
D O M A I N

Order No. 009492
Revision 00

# *Making the Transition to SR9.5*

# Making the Transition to SR9.5

Order No. 009492
Revision 00

Apollo Computer Inc.
330 Billerica Road
Chelmsford, MA 01824

First Printing: January, 1987
Latest Printing: January, 1987

This document was produced using the Interleaf Workstation Publishing Software (WPS). Interleaf and WPS are trademarks of Interleaf, Inc.

# Preface

*Making the Transition to SR9.5* describes several major changes at Software Release 9.5 that could introduce some inconsistencies with earlier software versions. It describes what features work differently than before, and how to adjust your programs accordingly.

We've organized this manual as follows:

| | |
|---|---|
| **Section 1** | Provides reasons for upgrading to SR9.5 to take advantage of the performance enhancements provided with this release. It tells which programs you *must* recompile with SR9.5 compilers, and which programs you are not required to recompile. |
| **Section 2** | Describes how to rebuild your programs using SR9.5 compilers. It includes a list of possibly incompatible features to check before recompiling. Also, it shows how to maintain pre–SR9.5 programs with updated SR9.2 tools. |
| **Section 3** | Describes changes made to the DOMAIN C compiler that might affect programs developed on earlier versions. |
| **Section 4** | Describes changes made to the DOMAIN FORTRAN compiler that might affect programs developed on earlier versions. |
| **Section 5** | Describes changes made to the DOMAIN Pascal compiler that might affect programs developed on earlier versions. |
| **Section 6** | Describes the new optimizing enhancements made with this realease, and how they could affect C, FORTRAN, or Pascal programs developed on earlier versions. |

# Related Manuals

Most of the information in this document deals primarily with updating to SR9.5. For general information on the features added with SR9.5, see the following manuals:

- *DOMAIN C Language Reference* (002093-03).
- *DOMAIN Pascal Language Reference* (000792-04).
- *DOMAIN FORTRAN Language Reference* (000530-05).
- *DOMAIN Language Level Debugger Reference* (001525-04).
- *DOMAIN Assembler Reference* (008862-01).
- *Programming With General System Calls* (005506-00).

# Problems, Questions, and Suggestions

We appreciate comments from the people who use our system. In order to make it easy for you to communicate with us, we provide the User Change Request (UCR) system for software-related comments, and the Reader's Response form for documentation comments. By using these formal channels you make it easy for us to respond to your comments.

You can get more information about how to submit a UCR by consulting the *DOMAIN System Command Reference*. Refer to the **crucr** (CREATE_USER_CHANGE_REQUEST) Shell command description. You can view the same description on-line by typing:

```
$ help crucr
```

For your documentation comments, we've included a Reader's Response form at the back of each manual.

# Documentation Conventions

Unless otherwise noted in the text, this manual uses the following symbolic conventions.

| | |
|---|---|
| **boldface** | Bold, lowercase words or characters in text represent either keywords or commands that you must use literally. |
| `typewriter` | Typewriter font words in command examples represent values that you must supply. We also use typewriter font for program examples. |
| CTRL/Z | The notation CTRL/ followed by the name of a key indicates a control character sequence. You should hold down <CTRL> while typing the character. |
| •<br>•<br>• | Vertical ellipsis points mean that irrelevant parts of a figure or example have been omitted. |

# Contents

# Tables



# Examples

# Making the Transition to SR9.5

This software release (SR9.5) is a major release of the DOMAIN operating system. Its primary goal is to improve overall system performance. It is especially concerned with improving the performance of C programs. To achieve these performance goals, however, it was necessary to change some internal data representations and runtime conventions. Some of these changes are internally inconsistent with earlier software versions. However, we attempted to minimize incompatibilities as much as possible.

Generally, you can upgrade your node to SR9.5 without any problems: *Programs developed on earlier versions will most likely run on SR9.5.* However, to take full advantage of the performance gains offered in this release, you will want to recompile existing programs with SR9.5 compilers. In doing so, you could find some differences with SR9.5 that will introduce some inconsistencies. This document is intended to describe the problems you might encounter when recompiling your programs with SR9.5 compilers.

This document contains:

- Advantages of rebuilding under SR9.5
- How to rebuild your programs under SR9.5
- Changes to C
- Changes to FORTRAN
- Changes to Pascal
- Optimization enhancements affecting generated code

# 1. Why You Should Rebuild with SR9.5 Tools

This section describes the advantages of rebuilding programs with SR9.5 and when you *must* rebuild programs with SR9.5.

## 1.1. Advantages of Rebuilding Existing Programs with SR9.5

The SR9.5 compilers generate more efficient object modules that run faster than modules generated by older compilers. On certain compute-bound programs we have seen performance gains of between 10 and 20 percent. Because your individual applications differ, your performance gains may vary, but the vast majority of programs should show considerable performance improvement.

## 1.2. When You Must Rebuild with SR9.5

Usually, you will want to recompile existing program modules with SR9.5 compilers to take advantage of the performance gains with this release. However, there are certain circumstances under which you *must* recompile programs with SR9.5 compilers. Table 1 lists these circumstances. Section 2, "How to Rebuild Your Programs under SR9.5," explains how to go about recompiling your programs with SR9.5 compilers.

**Table 1. What to Recompile with SR9.5 Compilers**

| What You Must Recompile: | Why You Must Recompile: |
|---|---|
| Any program module that you plan to bind with an SR9.5 object module | The DOMAIN Binder will not allow you to bind an SR9.5 object module with a pre-SR9.5 object module because runtime conventions and the object module format have changed at this release. The Binder prevents you from mixing object module formats to avoid any unexpected results. |
| Any user-installed library that will be called from SR9.5 modules | If you provide your own global or private libraries, you will depend on a new procedure pointer format. This means you must recompile the library *before* recompiling any programs that reference the library. During the transition, you can adjust your libraries to recognize both pre-SR9.5 and SR9.5 runtime conventions. |
| All type managers | If you developed a type manager using the Open Systems Toolkit as described in *"Using the Open Systems Toolkit to Extend the Streams Facility,"* (008863), you must recompile the manager with an SR9.5 compiler. Your manager depends on DOMAIN-supplied global libraries in SR9.5 format. These particular global libraries do *not* recognize pre-SR9.5 runtime conventions. |

Table 2 lists the types of program modules that will run as expected without having to recompile with SR9.5 compilers. (However, to benefit from SR9.5 performance enhancements, you might want to recompile these program modules.)

**Table 2. What Does Not Need to be Recompiled with SR9.5 Compilers**

| What You Do Not Have to Recompile: | Why Not: |
|---|---|
| Any module that calls DOMAIN global libraries | Global libraries recognize pre-SR9.5 conventions and make the necessary conversions to SR9.5 conventions. |
| Program modules relying on SR9.5 versions of DOMAIN layered products | Most layered products are backward compatible, so any programs developed with pre-SR9.5 compilers will run under SR9.5 standard software. See the Release Notes for the individual layered products for full information on compatibility issues. |

# 2. How to Rebuild Your Programs under SR9.5

SR9.5 comes with two versions of the debugger and each compiler. The primary versions are for developing programs using the performance enhancements provided with SR9.5. The secondary versions are for maintaining programs developed on earlier versions of DOMAIN compilers. The maintenance versions have the suffix, "_sr9.2." The following sections describe how to:

- Upgrade programs developed on earlier versions by recompiling with SR9.5.
- Maintain programs developed on earlier versions *without* recompiling with SR9.5.

## 2.1. Rebuilding Your Pre-SR9.5 Programs under SR9.5

In most cases, you'll want to upgrade your existing software programs to take advantage of the performance improvements of SR9.5. This section highlights changes to keep in mind when you recompile. It also describes how to recompile and bind your existing programs with SR9.5.

### 2.1.1. Before Recompiling under SR9.5

Before recompiling existing programs with SR9.5 compilers, you should know what might work differently from in the past. Most programs will compile as you expect. However, you might receive new warning or error messages if you don't make a few changes before you recompile. Table 4 highlights the changes you might need to make. The remainder of this document provides details about these changes.

Note that the new warning messages you might receive when you recompile occur largely because the compilers analyze programs more rigorously when optimizing them. These warning messages often reflect possible errors or improper use of programming constructs. So, it's important to check your program when these warnings occur.

**Table 3. SR9.5 Compiler Changes that Affect Your Programs**

| Change | Description |
|---|---|
| C **enum** Variables | DOMAIN C now has **short enum** and **long enum** data types, and the *default* **enum** is now 32 bits rather than 16 bits. This could make data files incompatible if they are shared by SR9.5 and pre-SR9.5 programs. |
| C Arrays as Formal Parameters | DOMAIN C fixes a bug in formal parameters declared as arrays. This new way is consistent with Portable C Compilers (PCC). DOMAIN C now treats all array formal parameters as pointers. This affects taking addresses of formal parameters, and using them as arguments to **std_$call**. |
| FORTRAN **nil** Pointer | FORTRAN programmers who used IAADR(0) to pass a **nil** pointer to a FORTRAN function or subroutine *must* replace it with 0. |
| Pascal Subrange of **char** | In DOMAIN Pascal, a subrange of the **char** data type is now 8 bits long rather than 16 bits. This could make data files incompatible if they are shared by SR9.5 and pre-SR9.5 programs. |
| Pascal **in out** Parameter | DOMAIN Pascal changes the interpretation of the **in out** parameter. It now means that you *must* pass a value to this parameter and that the called routine can (but is not required to) assign a value to it. Prior to this release, **in out** was treated the same as **var** in that you could, but were not required to, pass a value to the parameter. |
| Pascal **goto** Statement | Pascal does *not* allow you to use a **goto** statement to transfer control into the middle of the following statements: **for, while, case, with**, or **repeat**. You can still use them within **then** and **else** clauses, and from one **case-action** to another **case-action** within the *same* **case** statement. |
| Pascal Subrange of Integers | Pascal now performs unsigned comparisons for all unsigned variables. Prior to this release, PASCAL performed *signed* comparisons for any subrange of integers that required more than 8 bits, but fewer than 16 bits. |
| Pascal Runtime System | A number of errors in the Pascal runtime system have been corrected. If you improperly depended on these defects in previous releases of Pascal, you may find that your program no longer executes. |
| -opt Compiler Option | DOMAIN compilers now allow you to specify increasing degrees of optimization from 0 to 4. Prior to this release, you could only turn optimization on and off. For C and Pascal, the default -opt performs more optimizations so that, in some cases, your program might produce warnings that it didn't produce before. |
| Dead Store Optimization | DOMAIN compilers now perform an optimization, called the **dead store**, which eliminates assignments to variables whose values are subsequently never used. When this occurs, the compiler generates a warning that is a good indication that *your program is not doing what you intended.* |

**Table 3. SR9.5 Compiler Changes that Affect Your Programs (Concluded)**

| Change | Description |
|---|---|
| Procedure Pointers | Procedure pointers now point to the address of the routine, rather than to a data structure containing the ECB address. This affects you if you access the Known Global Table (KGT), install your own global libraries, and write type managers in C using the Open Systems Toolkit.<br><br>If you accessed the KGT prior to this release, the KGT returned a pointer to the ECB, and you were required to construct a pre-SR9.5 procedure pointer manually. Now the KGT returns a SR9.5 procedure pointer, so you do not need to build a procedure pointer before assigning a value to it. You can assign a pointer value to the variable directly.<br><br>If you use the Open Systems Toolkit to write your own type managers in C, you have a similar situation. Prior to this release, you had to construct a **nil** procedure pointer when you wanted to initialize a procedure pointer to **nil**. (You did so by defining it as an array of two long integers, each of which contained the value of 0.) Now, you can simply pass a value of 0. |

### 2.1.2. Compiling and Debugging Your Programs under SR9.5

To take advantage of the performance enhancements of SR9.5, you must rebuild the programs with SR9.5 compilers.

You must use the SR9.5 version of **debug** to debug programs compiled with the standard SR9.5 compilers. If you attempt to debug such a program with any earlier version of the debugger (including **debug_sr9.2**) you will receive an error message to that effect. At SR9.5, **debug** contains many enhancements that are documented in the *DOMAIN Language Level Debugger Reference* (001525).

### 2.1.3. Binding Your Programs under SR9.5

At SR9.5, there is one version of the DOMAIN Binder (unlike DOMAIN compilers and debuggers, which have both SR9.5 and pre-SR9.5-maintenance versions). Use the binder to bind *either* SR9.5 or pre-SR9.5 object modules. You will get an error if you attempt to bind object modules that are produced by an SR9.5 *and* a pre-SR9.5 compiler because the object module formats are different.

## 2.2. Maintaining Your Pre-SR9.5 Programs with the 9.2 Versions

You may not want to recompile existing program modules with the performance-enhanced SR9.5 compilers. This release provides you with updated compilers and a debugger that allow you to continue compiling programs with the pre-SR9.5 object module format and runtime conventions.

### 2.2.1. Compiling Pre-SR9.5 Programs with the Maintenance Compiler

To maintain earlier programs, we suggest that you label the pre-SR9.5 programs and the directories containing them with a descriptive extension (such as _sr9.2) so that you can easily distinguish the programs that depend on the earlier format. You can recompile programs using SR9.2 compilers by specifying the compiler labeled with the SR9.2 extension.

For example, to compile a C program using the SR9.2 version of the compiler, specify the following:

```
$ cc_sr9.2 your_sr9.2_program
```

These maintenance compilers are similar to those released at SR9.2 except that they include new bug fixes and syntax enhancements for SR9.5 constructs. For details, refer to the documentation accompanying the appropriate language compiler.

### 2.2.2. Debugging Pre-SR9.5 Programs with the Maintenance Debugger

You *must* use the maintenance debugger (**debug_sr9.2**) supplied with this release for pre-SR9.5 programs. That is, pre-SR9.5 debuggers *will not run* under SR9.5. If you attempt to debug pre-SR9.5 programs with an earlier debugger, the program crashes without an error message. (The **debug_sr9.2** debugger is an SR9.5 debugger *for* pre-SR9.5 programs.) This debugger contains many enhancements that are documented in the *DOMAIN Language Level Debugger Reference* (001525).

To debug the program using the debugger that recognizes the SR9.2 format, specify the following:

```
$ debug_sr9.2 your_sr9.2_program
```

# 3. Changes to C

SR9.5 contains a few new features to DOMAIN C. Most features are compatible with earlier versions, and they are described in detail in the C language documentation. The following sections describe only the changes made in DOMAIN C at SR9.5 that affect programs developed on earlier versions. Later sections in this document describe changes that affect all DOMAIN compilers. The following sections describe:

- An extension to DOMAIN C's enumerated type
- A bug fix to DOMAIN C's formal array parameters
- Usage of DOMAIN C's **register** storage class specifier

## 3.1. Changes to Enumerated Types in C

Prior to this release, DOMAIN C supported one enumerated type, declared with the keyword **enum**, which was 16 bits long. With this release, you can declare an **enum** variable to be either 16 bits (**short enum**) or 32 bits (**long enum**). When you specify the **enum** keyword without a size, the default size is 32 bits. This change was made to be consistent with other C compilers.

The change of the default enum size presents a problem for data structures or data files that depend on the internal storage representation. Take, for example, two programs: one compiled with a 9.5 compiler and the other compiled with an earlier compiler. If both programs define a structure containing an enumerated value, and use the same data file, the data file will be misused because the structure has different storage requirements. To correct this, you can do either of the following:

- Recompile the pre-9.5 program on a SR9.5 compiler. Do this if the data file is temporary, because its format will change.
- Change the **enum** declaration in your 9.5 program to be a **short enum** to be consistent with programs compiled with earlier versions. Do this if you do not expect to recompile the earlier source program, or if the format of the data file must be preserved.

Note that many DOMAIN system calls depend on Pascal enumerated types (which remain 16 bits long). Consequently, we changed any **enum** declarations in the C language insert files to be **short enum**. Therefore, any program that depends on a DOMAIN-supplied insert file will compile properly. However, if your program makes procedure calls to any DOMAIN system manager but does *not* use the supplied */sys/ins/xxx.ins.c* insert file, you must change the **enum** declarations in your program to be **short enum** to be compatible with the 16-bit Pascal enumerated type size.

Previous documentation states that DOMAIN C enforces strict separation between enumerated types and integer values. The current implementation, in keeping with most C compilers, will not generate errors if you assign integer values to enumerated types. However, it is good programming practice to treat enumerated type values as distinct variable types. Try to limit the operations on enumerated types to the following:

- Assigning an enumerated value to an enumerated variable
- Comparing an enumerated value to another enumerated value
- Using an enumerated value as a subscript to an array

## 3.2. Changes to Formal Array Parameters

In this release, DOMAIN C corrects the way it handles arrays as formal parameters. It now considers all arrays that are formal parameters to be *pointers* rather than arrays. Since the formal array is actually a pointer, if you take the address of the array, you will get a pointer to a pointer.

Note that this case occurs only when you are declaring an array as a *formal parameter*. If the array is declared as a global or local variable, DOMAIN C treats the array as you would expect.

Prior to this release, DOMAIN C considered any attempt to take an address of a formal array parameter to be redundant, and therefore, quietly ignored it. With this release, the compiler takes the address of a formal array parameter, because it converts any formal array parameter type to be a pointer type.

Example 1 shows how the C compiler treats formal array parameters *before* and *after* this bug fix:

**BEFORE**

```
foo(ar)

char ar[];

{
    bar(&ar); /* Compiler ignored the ampersand because it considered
               * that it was an address of an array. Function "bar"
               * must expect an argument of type (char*).            */
}
```

**AFTER**

```
foo(ar)

char ar[];

{
    bar(&ar);/* Compiler takes the address of "ar" because it now treats
              * a formal char array as a pointer to char. Function "bar"
              * must expect an argument of type (char**).              */
}
```

*Example 1. Change to Declaring Arrays as Formal Parameters*

This change is of particular interest if you use the standard calling convention (**std_$call**), because **std_$call** implicitly takes the address of each parameter when passing it. If you pass an array using **std_$call**, you must de-reference the array to avoid making a pointer-to-pointer reference.

Example 2 shows how you should pass an array as a formal parameter to a **std_$call** function with this release. The main procedure invokes a function that calls **ios_$create** (using **std_$call**) to create a file for write access. Since it passes the filename (which DOMAIN defines as a character string array) as a formal parameter, the name is already considered to be a pointer, so you must de-reference the name in the call to avoid a pointer-to-pointer reference.

```
#module create_file

/* Function create_name calls ios_$create to open a file. It returns the
   stream id to the main procedure.  */

ios_$id_t create_name(name)          /* Name is a formal parameter. */
    name_$pname_t  name;             /* name_$pname is a char array.*/
{
    ios_$id_t      stream;

/* Create an ASCII file and open it for write access. */
/* Notice that we dereference the array variable "name" because it was
 * declared as a formal array parameter, and is now regarded as a pointer
 * type.  If we didn't, we would have a pointer-to-pointer reference, which
 * is not what we want. */

    ios_$create (*name,                  /* Dereference array !!! */
                 (short)strlen(name)     /* Length of name. */
                 uasc_$uid,              /* ASCII uid */
                 ios_$write_opt,         /* Open options */
                 stream,
                 status);

/* Close file. */
    ios_$close (stream, status );

/* Pass value of stream to main */
return (stream);
} /* end create_name */

main()
{

    ios_$id_t  fid;
    status_$t  status;

/* Call create_name function to create the file. */

    fid = create_name ("myfile");

}  /* main */
```

*Example 2. Declaring Arrays as Formal Parameters to* **std_$call**

## 3.3. Register Storage Class Specifier

Prior to this release, the **register** storage class specifier (which you supply with a C variable declaration to suggest that the compiler store the variable in a register) was ignored. When performing optimizations, the compiler would treat the register variable like an automatic (**auto**) variable. This meant that automatic variables were just as likely to be placed in registers as register variables.

With this release, the compiler more frequently stores register variables in registers. This is because the compiler gives them a higher priority over automatic variables when determining which variables are allocated to registers. Still, the compiler does not *guarantee* that all register variables will actually be stored in registers; however, it's most likely.

# 4. Changes to FORTRAN

SR9.5 introduces only one change to DOMAIN FORTRAN that could affect existing programs: FORTRAN programmers who used IADDR(0) to pass a NIL procedure pointer to a FORTRAN function or subroutine *must* replace it with 0.

For details on FORTRAN enhancements at SR9.5, see the FORTRAN documentation accompanying this release. Later sections in this document describe changes that affect all DOMAIN compilers.

# 5. Changes to Pascal

This release contains a few features added to DOMAIN Pascal. Most features are compatible with earlier versions, and they are described in the Pascal documentation accompanying this release. The following sections describe some Pascal features that might produce unexpected results in your current programs due to the new data representations and runtime conventions. Later sections in this document describe changes that affect all DOMAIN compilers. The following sections describe changes to:

- Storage requirements for the subrange of **char** data type
- The **in out** parameter extension
- Use of the **goto** statement
- Subrange of integers
- Errors in the runtime system

## 5.1. Changes to Subrange of char in Pascal

SR9.5 changes the storage allocation for the subrange of **char** data type in a packed record. At SR9.5, the subrange of **char** requires eight bits and is byte-aligned. (Prior to this release the subrange of **char** incorrectly required 16 bits.) Since this could create incompatibilities with earlier compiler revisions, the compiler will issue a warning message *for this revision only.*

Pascal's subrange data type allows you to specify a variable's valid range of values. This range of values is a subset of another data type called the base type. For example, a subrange of a **char** base type is:

```
VAR { Declaring a variable as a subrange of CHAR.}
    capital_letters  : 'A' .. 'Z';
```

Usually, the storage allocation for subrange variables is the same as its base type, except when the subrange is a field in a packed record. Prior to this release, a field in a packed record with subrange of **char** data type incorrectly required 16 bits and was word-aligned. All other subrange fields in a packed record took up the number of bits required for their extreme values and were bit-aligned. At SR9.5, subrange of **char** in a packed record requires eight bits and is byte-aligned.

Due to this change in storage allocation, data files that are shared by programs compiled with different versions -- some compiled with an SR9.5 compiler, others compiled with a pre-SR9.5 compiler -- will be misused. For example, if you declare a field in a record to be a subrange of **char** type and you manipulate the same data file using programs compiled with different compiler versions, some data gets destroyed because the storage requirements differ.

To correct this, you can do either of the following:

- Recompile the pre-9.5 program on an SR9.5 compiler. Since the format of the data file will change, do this if you don't need to preserve the format of the file.
- Adjust the subrange of **char** declaration in your 9.5 program to be consistent with programs compiled with earlier versions. To do so, pad the record by inserting an 8-bit field *before* the subrange field. Do this to preserve the format of a data file, or if you do not expect to recompile the earlier source program.

## 5.2. Pascal's in, out, and in out Parameter Extensions

DOMAIN Pascal supports **in**, **out**, and **in out** parameter extensions to allow you to specify the *direction* of parameter passing. Previous versions considered that the **in out** parameter extension was synonymous with the **var** parameter type. However, the SR9.5 Pascal compiler makes a distinction between **in out** and **var** parameter types when optimizing code. Consequently, when you recompile programs with this version, you might receive a warning message that you never received before.

The warning message states that the compiler is eliminating an assignment statement if you never use the new value after the assignment. This is a result of a compiler optimization, called the **dead store elimination**, which we describe in detail in Section 6.2., "Eliminating Unused Assignment Statements." In this section, we describe how DOMAIN Pascal interprets parameter types when optimizing code.

The reason you get this warning is that DOMAIN Pascal performs certain optimizations depending on the way you declare parameters. By specifying a parameter direction when declaring a parameter, you *promise* to use the parameter values in a specific way. The compiler uses the information from your parameter declaration to determine how to optimize code.

DOMAIN Pascal refers to the following rules regarding the parameter extensions. A **parameter** is the name declared in the formal routine declaration. An **argument** is the actual expression or variable that the caller passes when making the subsequent procedure or function call. In the following list, only the interpretation of the **in out** parameter changes with this release.

| | |
|---|---|
| **in** | Tells the compiler that you are *passing* a value to this parameter and that the called routine *cannot change* the value. If the routine tries to change its value (by using it on the left side of an assignment statement) you will usually get the error, "Assignment to **in** argument." |
| **out** | Tells the compiler that you are *not passing* a value to this parameter, but the called routine *will assign* a value to it. If the routine does not assign a value to the parameter before returning, you might get the warning, "Variable not initialized before this use." |
| **in out** | Tells the compiler that you are *passing* a value to this parameter and that the called routine *might (but is not required to) assign* a value to it. You must initialize a variable before passing it as an argument to the routine. |
| **var** | Tells the compiler that you *might pass* a value to the parameter. The compiler cannot count on any initial or assignment values so it cannot perform any dead store eliminations. |
| **(blank)** | Tells the compiler to make a *private* copy of the formal parameter. The called routine can change the copy, and it does not affect the value of the formal parameter. (This is often referred to as **passing by value**.) |

The following example shows how you can get unexpected results if you don't use the arguments as specified in the routine declaration. In this example, *plist* is declared as an **out** parameter, which indicates that you're passing in an uninitialized variable, and you expect to assign a value to it within the procedure.

```
PROCEDURE list_names (OUT: plist: listptr);
•
•
•
    if ( condition ) then
        plist := 1;
```

If you call the above procedure with the following statements

```
lp := NIL;
list_names (lp);
```

Pascal matches the actual argument *lp* to the formal parameter, *plist*. Since *plist* is declared as an **out** parameter, it assumes that *lp* will be assigned a value within the procedure. Thus, the assignment statement *(lp := NIL)* is redundant, so Pascal eliminates it. However, the procedure assigns a value to *lp* only if the condition is true. Since *lp* might not get a value, you receive a warning.

To correct this, declare *plist* to be a **var** parameter rather than an **out** parameter. This way, Pascal executes the assignment statement preceding the procedure because the procedure expects an initial value. In addition, by specifying **var** rather than **out**, Pascal cannot assume that the procedure will always assign a value to *lp*.

## 5.3. Changes to Pascal's goto Statement

At this release, Pascal detects more invalid uses of the **goto** statement. DOMAIN Pascal now complies with the ISO/ANSI standard in that it does not allow you to use a **goto** statement to transfer control into the *middle* of a structured statement such as the **for, while, case, with,** or **repeat** statements. If you try to transfer control into one of these statements, you will get the error: "GOTO transfers control to a structured statement outside of its scope (cleanup)."

Note that DOMAIN Pascal continues to permit you to use the **goto** statement to transfer control:

- Into a **then** or **else** clause of an **if-then-else** statement from inside *or* outside the statement.
- From one **case-action** to another **case-action** in the *same* **case** statement.

## 5.4. Change to Pascal's Treatment of Subrange of Integers

Prior to this release, PASCAL performed *signed* comparisons for any subrange of integers that required more than 8 bits, but fewer than 16 bits. However, if the unsigned integer required the entire 16 bits, Pascal performed an *unsigned* comparison.

For example, if a program declared a 15-bit unsigned integer as follows

```
VAR
    posint : 0..32767;    { 15-bit unsigned integer, stored in a WORD }
```

Pascal would perform the following code fragment as a signed conditional branch.

```
    IF posint > 10 THEN ...
```

Normally, this wouldn't matter, because the high bit should be zero. However, in one case, the previous Pascal compiler generated a signed comparison and did *not* execute the THEN clause.

```
posint := 65535;    { Out-of-range assignment  }
...                 { is not defined in PASCAL }
IF posint > 10 THEN....
```

With this release, Pascal is more consistent by performing unsigned comparisons for *all* unsigned variables. That is, referring to the above example, Pascal evaluates the IF clause to TRUE (65535 is greater than 10), and executes the THEN-clause.

Note that Pascal always performs unsigned comparisons on unsigned subrange variables requiring a full 16 bits (for example, *TYPE pinteger= 0..65535*). This change was added so that unsigned variables requiring less than 16 bits are treated the same as unsigned variables requiring the entire 16 bits.

As a result of this change, programs that attempt to check ranges of an unsigned variable will receive a warning message. For example, in the following program fragment

```
IF posint < 0 OR ELSE posint > 32767
THEN
    { Out-of-range value! }
```

Pascal generates a warning message, indicating that the expression *posint < 0* was known to be FALSE at compiletime and has been constant-folded. You can avoid these warnings (and simplify your program) by using the **in_range** Pascal function. That is, you can replace the above program fragment with:

```
IF NOT IN_RANGE(posint)
THEN
    { Out of range value! }
```

The **in_range** function generates efficient code to determine whether any scalar variable is within its defined range.

## 5.5. Changes to Pascal's Runtime System

As part of the work to comply with the ISO/ANSI Pascal standard, a number of errors in the Pascal runtime system have been corrected. If you improperly depended on these defects in previous releases of Pascal, you may find that your program no longer executes.

For example, the Pascal standard says that it is illegal to call **eoln** when **eof** is true. Previous releases of Pascal did not enforce this restriction. SR9.5 Pascal correctly issues an error message at runtime and terminates the erroneous program.

# 6. Optimizing Enhancements Affecting Generated Code

The following sections describe the new optimizing enhancements, and how they could affect your C, FORTRAN, and Pascal programs. They describe:

- A change to the compiler -opt option that allows you to specify varying degrees of optimization.
- The new dead store optimization that might produce a new warning message.
- The **volatile** feature, which prevents the compiler from making unwanted optimizations.

## 6.1. Changes to the -opt Compiler Option

At SR9.5, the -opt compiler option allows you to specify increasing levels of optimization. In prior releases, you could either generate optimized code (by default, all DOMAIN compilers optimize programs) or you could suppress optimization with the -nopt compiler option. Now you can specify the degree of optimization you want: from no optimization (-opt 0), to some optimization (-opt 1, 2 or 3) to the greatest amount of optimization (-opt 4). The -opt 0 level is equivalent to the -nopt compiler option.

For details on the specific optimizations that occur at each optimization level, refer to the documentation accompanying the appropriate language compiler.

By specifying various levels of optimization, users have greater control over how the compiler generates code. In general, as the optimization level increases, the compiletime increases. Consequently, you may want the compiler to perform fewer optimizations when you are compiling for syntax checking and other simple error detection. At a later stage of development, you're likely to compile your program with greater amounts of optimization so the compiler can generate the most efficient code possible.

In certain cases, some programs may take longer to compile under SR9.5 and still not benefit significantly from the new optimizations in terms of execution speed. Therefore, you can compile such programs with a lower -opt level to reduce the amount of time they take to compile.

Previously, DOMAIN FORTRAN had two levels of optimization with the -opt and the -optall options. FORTRAN users were encouraged to use -optall (which is equivalent to C and Pascal's -opt) to get the *most* optimizations. FORTRAN's -opt option contained fewer optimizations but was the desirable optimization level for extremely large application programs. In such cases, using -optall took considerably more time to compile, but it did not improve runtime performance significantly.

Currently, FORTRAN's default -opt level is -opt 1, which generates about the same degree of optimization as FORTAN's previous -opt (except that -opt no longer performs any register optimizations at this level). Consequently, the compiletime for your program should not greatly increase. FORTRAN's -optall option is equivalent to -opt 3. You can specify an even greater amount of optimization with -opt 4. Note that even though DOMAIN FORTRAN continues to support the -optall option, you are encouraged to specify levels of optimization with -opt.

For C and Pascal, the default level for the -opt option is -opt 3. The number of optimizations that occur at this level are significantly greater than those performed under SR9.2 with the default -opt option. In most cases, your programs will not take significantly longer to compile and most programs will run as fast or faster than they did under SR9.2. In addition, the size of the generated code for your program is significantly reduced.

The expanded levels of optimization can make debugging optimized programs difficult. For this reason, the -dbs debug option now works independently of the -opt option. So you must specify *both* the -opt and the -dbs options on the command line to debug optimized programs. The -dbs option creates a line number table and a symbol table, while the -opt [1, 2, 3, 4] option turns on the particular level of optimization you want. If you cannot debug your progam successfully at that optimization level, you can reduce the level. As in previous releases, the -dba option does not perform *any* optimizations (even fewer than those performed at the -opt 0 level). So, -dba removes *all* optimization-induced obstacles to debugging.

When recompiling your program under SR9.5, you might receive new warning messages because of a new optimization. The most common warning you will see is a result of the **dead store elimination**, which we describe in the next section.

## 6.2. Eliminating Unused Assignment Statements

SR9.5 compilers optimize source code by eliminating an assignment if the new value is never used after the assignment. This is often referred to as **dead store elimination**. If the computation has side effects, such as a routine call or references to variables with a **device** attribute, the compiler will compute the *value* of the assignment, but will eliminate the *assignment to the identifier*. If the value computed was not a result of a function or device reference call, the *value computation* is eliminated also.

The following example shows how the compiler eliminates useless statements. In this example the compiler does not execute the first assignment to *m (m := j + i)*, because the program reassigns a value to *m* without ever using the initial value.

```
PROGRAM deadstore;
VAR
  i, j, k : integer;
  m, n    : integer;

BEGIN
    readln (j, k);
    FOR i := j TO k
    DO BEGIN
        m := j + i;                    {* Statement eliminated ! *}
        n := 3;
        IF (i > (j + n))
            THEN n := k + j;
        m := n;                        {* m reassigned before it's used. *}
        writeln (m);
    END;

END. {* deadstore *}
```

In some cases, you might have intended to create a *throw-away* value, so you can disregard the compiler's warning message. For example, your program could call a function and not care about what it returns. (In Pascal, you can suppress the warning message with the **discard** builtin function. See the Pascal documentation for details.)

*In many cases, though, the compiler's warning message is a good indication that your program is not doing what you intended.* Consequently, you should review your program for a possible error. For other details in Pascal, see Section 5.2., "Pascal's **in**, **out**, and **in out** Parameter Extensions."

## 6.3. Using volatile to Prevent Compiler Optimizations

To support better compiler optimizations, DOMAIN compilers are permitted to operate on variables (including variables within structures and arrays) in any manner that generates the most efficient code, as long as they obey the semantics of the source language. In most cases, this will make your program more efficient. However, the compiler might not perform optimizations correctly in the following situations:

- When a variable is in a shared memory location that is accessed by more than one process.
- When a variable can be accessed by two different paths, such as two or more pointers referring to the same data area.
- When a variable's value changes during an asynchronous fault (a CTRL/Q sequence), and the program continues execution. This is called an **asynchronous update**. Note that this is not usu-

ally a problem, because DOMAIN compilers do not optimize variables associated with DOMAIN's **pfm_$cleanup** or C's **setjmp/longjmp**.

The reason the compiler cannot perform optimizations properly is due to the way the compiler stores variables in memory. Previously, compilers updated memory after most operations. Now, the compiler can choose to save the variable in a register and *not* update memory until necessary.

In cases where the compilers cannot keep track of memory contents correctly, DOMAIN Pascal and C provide an extension that allows you to mark these variables with the **volatile** keyword. This keyword tells the compiler not to perform the optimizations that it normally would.

The following example demonstrates how a compiler can misinterpret data. The variable *i* can be accessed by two alternate paths. That is, the statement *p^ := p^ + 1* actually changes the value of *i* since *p* and *i* refer to the same memory location.

```
MODULE volatile_example;

VAR
  p : ^integer;
PROCEDURE init(VAR v : integer );
BEGIN
    p := addr(v);
END;

PROCEDURE top;
VAR
  i : integer;      { Correct this with:   i : [volatile] integer; }

BEGIN
    init(i);
    i := 0;
    WHILE i < 10 DO
        p^ := p^ + 1;   { Anonymous path: p and i point to same variable }
                        { Hidden modification to i }
END;
```

Since the compiler does not know that *i* and *p^* share the same memory locations, it believes that the value of *i* is always zero. So, the *while* loop becomes *while 0 < 10 DO;* which is an infinite loop. Note that this occurs because a program statement created an anonymous path. If you replaced the statement *p^ := p^ + 1* with *p := ADDR(i)* then the compiler notices that *i* has had its address taken and makes the variable implicitly volatile.

Similarly, if a program declares two pointers, *p* and *q*, with the same base type, the compiler conservatively treats all assignments to *p^* as a possible change to *q^*.

To ensure that the compiler evaluates the value of *i* correctly, you can declare it with the **volatile** attribute. To identify and mark the variables as volatile, you can do the following:

1. Locate any variables (usually pointers) that are shared by more than one process or that can be accessed by two different paths.

2. Mark the variable declaration with the keyword **volatile** as defined by the language syntax rules:

   In C, you declare a variable with the **volatile** specifier of the **#attribute** declaration modifier:

   ```
   extern char KEYBOARD #attribute[volatile];
   ```

   In Pascal, you signify a volatile variable with the **volatile** attribute extension:

   ```
   keyboard : [volatile] CHAR;
   ```

Note that all DOMAIN compilers attempt to recognize possible volatile variables associated with cleanup handlers. So, cleanup handlers will act as you would expect.

### 6.3.1. Rules for Using volatile in C and Pascal

The implementation of **volatile** is similar for both Pascal and C. This section lists how each compiler interprets the **volatile** attribute. Since FORTRAN does not support the appropriate syntax to add attributes such as **volatile**, the next section describes what you can do to prevent optimizations in FORTRAN.

When a variable is not marked as **volatile**, DOMAIN Pascal and C compilers optimize ordinary variable references any way that they see fit. They perform the following optimizations for external (global) variables:

- Save values of variables in registers during execution of the routine body.
- Do *not* save values of variables across routine calls.
- Write the value of a variable to memory after an assignment before completing the execution of a statement.

When you use the **volatile** attribute to suppress compiler optimizations, the compilers obey the following rules. Note that the only difference between the C and Pascal implementation concerns inheritance.

- C's **volatile** attribute is inherited by higher constructs. That is, if you mark a field within a struct with the **volatile** specifier, the compiler interprets the *entire* structure to be volatile. Consequently, the compiler will not perform any optimization on the entire structure. (This is compatible with other C implementations. However, this differs from DOMAIN Pascal.)
- Pascal's **volatile** attribute is *not* inherited. Consequently, you can mark a single field of a record to be volatile. The compiler will not perform any optimizations on that field, *but it can optimize other fields in the record.* (Note that this differs from DOMAIN C's implementation.)
- The **volatile** attribute prevents the compiler from saving a value in a register for more than one statement.
- The **volatile** attribute guarantees that the compiler will update memory by the end of the statement if the statement changes the contents of memory.

Note that another DOMAIN C and Pascal extension, the **device** specifier, is implicitly volatile. The **device** specifier informs the compiler that a device register (control or data) is mapped as a specific virtual address. In addition to suppressing the optimizations mentioned above, a **device** variable makes the following constraints on optimization. It:

- Prevents the compiler from merging references to adjacent memory locations into a larger reference.
- Guarantees that the compiler accesses the variable only once per source-language reference. So, if you need to access a variable or field more than once, you must use more than one statement.
- Tries to perform in a single instruction those operations that the 68000 can perform in one instruction. For example, the statement *v := v + 1* generates the instruction *ADD.W #1,V.*
- Prevents the compiler from generating unnecessary read-modify-write references for device registers. (This means that the compiler will avoid generating 68000 CLR and ST instructions for any **device** variables.)

For more information on the **volatile** and **device** specifiers, see the *DOMAIN Pascal Language Reference* (000792) and/or the *DOMAIN C Language Reference* (002093).

### 6.3.2. Preventing Compiler Optimizations in FORTRAN

FORTRAN does not support the appropriate syntax to add attributes such as **volatile.** To get nearly the same effect in FORTRAN as **volatile** has in Pascal and C, place any variables you want treated as volatile in **common,** and avoid using the –opt 4 optimization level. The FORTRAN compiler can save common variables in registers inside a routine, but it updates common variables to memory at each assignment statement. Note also that the variables declared with FORTRAN's **namelist** statement are implicitly volatile.

# Index

The letter *f* means "and the following page"; the letters *ff* mean "and the following pages".

## S



## T



## U



## V



## W

## Reader's Response

Please take a few minutes to send us the information we need to revise and improve our manuals from your point of view.

Document Title: *Making the Transition to SR9.5*

Order No.: 009492 Revision: 00

What type of user are you?

_____ System programmer; language _____
_____ Applications programmer; language _____
_____ System maintenance person
_____ System Administrator
_____ Student Programmer
_____ Other

_____ Manager/Professional
_____ Technical Professional
_____ Novice

How often do you use the DOMAIN system?_____

What parts of the manual are especially useful for the job you are doing?_____

What additional information would you like the manual to include?_____

Please list any errors, omissions, or problem areas in the manual. (Identify errors by page, section, figure, or table number wherever possible. Specify additional index entries.)_____

Your Name Date

Organization

Street Address

City State Zip

cut or fold along dotted line

FOLD

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

**BUSINESS REPLY MAIL**

FIRST CLASS PERMIT NO. 78 CHELMSFORD, MA 01824

POSTAGE WILL BE PAID BY ADDRESSEE

**APOLLO COMPUTER INC.**
**Technical Publications**
**P.O. Box 451**
**Chelmsford, MA 01824**

FOLD


*009492*