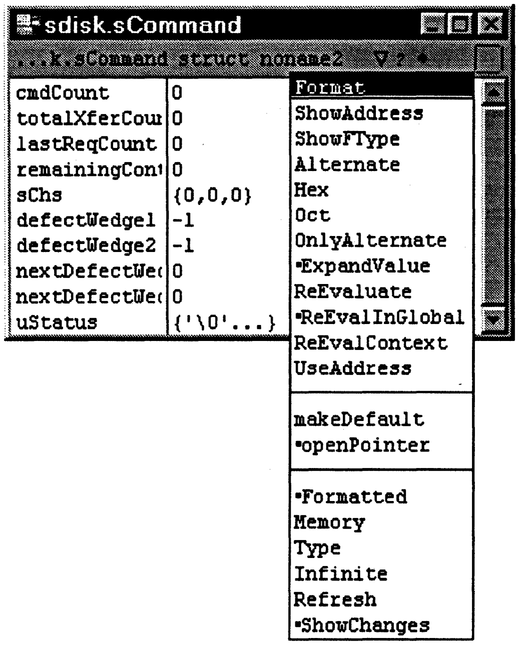
You can keep double clicking on all the structures down to the bit definition level.

7.13.3 Highlighting a Data Structure Method

If you wish to see the contents of a data structure more quickly, you can use the mouse to highlight a data structure displayed in the Multi source code window. The data structure and its contents will be displayed in the Multi input window.

7.14 Changing the Data Display from Decimal to Hex (and other forms)

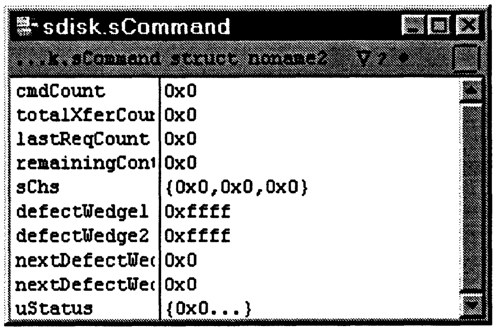
The data displays default to signed decimal, which is very inconvenient for firmware engineers. You can change the form of data display by clicking on the inverted triangle on the data display's menu bar and then clicking on 'Hex':



cmdCount	0
totalXferCou	0
lastReqCount	0
remainingCon	0
sChs	{0,0,0}
defectWedge1	-1
defectWedge2	-1
nextDefectWe	0
nextDefectWe	0
uStatus	{'\0'...}

Figure 7-12. Data Display Format Change Menu

You may then click on 'Hex' and 'OnlyAlternate' and you get a display like the one below:



cmdCount	0x0
totalXferCou	0x0
lastReqCount	0x0
remainingCon	0x0
sChs	{0x0,0x0,0x0}
defectWedge1	0xffff
defectWedge2	0xffff
nextDefectWe	0x0
nextDefectWe	0x0
uStatus	{0x0...}

Figure 7-13. Hex/OnlyAlternate Data Display

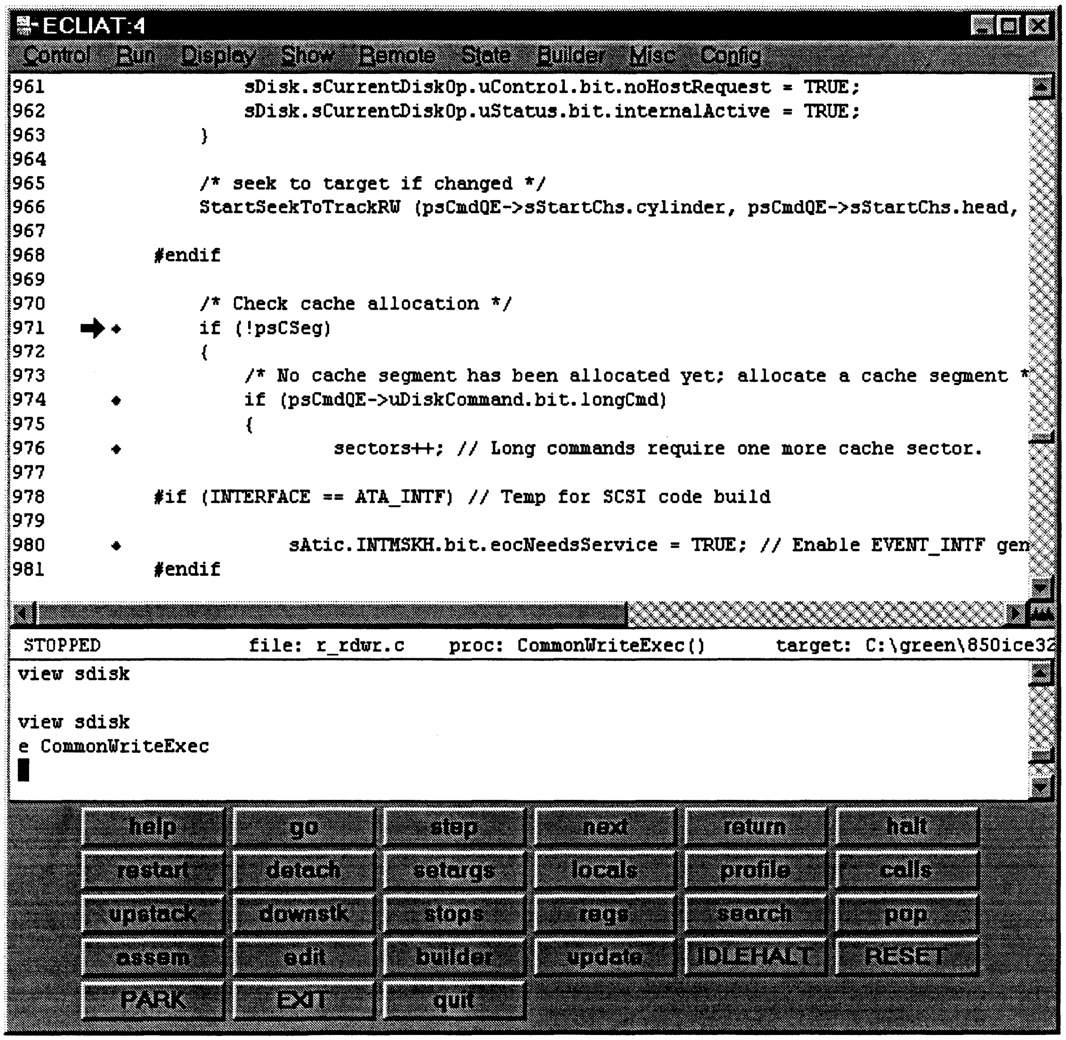
7.15 Making Hex the Default Display

It's really inconvenient to see your data in decimal mode but Green Hills won't change that because their other customers prefer this default. However, you can ensure that your data comes up with a hex display by entering the following configuration command in your *project*.rc file:

configure viewdef=Alternate Hex ExpandValue

7.16 Viewing a Source Code Module

You can bring any function in a source code module into view in the 'Multi' window so that you can set breakpoints or view assembly code by typing 'e *functionname'*:



```
                sDisk.sCurrentDiskOp.uControl.bit.noHostRequest = TRUE;
                sDisk.sCurrentDiskOp.uStatus.bit.internalActive = TRUE;
            }

            /* seek to target if changed */
            StartSeekToTrackRW (psCmdQE->sStartChs.cylinder, psCmdQE->sStartChs.head,

        #endif

            /* Check cache allocation */
            if (!psCSeg)
            {
                /* No cache segment has been allocated yet; allocate a cache segment *
                if (psCmdQE->uDiskCommand.bit.longCmd)
                {
                        sectors++; // Long commands require one more cache sector.

        #if (INTERFACE == ATA_INTF) // Temp for SCSI code build

                    sAtic.INTMSKH.bit.eocNeedsService = TRUE; // Enable EVENT_INTF gen
        #endif
```

```
 STOPPED            file: r_rdwr.c    proc: CommonWriteExec()      target: C:\green\850ice32
```

```
view sdisk

view sdisk
e CommonWriteExec
```

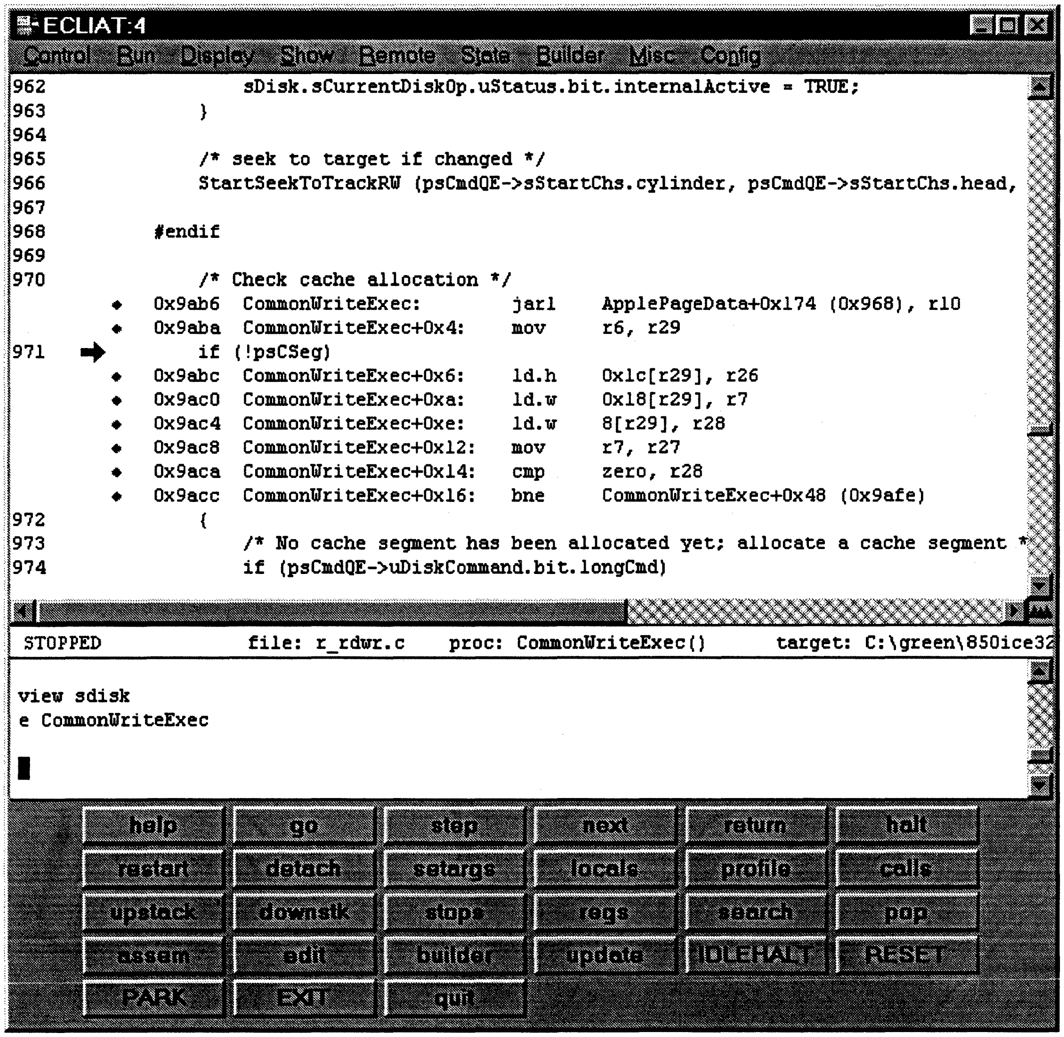
Figure 7-14 . Viewing a Function in a Source Code Module

By using the syntax 'e *functionname*', you instruct Multi to display the function source code and to allow you to work in debugging mode; you will see breakpoint buttons, you can see assembly language, etc.

Another syntax is to type 'e *filename.c*'. This will display the source code file in an editing mode. You will not see breakpoint buttons. However, you can click on the 'assem' button to see assembly language. This will switch you to the debugging mode.

7.17 Viewing Assembly Language Code

You can view the assembly language produced by the compiler and running in the emulator within the source code module by clicking on the 'assem' button:



```
                sDisk.sCurrentDiskOp.uStatus.bit.internalActive = TRUE;
            }

            /* seek to target if changed */
            StartSeekToTrackRW (psCmdQE->sStartChs.cylinder, psCmdQE->sStartChs.head,

        #endif

            /* Check cache allocation */
    ◆   0x9ab6  CommonWriteExec:        jarl    ApplePageData+0x174 (0x968), r10
    ◆   0x9aba  CommonWriteExec+0x4:    mov     r6, r29
  ➡         if (!psCSeg)
    ◆   0x9abc  CommonWriteExec+0x6:    ld.h    0x1c[r29], r26
    ◆   0x9ac0  CommonWriteExec+0xa:    ld.w    0x18[r29], r7
    ◆   0x9ac4  CommonWriteExec+0xe:    ld.w    8[r29], r28
    ◆   0x9ac8  CommonWriteExec+0x12:   mov     r7, r27
    ◆   0x9aca  CommonWriteExec+0x14:   cmp     zero, r28
    ◆   0x9acc  CommonWriteExec+0x16:   bne     CommonWriteExec+0x48 (0x9afe)
            {
                /* No cache segment has been allocated yet; allocate a cache segment *
                if (psCmdQE->uDiskCommand.bit.longCmd)
```

```
STOPPED             file: r_rdwr.c    proc: CommonWriteExec()       target: C:\green\850ice32
```

```
view sdisk
e CommonWriteExec

▌
```

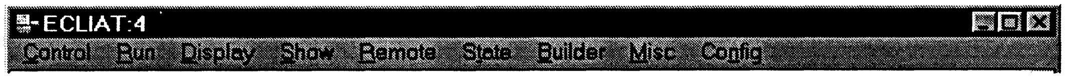
Figure 7-15. Click on the 'assem' Button to See Assembly Language Code

You may now set breakpoints and step through the code at the assembly language level. **It is important that you work at this level when you get into the detail level of debugging.** The code optimizer will move the assembly instructions to unexpected locations relative to your source code. In order to fully understand what's really going on in your code when you get to the source of your bug, you need to understand the assembly language and the results of the compiler. You will often find your bugs in this area.

To return to source code level debugging click on 'assem' again.

7.18 The 'Multi' Menu Bar

The 'Multi' menu bar provides some new tools. Here we will overview a couple of useful features. The menu bar appears as follows:



7.18.1 Control

QuitAll This selection allows you to exit the emulator. A safer method, however, is to use the **EXIT** button.

7.18.2 Display

UpStack This allows you to display the path that called the code you are currently displaying. You can repeatedly click UpStack, going further and further back into the calls that have occurred.

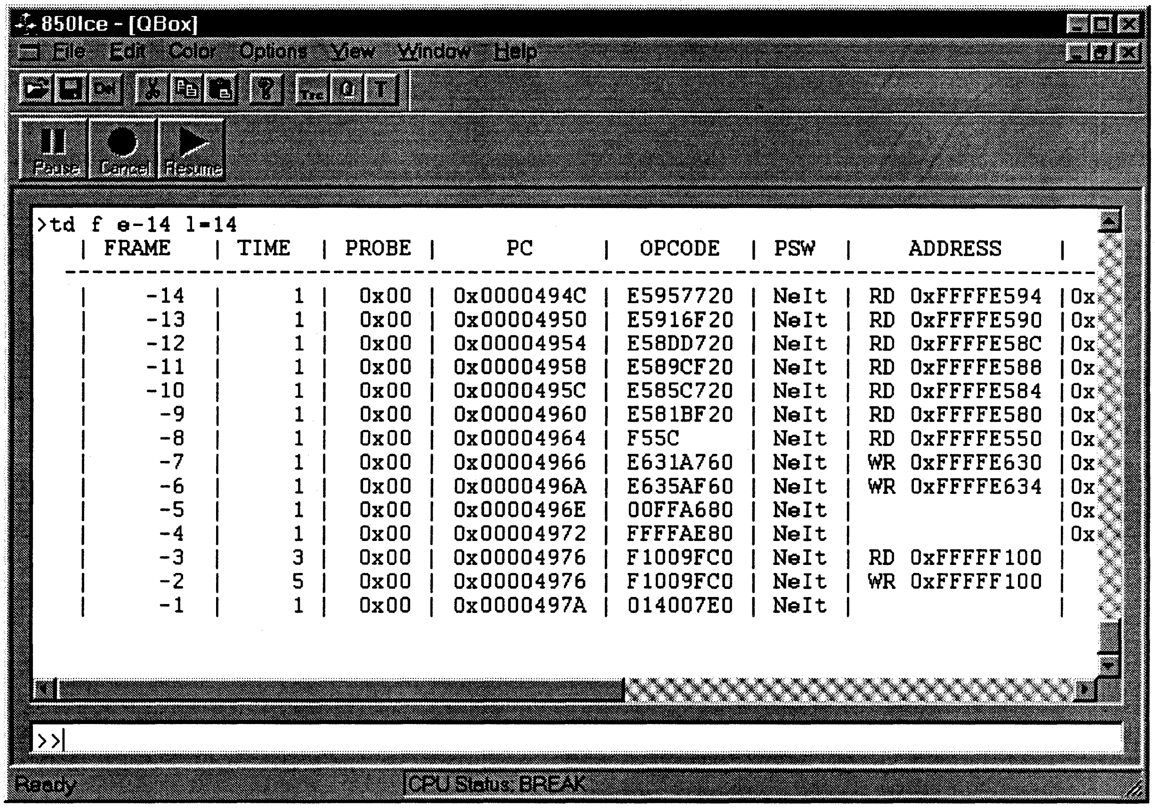
DownStack If you had previously done UpStack, you can click this to back down the calls until you get to the code that is currently executing.

DisplayPC This allows you to display the code at the current PC location. It's a good way to get back to where you are executing after you have done many UpStack operations.

7.18.3 Config

This allows you to configure your debugger in ways too numerous to mention. Check it out.

7.19 Trace Dumps



```
>td f e-14 l=14
   | FRAME     | TIME    | PROBE |      PC      |  OPCODE  | PSW  |      ADDRESS       |
  -----------------------------------------------------------------------------------------
   |      -14  |       1 |  0x00 | 0x0000494C  | E5957720 | NeIt | RD  0xFFFFE594  |0x
   |      -13  |       1 |  0x00 | 0x00004950  | E5916F20 | NeIt | RD  0xFFFFE590  |0x
   |      -12  |       1 |  0x00 | 0x00004954  | E58DD720 | NeIt | RD  0xFFFFE58C  |0x
   |      -11  |       1 |  0x00 | 0x00004958  | E589CF20 | NeIt | RD  0xFFFFE588  |0x
   |      -10  |       1 |  0x00 | 0x0000495C  | E585C720 | NeIt | RD  0xFFFFE584  |0x
   |       -9  |       1 |  0x00 | 0x00004960  | E581BF20 | NeIt | RD  0xFFFFE580  |0x
   |       -8  |       1 |  0x00 | 0x00004964  | F55C     | NeIt | RD  0xFFFFE550  |0x
   |       -7  |       1 |  0x00 | 0x00004966  | E631A760 | NeIt | WR  0xFFFFE630  |0x
   |       -6  |       1 |  0x00 | 0x0000496A  | E635AF60 | NeIt | WR  0xFFFFE634  |0x
   |       -5  |       1 |  0x00 | 0x0000496E  | 00FFA680 | NeIt |                 |0x
   |       -4  |       1 |  0x00 | 0x00004972  | FFFFAE80 | NeIt |                 |0x
   |       -3  |       3 |  0x00 | 0x00004976  | F1009FC0 | NeIt | RD  0xFFFFF100  |
   |       -2  |       5 |  0x00 | 0x00004976  | F1009FC0 | NeIt | WR  0xFFFFF100  |
   |       -1  |       1 |  0x00 | 0x0000497A  | 014007E0 | NeIt |                 |

>>|
```

Figure 7-16. A typical Trace Dump Using the QBox.

There are a variety of trace dumps, filters, and sections. Note that the 'QBox' is used for trace and memory dumps. These are documented in the NEC 850serv manual.

7.19.1 New Sectional Trace Method

One interesting addition is the ability to do sectional traces by two methods. One method is familiar to you. It is a sectional trace which has a start and end:

brs 1 a=0x4910
brs 2 a=0x5910
t s=brs1 e=brs2

This kind of trace is triggered when the starting point is executed and turned off when the ending point is executed. The advantage to this is that it includes all calls and ISRs. The disadvantage of this is that it can be cluttered with servo interrupts and such.

The new method is tracing within one or more ranges:

brs 1 a=0x4910-0x5910
brs 2 a=0x6645-0x7700
t q=brs1 | brs2

The above trace is 'qualified' by the two address ranges specified by brs1 and brs2

Note: You must declare brs's with a space between their name and number. The qualify syntax requires a Boolean 'or' operator (|).

This trace will not include any code that is not within the given ranges. The starting and ending points need not be executed to trigger the trace; they need only be within the range. This will exclude any called code outside the range, such as servo and ISRs.

7.19.2 Trace Searches

Another handy new feature is the ability to search your trace buffer for a particular address. This will display every occurrence of an instruction executed at that address. This is a good way to find out how many times a particular function or instruction is executed at a given point in time. The syntax is:

```
tsearch a=0xnnnn
```

7.20 Assertion Traps

A good coding practice is to lace your code with 'assert' statements. An example would be **assert (lba >= 0);** The expression in parentheses must be non-zero or the emulator stops at a place called 'trap'. Upon start-up, brs11 is set to break on 'trap'.

Another way to get to the 'trap' is to place the macro 'SHOULDNT_GET_HERE' at strategic places in your code.

If (or should I say **when**) your code stops in an assertion trap, your first desire will be to find out how you got there. Click on the Multi tool bar entry, **Display** and then select **UpStack** from the pull down menu. This displays the 'caller' of the assertion trap to you.

7.21 Experiment!

This document gives you an overview of some of the features of this emulator. There are many, many features and capabilities to explore. The NEC 850ice manual is comprehensive. However, there is scant Green Hills documentation on the 'Multi' debugger window for Win95. Start experimenting and read your manuals!

7.22 Starting Diag

The following description is an example only. Every debugging environment is different. Typically, you will have a directory in your Diag station with a batch file that looks something like this:

```
diag  -ata -s ecliat.atd
```

The 'atd' file is a Diag macro file that you have retrieved from PVCS for your project. Once Diag is started, typically you will load ramware as follows:

```
>> rw
```

You then may then write sector descriptors, channel parameters, and configuration pages using a macro like this:

```
>> init
```

In any case, your drive should be spun up and as ready to debug as it can be.

7.23 Overlay Management

A strategy has been developed to support debugging of overlays. It allows you to set breakpoints in overlays and allow the debugger to differentiate addresses that may be at exactly the same place in external RAM but be part of different overlays.

We do this by placing overlays in RAM sections ('sections' are set up by the linker) that are 'mirrors' of external RAM (see 'Eclipse Hardware and Firmware Memory Map'). These mirror sections have values in the high order bits of their addresses that are ignored by the actual drive hardware so they load into the same external RAM locations addressed by the low order bits. Here are the address ranges in question:

- External RAM is at locations 0x100000 - 0x13ffff.
- Overlay sections are at locations 0x1100000 - 0xf100000.

So, bits 24-27 of the address are the overlay number. When the overlay loader tries to load an overlay at location 0x1100000, it really loads at location 0x100000 in RAM.

The new problem becomes how to use hardware instruction execution events (which are necessary instead of software breakpoints because of Combo mode) to stop at breakpoints in overlays.

We solve this problem by placing bits 24-27 of the address on the external trigger pins of the emulator. We invented a board that does this (the overlay manager board). It goes between the emulator pod and the microprocessor.

When an overlay is loaded, the overlay manager firmware writes the values that are to be in bits 24-27 to a special array located in a place recognized by the overlay manager board. It then latches the appropriate value onto the four pin connections. These pin connections are wired to the external pin connections of the emulator.

We can then set an instruction execution event for our example overlay as follows:

```
brs 1 a=0x100000 P=0x01
```

When the PC reaches 0x0100000, and the overlay address is 1 the emulator will stop.

7.23.1 Multi Support For Overlay Breakpoints

The Multi Debugger allows you to click on a green dot to set a breakpoint in an overlay just as you would set a breakpoint anywhere else. It does this by sending an instruction execution event to the 'QBox' server just like the one above. The overlay manager does a read to one of 16 consecutive addresses. When the overlay board decodes the upper 20 bits of address, it latches the low 4 bits as the overlay number.

Chapter 8 Eclipse Memory Map (Hardware and Firmware)

8.1 Overview

This document will describe both the Eclipse AT product's hardware memory map and the way in which the firmware has allocated memory resources. This document also covers the memory sections found in the firmware link map.

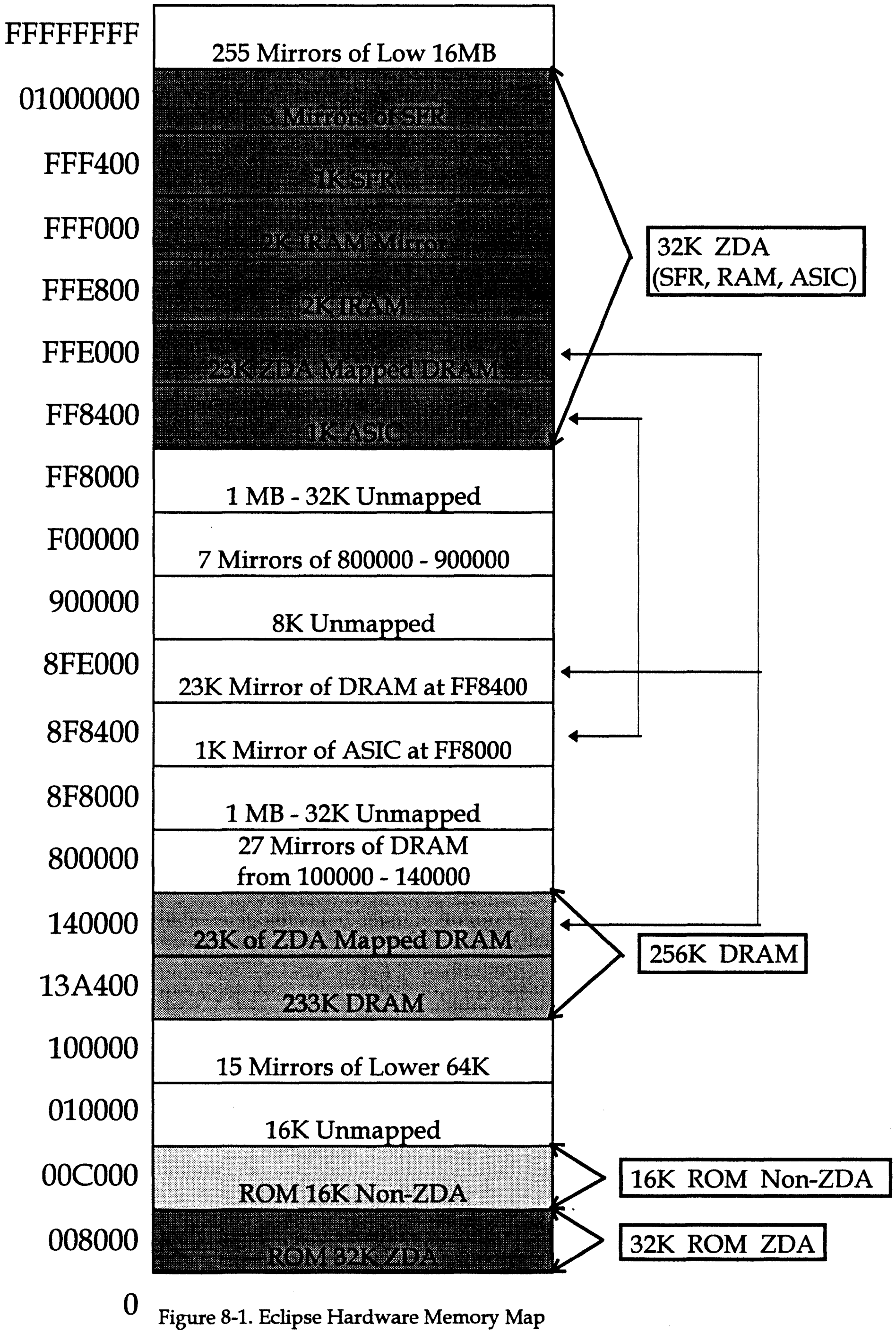


Figure 8-1. Eclipse Hardware Memory Map
(Refer to link file from SRC directory for latest updates on the memory map.)

8.2 'ZDA' - Zero Data Area Access Method

The Green Hills compiler uses this method of data access to reduce the size of instructions necessary to read and write memory. These instructions contain a sign-extended 16-bit offset from a register also specified in the instruction, providing a range of +/- 32K relative to the register. The processor contains a hard-wired register, r0, which always contains zero.

The compiler can take advantage of the 16-bit offset in the instruction and the register, which it does not have to initialize to access a range of data that is +/- 32K from zero. This is ZDA.

The Eclipse architecture takes advantage of this efficiency by placing most of ROM (for table access), all of internal RAM, some external RAM, and the ASIC all within this range (See Figure 8-1. Eclipse Hardware Memory Map).

Notice that a range of external RAM (the topmost 23K) also decodes into the ZDA space. This is where external RAM variables and tables are stored.

Memory not accessed via ZDA must be accessed by using two instructions to initialize a register pointer followed by the actual access based on a 16-bit (typical) offset.

8.3 'TDA' - Tiny Data Area Access Method

The microprocessor allows for even more compact instructions using a method similar to ZDA except that the offset in the instruction varies from 0 - 127 bytes to 0 - 255 for words and halfwords and the base register always must be the 'ep' (R30). This has no effect on the hardware map and is mentioned here only to help explain 'TDA' references later in this document. There is more discussion on this subject in 'C Coding Efficiency'.

8.4 Hardware Memory Components

8.4.1 ASIC

The ASIC registers are described in the Rainier-A or -S manual provided by the ASIC group. Individual blocks within the ASIC are referenced by structures named after them and assigned individual sections in the linker. (see the link map section of this document). They are accessible via the ZDA method.

8.4.2 Internal RAM

This 2K of RAM is internal to the V851 microprocessor. Access to this RAM is extremely fast; one cycle with one pipeline delay (see 'V851 Microprocessor'). Critical variables, stacks, and contexts (see 'Multi-Stack Kernel') are stored there.

Internal RAM is assigned many sections in the linker. Most of internal RAM is dedicated to the Multi-Stack Kernel's stacks and contexts. Other sections are used for critical variables (via ZDA) and Servo, and are only accessed in TDA mode by the assembly-language servo code. The TDA option is not used by the C code.

8.4.3 SFRs

These are Special Function Registers defined by the V851 microprocessor (see NEC's 'V851 Microprocessor' manual). They are accessible via the ZDA method.

8.4.4 ROM

ROM is where we keep power-up and performance code and tables. For constant access, only the first 32K can be accessed via the ZDA method. **NOTE: An attempt to access a constant via ZDA outside the 32K range could be the cause of a linker error.**

8.4.5 External RAM

The Eclipse platform provides for 256K of external RAM. This RAM is very slow to access (approximately 1 microsecond per byte when the disk and host are running) because access to it must be multiplexed with the disk and host channels.

23K of external RAM decodes into the 'ZDA' range, so this is where most of the global variables are stored. This is the topmost decoded RAM. This is where non-critical variables are kept.

8.4.6 Mirrors

Internal RAM and ROM have 'mirror' decodes which are caused by ignored address bits. Normally, these areas are not referenced by the firmware. However, a mirroring strategy is used by the overlay manager. Sections into which overlays will be loaded are defined in a range that mirrors external RAM. This allows the linker to define code that has overlapping addresses without reporting an error.

8.4.7 Unmapped

Unmapped areas have no hardware component at all and should never be referenced by the firmware.

8.5 Overview of Linker Sections

The following table contains addresses that could change frequently. It was taken from one snapshot of the Eclipse map file. Refer to your own link map to be certain of a particular section's location and size. However, this document will give you a general idea of the locations of important sections along with their descriptions.

Hardware	Name	Start	End	Size	Description
ROM	intvecs	0x00000000	0x00000153	0x00000154	Interrupt Vectors
ROM	romrozda	0x00000154	0x000006a3	0x00000550	Tables accessible via ZDA
ROM	. text	0x000006a4	0x00000c7f	0x000005dc	Green Hills Library Functions (Note 1)
ROM	romtext	0x00000c80	0x00009833	0x00008bb4	Power-up and Performance Code
ROM	compvec	0x0000bef0	0x0000bfef	0x00000100	Compressed Vector Table (Note 2)
ROM	ROMHdata	0x0000bff0	0x0000bfff	0x00000010	Default diskware download header (Note 3)
SFR	pinmask	0x000f0000	0x000f0000	0x00000000	Emulator Control Mask
EXRAM	DWVHdata	0x01100000	0x0110001f	0x00000020	Diskware Vector Table Header (Note 4)
EXRAM	romvec	0x00100020	0x001000a7	0x00000088	Un-Compressed Vector Table (Note 2)
EXRAM	dwvec	0x01100020	0x011007ff	0x000007e0	Diskware Vector Table (Note 4)
EXRAM	RESHdata	0x00100800	0x0010081f	0x00000020	Diskware Resident Code Header
EXRAM	restext	0x00100820	0x00105e07	0x000055e8	Resident Diskware
EXRAM	OvlyHdr	0x00107400	0x00107400	0x00000000	Place Holder for Overlay Header (Note 5)

EXRAM	OvlyText	0x00107420	0x00107420	0x00000000	Place Holder for Overlay Code (Note 5)
EXRAM	Ovl1Hdr	0x01107400	0x0110741f	0x00000020	Overlay 1 Header (Note 6)
EXRAM	Ovl1Text	0x01107420	0x01107b0b	0x000006ec	Overlay 1 Code (Note 6)
EXRAM	Ovl2Hdr	0x02107400	0x0210741f	0x00000020	Overlay 2 Header (Note 6)
EXRAM	Ovl2Text	0x02107420	0x02107927	0x00000508	Overlay 2 Code (Note 6)
EXRAM	Ovl3Hdr	0x03107400	0x0310741f	0x00000020	Overlay 3 Header (Note 6)
EXRAM	Ovl3Text	0x03107420	0x03107673	0x00000254	Overlay 3 Code (Note 6)
EXRAM	Ovl4Hdr	0x04107400	0x0410741f	0x00000020	Overlay 4 Header (Note 6)
EXRAM	Ovl4Text	0x04107420	0x0410781f	0x00000400	Overlay 4 Code (Note 6)
EXRAM	Ovl5Hdr	0x05107400	0x0510741f	0x00000020	Overlay 5 Header (Note 6)
EXRAM	Ovl5Text	0x05107420	0x0510786b	0x0000044c	Overlay 5 Code (Note 6)
EXRAM	WList	0x00107c00	0x00109bff	0x00002000	'W' List (Defect List)
EXRAM	Buffer	0x00109c00	0x0010a400	0x00000800	'Temp' Buffer (Note 7)
EXRAM	BfrCache	0x0010a400	0x0013a3ff	0x00030000	Read/Write Cache
EXRAM	ZdaBase	0x0013a400	0x0013ffff	0x00005c00	External RAM Mirrored in ZDA (Note 8)
EXRAM	FSDHdata	0x04108000	0x0410801f	0x00000020	File System Header (Notes 4, 9 & 10)
EXRAM	FSDir	0x04108020	0x04108063	0x00000044	File System Directory (Notes 4, 9 & 10)
EXRAM	DWLHdata	0x0410a400	0x0410a41f	0x00000020	Diskware Download Header (Notes 4, 11 & 12)
EXRAM	DWLtext	0x0410a420	0x0410a70f	0x000002f0	Diskware Download Code (Notes 4, 11 & 12)
EXRAM	DWLbss	0x0410a710	0x0410a75f	0x00000050	Diskware Download Vars. (Notes 4, 11 & 12)
EXRAM	BBLHdata	0x05110000	0x0511001f	0x00000020	Boot Loader Header (Notes 4, 12, & 13)
EXRAM	BBLtext	0x05110020	0x05110203	0x000001e4	Boot Loader Code (Notes 4, 12, & 13)
EXRAM	BBLbss	0x05110204	0x05110204	0x00000000	Boot Loader Variables (Notes 4, 12, & 13)
EXRAM	SSLHdata	0x0610a400	0x0610a41f	0x00000020	Selfscan Loader Header (Notes 4, 10, & 14)
EXRAM	SSLtext	0x0610a420	0x0610a553	0x00000134	Selfscan Loader Code (Notes 4, 10, & 14)
EXRAM	SSLbss	0x0610a554	0x0610a554	0x00000000	Selfscan Loader Variables (Notes 4, 10, & 14)
EXRAM	ModeD	0x0010d400	0x0010d46b	0x0000006c	Mode Page Defaults (Note 15)
EXRAM	ConfP	0x0010d800	0x0010ef47	0x00001748	Configuration Page Defaults (Note 15)
EXRAM	SSRCHder	0x00110400	0x0011041f	0x00000020	Selfscan Resident Code Header (Note 12)
EXRAM	SSRCtext	0x00110420	0x00111637	0x00001218	Selfscan Resident Code (Note 12)
EXRAM	SSO1Hder	0x00111c00	0x00111c1f	0x00000020	Selfscan Code Overlay 1 Header (Note 12)
EXRAM	SSO1text	0x00111c20	0x001149f7	0x00002dd8	Selfscan Code Overlay 1 Code (Note 12)
EXRAM	SSO1data	0x001149f8	0x00114a2f	0x00000038	Selfscan Code Overlay 1 Data (Note 12)
EXRAM	SSO1bss	0x00114a30	0x00114a63	0x00000034	Selfscan Code Overlay 1 Variables (Note 12)
EXRAM	SSO2Hder	0x01111c00	0x01111c1f	0x00000020	Selfscan Code Overlay 2 Header (Notes 4 & 12)
EXRAM	SSO2text	0x01111c20	0x01113b67	0x00001f48	Selfscan Code Overlay 2 Code (Notes 4 & 12)
EXRAM	SSO2data	0x01113b68	0x01113bab	0x00000044	Selfscan Code Overlay 2 Data (Notes 4 & 12)
EXRAM	SSO2bss	0x01113bac	0x01113c17	0x0000006c	Selfscan Code Overlay 2 Vars. (Notes 4 & 12)
EXRAM	SSO3Hder	0x02111c00	0x01111c1f	0x00000020	Selfscan Code Overlay 3 Header (Notes 4 & 12)
EXRAM	SSO3text	0x02111c20	0x0211229b	0x0000067c	Selfscan Code Overlay 3 Code (Notes 4 & 12)
EXRAM	SSO3data	0x0211229c	0x0211229c	0x00000000	Selfscan Code Overlay 3 Data (Notes 4 & 12)
EXRAM	SSO3bss	0x0211229c	0x021122a8	0x0000000c	Selfscan Code Overlay 3 Vars. (Notes 4 & 12)
EXRAM	SSRDHder	0x00115c00	0x00115c1f	0x00000020	Selfscan Resident Data Header (Note 12)
EXRAM	SSRDdata	0x00115c20	0x00115cf7	0x000000d8	Selfscan Resident Data (Note 12)
EXRAM	SSRDbss	0x00115cf8	0x00115d54	0x0000005c	Selfscan Resident Variables (Note 12)
EXRAM	SSCMDRST	0x00115e00	0x00116200	0x00000400	Selfscan Command & Result Overlay (Note 12)
EXRAM	SSCMDHTR	0x00116200	0x001163ff	0x00000200	Selfscan Command History Buffer (Note 12)
EXRAM	SSDOvlyA	0x00116400	0x001173ff	0x00001000	Selfscan 'script' Buffer (Note 12)
EXRAM	SSDOvlyB	0x00117400	0x001193ff	0x00002000	Selfscan 'result' Buffer (Note 12)
EXRAM	SSDOvly2	0x01116400	0x011193ff	0x00003000	Selfscan 'hard defect list' (Notes 4 & 12)
EXRAM	SSDOvlyC	0x01119400	0x0111a3ff	0x00001000	Selfscan 'soft/servo error lists/tables'
EXRAM	SSDOvly3	0x02116400	0x021199ff	0x00003600	Selfscan 'data tables overlay' (Notes 4 & 12)
OVMGR	ovlymgr	0xf783f0	0xf783ff	0x00000010	Overlay Manager Array (Note 16)
IRAM	.sdata	0xfffefc00	0xfffefc00	0x00000000	Small Data Area is Unused
ASIC	bcb	0xffff8000	0xffff803b	0x0000003c	Buffer Controller ASIC Block
ASIC	sser	0xffff80a8	0xffff80ab	0x00000004	Serial ASIC Block
ASIC	suproc	0xffff80b0	0xffff80bf	0x00000010	Microprocessor Interface ASIC Block
ASIC	fmtr	0xffff8100	0xffff813f	0x00000040	Formatter ASIC Block
ASIC	apll1	0xffff8140	0xffff8143	0x00000004	APLL ASIC Block #1
ASIC	apll2	0xffff8148	0xffff814b	0x00000004	APLL ASIC Block #1

ASIC	sadc	0xffff8150	0xffff8157	0x00000008	AtoD Converter ASIC Block
ASIC	smotor	0xffff8160	0xffff8173	0x00000014	Motor Control ASIC Block
ASIC	atic	0xffff8180	0xffff81e7	0x00000068	AT Interface ASIC Block
ASIC	sthermal	0xffff8200	0xffff820f	0x00000010	Thermal Asperity ASIC Block
ASIC	stuna	0xffff8240	0xffff828f	0x00000050	TNA ASIC Block
ASIC	secc	0xffff8300	0xffff832f	0x00000030	ECC ASIC Block
EXRAM	svdata	0xffffa400	0xffffa4eb	0x000000ec	Servo Data
EXRAM	svbss	0xffffa4ec	0xffffadff	0x00000914	Servo Variables
EXRAM	romzdata	0xffffae00	0xffffaf4b	0x0000014c	Initialized Global ZDA Tables & Vars (Note 17)
EXRAM	romzbss	0xffffaf4c	0xffffcf03	0x00001fb8	Uninitialized Global ZDA Variables (Note 18)
EXRAM	CVecdata	0xffffd200	0xffffd30f	0x00000110	Command Decode Tables
EXRAM	resrozda	0xffffd310	0xffffd363	0x00000054	Resident Diskware's Global Tables (Note 19)
EXRAM	reszdata	0xffffd364	0xffffd737	0x000003d4	Resident Diskware's Global Data (Note 20)
EXRAM	reszbss	0xffffd738	0xffffd8c7	0x00000190	Resident Diskware's Global Variables
EXRAM	bgstbtm	0xffffdd80	ffffdd80	0x00000000	Background Task Stack Bottom (Note 21)
EXRAM	bgsttop	0xffffdf00	0xffffdf03	0x00000004	Background Task Stack Top (Note 21)
EXRAM	excstbtm	0xffffdf04	0xffffdf04	0x00000000	Exception Task Stack Bottom (Note 21)
EXRAM	excsttop	0xffffdf80	0xffffdf83	0x00000004	Exception Task Stack Top (Note 21)
EXRAM	cmdstbtm	0xffffdf84	0xffffdf84	0x00000000	Command Task Stack Bottom (Note 21)
IRAM	cmdsttop	0xffffe100	0xffffe103	0x00000004	Command Task Stack Top (Note 21)
IRAM	prestbtm	0xffffe104	0xffffe104	0x00000000	Pre-Processor Task Stack Bottom (Note 21)
IRAM	presttop	0xffffe180	0xffffe183	0x00000004	Pre-Processor Task Stack Top (Note 21)
IRAM	isrstbtm	0xffffe184	0xffffe184	0x00000000	ISR Stack Bottom (Note 21)
IRAM	isrsttop	0xffffe1f8	0xffffe1fc	0x00000004	ISR Task Stack Top (Note 21)
IRAM	irdata	0xffffe1fc	0xffffe48f	0x00000294	Internal Global Variables (Note 22)
IRAM	.tdata	0xffffe490	0xffffe577	0x000000e8	TDA Servo Variables (Note 23)
IRAM	.tidata	0xffffe578	0xffffe637	0x000000c0	TDA Servo Variables (Note 24)
IRAM	svopatch	0xffffe700	0xffffe6ff	0x00000100	Servo Code Patch Area
SFR	atlas	0xfffff000	0xfffff3ff	0x00000400	V851 Microprocessor Control Registers

8.6 Hardware Legend

- **ROM** : Internal to the V851, 48K Read-only memory.
- **SFR** : Special function Registers, associated with V851 functions.
- **EXRAM** : External DRAM, 256K. Slow access, located on logic board and accessed via ASIC decode.
- **OVMGR** : A block of addresses used by the overlay board. (See Note 16.)
- **IRAM** : Internal to the V851, 2K Fast-access RAM.
- **ASIC** : The drive's main ASIC.

8.6 Notes

1. We want to avoid using Green Hills Library Routines. This is a default section which we do not name ourselves, but in which library functions go if they are invoked by the compiler. One objective is to eliminate this section.

2. This is a compressed vector table that is created by the build process 'makeprom' program. It compensates for a bug in the Green Hills compiler which cannot create a defined halfword containing the address of a function. Instead, we create a table of 'jump relatives' to the functions which is defined in external RAM in the location of the default vector table. This allocates the needed space in RAM (section 'romvec') but we don't want to use up that much space in ROM. So makeprom compresses it (strips of the jump relative part, leaving the offset) and makes it load into the smaller ROM space, 'compvec'. On power-up, it is decompressed (adding the jump relatives and adjusting the offsets) back into 'romvec'.

3. This header field is required as part of the diskware download process and must always appear at the beginning of each code element. The makeprom utility will modify some of these fields.

4. These sections are defined in a range that mirrors external RAM. This allows the linker to define code that has overlapping addresses without reporting an error.

5. The overlay system uses overlapping mirrored sections (see Note 4). These sections are place holders for this system. They allocate no space.

6. These are the actual overlays. They load into the same physical location in RAM, vary in size according to their code content, and vary in header content (see Notes 4 & 5).

7. This is a buffer area that can be used by functions to read/write into, etc. without any cache functions being in place.

8. This section of external DRAM is seen in two places. It appears at the top of the 256K of external RAM and it decodes to the range accessible via ZDA.

9. This is the main directory for the file system.. This data structure is removed from the final output file by gover and used to build the diskware upd file.

10. Note that this section overlaps with the W-list. There should be no need for them both in memory at the same time.

11. Diskware loader firmware that will be downloaded to the drive and executed to load ramware or diskware.

12. Note that this section overlaps with the Read/Write Cache buffer. There should be no need for them both in memory at the same time.

13. This is the intermediate loader that is read up by the rom base boot loader. This loader is in memory only temporarily and is responsible for reading up diskware, the defect lists, the primary overlay, and the vector table.

14. Selfscan firmware loader that will be downloaded to the drive and executed to load Selfscan modules.

15. The object for these values is extracted from the object file by 'Gover' and inserted into the 'scd' file (this contains the configuration page values). For this reason, the location of this section is meaningless.

16. This is a region decoded by the overlay manager board. When there is a memory read in the region, the overlay board laches the 4 low order address bits which encode the overlay number. Pins are connected by wires to the emulator's external probe pins, enabling debugging of overlay code.

17. These are tables and variables that are initialized in various places in the 'C' code. The initial values are collected into ROM and on power-up, copied to the external RAM locations.

18. These are variables that start with an initial value of zero. This is a standard in 'C'. So, you can declare any 'bss' variable uninitialized and assume that its starting value (on power-up).

19. These are ZDA accessible Read Only global tables and values that would be ordinarily ROM tables except that they are part of diskware, so they are loaded into external RAM.

20. These are ZDA accessible read/write global tables and values.

21. The size of the stacks is not seen based on a 'size' in the linker map. You can determine a stack size by subtracting its bottom from its top (normally, you would subtract the top from the bottom but, since the stacks are in the ZDA space in a negative direction from zero, we subtract a more negative number from a less negative number). A stack can even cross sections. For example, the command stack can grow from internal RAM into external RAM. Each of these stacks belong to the Multi-Stack Kernel. (See 'Multi-Stack Kernel').

22. These are uninitialized Global ZDA accessible variables kept in internal RAM for fast access.

23. These are initialized TDA accessible Servo Variables.

24. These are initialized TDA accessible Servo Variables.

(This page is intentionally left blank.)

Chapter 9 System Cylinders

9.1 Introduction

"N" tracks after physical cylinder 0 on all drives are reserved for servo, firmware and test usage. This creates a logical cylinder 0 that starts after these reserved cylinders. These reserved cylinders contain both executable code and drive data. Customers cannot access these cylinders; they are only accessible with physical or file system commands, which are protected diagnostic commands.

9.2 File System

The "File System" name is used to emphasize the attempt to produce a layer of abstraction between the main firmware and the system cylinders (rather than to describe a true file system). This abstraction relieves the firmware of any specific knowledge regarding the location and number of the copies of a file within the file system. The file system also masks the differences between the files used to boot up the system and the rest of the files in the system. Firmware is not completely oblivious to the nature of files within the file system, primarily due to the requirement that all files have a header. This header exists at the beginning of each file and consumes 16 bytes for boot (ROM header only) and 32 bytes (ROM + diskware header) for normal files of the available data space. It is used during diskware download to validated individual elements (files) and whenever a file is read up from the system cylinders it is validated using the data contained within the header.

ROM Header		
Byte	Contents	Example
1	File Identifier	0xE0
2	Product Code	0x1F
3-5	ROM Version	A6K
6	File Length in Sectors	6
7-8	ROM Checksum	0x940C
9-12	Execution Address	0x0010E820
13	Flash Count	0
14	Flags	0x08
15-16	File Checksum	0x98F1

Diskware Header		
Byte	Contents	Example
1-4	Load Address	0x0010E800
2-6	Diskware Version	0200
7-11	Reserved	0
12-16	Reserved	0

The nature of a drive with diskware demands that the static code in ROM know as little about the frequently changed diskware as possible. This is achieved by having the ROM load a small section of code from a well-defined place on disk and allowing that code to bring up the remaining diskware. This small section of code is part of the boot files that are treated differently from the normal files in the file system. This difference is due to the different capabilities of the system when only ROM code is available.

The files within the file system reside on heads 0 and 1, on logical cylinders -2 through -11; this provides sufficient space for four copies of the boot files and the normal files. The choice of four copies of each normal file is arbitrary and could be two or three with a minor algorithmic change.

9.2.1 Boot Files

The boot files together represent the subset of files (3 files) that are read in prior to loading diskware and as such can depend only on the ROM for support routines. The ROM loader is the one part of ROM code that has to know where files are found in the file system. An algorithm that knows the range of file system cylinders and the location of the boot files within each track provides this knowledge. The starting location of each copy of the boot file group is shown in the following table:

Cylinder	Head	Sector
-2	0	0
-3	0	0
-2	1	0
-3	1	0

This table represents the starting locations of each copy of the boot loader, which is the first of the three files in the boot block. The remaining boot files are abutted to the end of each copy of the boot loader. The following table illustrates the layout of one copy of the three boot files:

Sector 0	Sector 2	Sector 3
Boot Loader	Directory	System Defect List

9.2.1.1 Directory

This boot file contains the directory for all normal (non-boot) files in the file system. The ROM firmware does not directly read up this file (it's loaded by the disk based boot loader), so its format can be changed whenever diskware changes merit it. The directory is currently defined to fit within a single sector and is implemented as an array as depicted below:

Type (1 byte)	Length (1 byte)	Valid Copy Count (1 byte)
RESIDENT	64	4
VECTOR_TABLE	4	4
CONFIG_PAGES	32	4
PLIST	8	4
SelfScan Results	16	2
•	•	•
NOT_A_FILE	0	0

The preceding array can support 160 files (array size plus header equals 512 bytes) in addition to the three boot files supported by the boot directory. The type field identifies the file that this entry describes and is used when the directory is searched for a particular file. The length field contains the file length in sectors. The "Valid Copy Count" field indicates how many copies of a file must be successfully written during the test process and drive initialization. The last entry in the above table is an unused one as indicated by the NOT_A_FILE entry for the file type.

As can be seen in the above directory example, there is no location information for the files. Maintaining a starting CHS (or two) for each file would greatly increase the size of the directory and also becomes a maintenance issue, as an increase in the size of a file would require the update of numerous directory entries. The firmware determines on the fly where a file resides within the system cylinder area. The file system area is divided into four areas (one for each copy of a file) that consist of the files abutted together based on their order within the directory. The firmware merely adds up the size of each file as it's scanning the directory until it finds the desired file. Naturally it has to handle wrapping files at the end of a track (files are not allowed to span tracks, so if a file won't fit on the end of a track it starts at the beginning of the next track). Along with reserving space at the end of each track for inline sparing in the system cylinder area.

9.2.1.2 Boot Loader

The boot block loader handles the transition from ROM based to diskware based code. The ROM loader reads this intermediate stage loader from a fixed place on disk into a fixed place in memory. The emphasis on a "fixed place" is to reinforce the fact that the ROM must know where this loader is on disk and where to place it in memory. This prevents any location or size changes to this file without a corresponding ROM turn. Fortunately this is not a significant restriction as the current loader only occupies less than a single sector of the two allocated, leaving plenty of room for additional features.

After being loaded and validated, control is passed to this loader which first proceeds to read up the remaining boot files (the directory and the system defect list). Next the resident block, primary overlay, defect lists, and the vector table are read in. Control is then passed to the resident init function, which does the final initialization prior to enabling diskware.

9.2.1.3 System Defect List

This is a small fixed format defect list that covers the range of negative cylinders that the file system uses. This list provides support for all normal files within the file system, the boot files have no defect management support. This combined with single and double burst ECC correction in ROM, allow us to have only two copies of each normal file. The boot loader loads this list prior to reading up any normal file. As currently implemented, the lack of this list is not a fatal condition as the retry algorithm and ECC correction is sufficient to read the normal files.

It is intended that UPT assay the system cylinders and initialize this defect list prior to the diskware download of the files in the file system.

9.2.2 Normal Files

The majority of files in the file system are "normal" files that are accessed via the main directory loaded by the boot loader. The normal files differ in that they don't have to be fixed (diskware changes may well cause the files to move around), and they have the benefit of defect management. An additional distinction could exist if the number of copies of the normal files were to be changed. All normal files have the same file header as required of any diskware element (see the Quantum diskware download spec), consisting of both a ROM and diskware header. These headers provide for the same level of verification that exists during diskware download. This verification includes valid checksum, correct product code, matching ROM checksum, and a matching ROM version stamp.

9.2.2.1 Directory

The directory of normal files is described in the previous section (Boot Files) as it is actually a boot file that must be read up prior to loading any normal file. The directory is written during diskware download time and is static from that point on. Specifically, this means that adding a file to the file system requires downloading a new directory and the associated file (generally this would also involve a new diskware build), there is no facility for firmware to modify the directory on the fly.

9.2.2.2 File ID's

Contained within the firmware is an enumerated type (C language syntax) list of file identifiers. This list is based on the Quantum diskware download specification, which assigned many of the file identifiers. The firmware interface to the file system exclusively uses file ids to uniquely specify which file to operate on. Please refer to the "Read Layout of Negative Cylinders" command in the Unified Firmware Manual for a complete list of file identifiers.

9.2.3 File System Firmware

The firmware that implements the file system is actually spread across four different parts of the firmware. The low level read routines are embedded in ROM and are used by all parts of the file system firmware (there is only one read function). The next level is the temporary boot loader code that replicates parts of the diskware file system routines as necessary until diskware is loaded. The file system API calls invoked by firmware exist with the resident firmware module. The last section is the file system superset command, which will ultimately reside in an overlay.

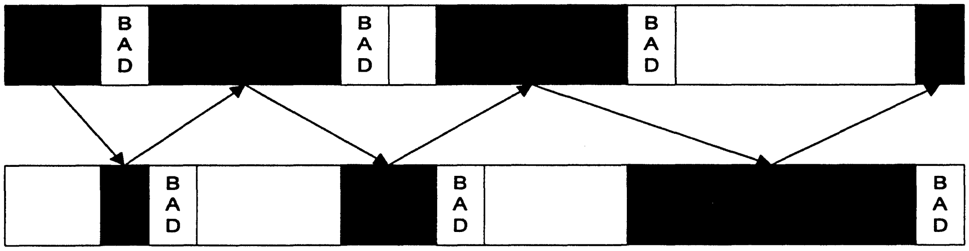
Irregardless of file type or where the call originated, all file read request funnel through a single read routine that implements a version of the "Firmware Redundancy Algorithm" to nearly guarantee that a file can be read. The differences in file types are masked by a firmware data structure (named sFile) that the lower level file system routines use during file access.

9.2.3.1 Firmware Redundancy Algorithm for Reads

The firmware redundancy algorithm is an attempt to exhaustively use all copies of a file as necessary to guarantee a successful file read. The algorithm's success depends on errors being distributed at different places among the different copies of a file. The single failure mode occurs when the same sector is damaged in each copy of a file. Given the availability of ECC correction, defect management in the system cylinders, and redundant copies of files, this type of failure should be rare. The following "C" like pseudo code depicts the nested retry loops that implement the firmware redundancy algorithm:

```
for ( each copy of a file )
{   // start from the first sector of each file as necessary
    for (each copy of a file )
    {   // start with the first unread sector
        for ( number of read retries )
        {   // retry read on same copy for channel training
            read from first unread sector to end of file;
            if (read was successful) break;
        }
        if (read was successful) break;
        else
        {
            move to next file copy at same sector offset;
        }
    }
    if (read was successful)
    {
        validate file;
        if (validation successful) return(NO_ERROR);
    }
    start over with next copy of file;
}
```

The preceding algorithm can be further illustrated by the following example which depicts the different "chunks" of two copies of a file that were knitted together for a successful read. The dark areas represent valid sectors while the light areas indicate regions that start with an invalid sector (but may have valid sectors following).



9.2.3.2 File Write Algorithm

The algorithm used for writing files differs somewhat from the read algorithm in that an entire copy of a file is written and read back. The other difference is that not all copies need to be successfully written depending on the criteria for long term reliability of the drive. As part of the write retry loop we also delay a decaying amount of time in an attempt to circumvent vendors who attempt to diskware a drive while on a variable frequency "paint shaker" as part of their testing.

for (each copy of a file)

```
{
    for ( number of write retries )
    {
        write a single copy of the file;
        if ( write was successful )
        {
            for ( number of read retries )
            {
              read entire copy of the file;
              if ( read successful ) break;
            }
        }
        if (read successful) break;

        // Either the initial write failed or the read back
failed:
        // wait for a little while (for the paint shaker to
switch
        // to a benign frequency ?), then try again.
        Delay4Usec( attempts );
    }
    if (read successful)
    {
        increment copiesWritten;
    }
}
```

After completion a test is made to determine if the number of copies written exceeds the desired threshold. This is actually a far more stringent test than the read algorithm requires. The read algorithm has the luxury of reading up pieces from various copies of a file, while the read back after a write either reads a single copy up or not. Thus the actual loading up of files during power up could actually make it even if there was no single successfully written file copy.

9.2.3.3 Current File Structure

A common data structure is used to mask differences between the boot and normal files. This structure is filled in with file data whenever a successful search for a file id is conducted. The parts of the structure that assist in masking the file differences are the CHS structure and the variables: copy and maxCopy. During the call to the FindFile routine the CHS structure is filled in with the starting address of the desired copy of the file. The maxCopy field is also filled in with the desired number of copies, while the mfgCopiesRequired field contains the number of valid copies for a passing drive. The remaining routines use the copy and maxCopy values to determine which copy to work on and how many total copies there are. Finally the offset value is used to offset the starting sector value when we are attempting to read in pieces of a file from the different copies.

The uFlags field is a 2-bit bitfield structure that currently indicates whether or not the directory and the system defect list have been successfully read. The sub structure fileSysVars buried within the parent structure sFile is only enabled during firmware development and serves as a place to collect error information for the current file operation. The first two fields are of particular interest as they indicate how difficult it was to read in the current file. The chunks field indicates how many separate pieces from among the various copies had to be knitted together for a successful read. The retries field is a count of the total number of read attempts required to read in all the chunks. The rest of the fields are error return codes from various related structures and the next nested sub structure sFirstDiskErr is a snapshot of the values that existed when the first disk error occurred. The sFile structure definition follows:

```
uword        loadAddress;         // buffer address for
                                  // read/write file
sdaChs       fileChs;             // CHS value for current
                                  // copy of file
errorIdent   eErrorCode;          // Firmware error code for
                                  // last file operation
byte         copy;                // current copy number
byte         mfgCopiesRequired;   // number of valid copies for
                                  // a passing drive
byte         maxCopy;             // maximum number of file copies
byte         length;              // length of file in sectors
byte         offset;              // sector offset to add to chs value
fileIdent    eFileId;             // enumerated file id
union                             // status of various parts of
                                  // file system
{
    byte all8;
    struct
    {
              byte    directoryOK    : 1;    // normal directory
                                             // has been loaded
              byte    sdlOK          : 1;    // system defect list
                                             // has been loaded
    } bit;
} uFlags;

// debug mode only structure for logging file system errors.
struct fileSysVars
{
       hword        chunks;       // pieces the had to be read
                                  // to get the whole file
       hword        retries;      // the number of retries performed
                                  // on the current file
       errorIdent   eDiskError;   // SDisk.errorCode copy
       errorIdent   eValidate;    // diskware element verify routine
                                  // return code

       struct                     // snap shot of the first reported
                                  // disk error
       {
           errorIdent          eDiskError;    // SDisk.errorCode copy
           byte                copy;          // file copy number
           hword               retries;       // number of retries performed
                                              // to date
           chs                 sLocn;         // CHS of last disk error
           union diskCmdType   uOperation;    // disk op when the first disk
                                              // error occurred
       } sFirstDiskErr;
} sDebug;
```

9.2.3.4 API Calls

Since it is intended that the file system be accessed only through a few function calls, it seem appropriate to describe those functions.

ROMBootLoader — This function is called by the power up task to initiate the first phase of diskware loading. It will load up a copy of the boot block loader if possible, validate the copy, and execute the function pointed to by the vector in the loaded diskware header.

RedundantReadFile This routine will attempt to read up either a boot file or a normal file from the file system. Prior to calling this routine the desired file information must be set in the sFile structure (typically done by the FindFile routine). The Firmware Redundancy Algorithm is used to exhaustively use all file copies as necessary to insure a successful load.

PrepNextFileCopy When it's necessary to access the next copy of a file, this function updates the sFile data structure to reflect the change. The passed parameter is a sector offset that is added to the base sector value for the copy. This offset allows us to start from the failed point in the next copy of a file.

PrepFileCopy This function copies the data contained in the sFile structure to both the sDisk and sSystemCSeg structures in preparation for a disk request.

BootBlockLoader This is the intermediate loader that is read up by the ROM based boot loader. This loader is in memory only temporarily and is responsible for reading up diskware, the defect lists, the primary overlay, and the vector table. This routine can only use ROM based or local functions since diskware is not live when it runs. The final step in the load process is to invoke the resident init function from within the loaded diskware module.

FindFile Scans the directory looking for a match on the passed fileId and if it is found the file details are moved to the global sFile data structure in anticipation of a file read/write operation. ReadFile and WriteFile call this function prior to the actual read/write operation.

GenerateFileHeader Build a valid file header at the passed address. The resident header is used as a template to get the correct ROM versions and checksum.

ReadFile Read up a normal file (or one of the three boot files from the system data area into memory. File is validated using the checksum in the diskware header block at the beginning of each file.

WriteFile Write normal file (or one of the three boot files) from memory to the system data area. File checksum is computed and saved in the diskware header block at the beginning of each file.

FileSize Scan the directory looking for the passed file id and return the size of the file in sectors if found.