Delphi Optimization Guidelines

The guidelines presented here fairly general but with a Delphi specific focus. Additionally the focus is on modern Intel CPUs (Pentium and Pentium II). Other CPUs, for instance the AMD K6-2 and later, may also be good choices and may benefit from the optimizations listed here. However, they are not as well documented publicly so it is difficult to know for sure.

The guidelines fall into two general types: 1) coding styles and 2) specific optimization techniques. When writing code, it is common for there to be multiple ways to proceed. Some of these ways generally tend to result in faster code. You might call it passive optimization. This is the "coding style" type of optimization. However, for those specific routines that are performance bottlenecks this is often not sufficient. In these cases you need to actively optimize the routine. The guidelines that fall into this group are of the second type, "optimization techniques". Additionally, they are divided into functional groups:

Note: that all the techniques presented here assume that optimizations are ON. (Obvious, but needs to be said nonetheless.)

General Optimization Guidelines

Style Guidelines

Coding Style versus efficiency

Optimization involves not only the speed of your code, but also the speed with which you create and debug your code. This means that you are not doing yourself any favors by creating fast but incomprehensible code. Fortunately creating optimal code in Delphi rarely requires ugly code. In fact, optimal code is often elegant code as well. Additionally, within a given application it is likely that the same sort of techniques will be used frequently. Thus, you can essentially set the coding style to what gives the best performance when it matters.

Keep it simple

When it comes to the Delphi optimizer, complexity kills. Keep routines simple (no more than about 5 to 8 variables "in play"). Do not do too much in any single loop. Overloading a loop causes variable addresses, array indices etc. to be reloaded on each iteration. Loop overhead is actually quite low so it is often advantageous to split a complex loop into multiple loops or to move the innermost one or two loops to a separate routine. Done properly, this will have the added benefit of improving the readability of your code.

Strongly favor local variables

Local variables are those declared within a routine in addition to any parameters passed. Only local variables can be converted into register variables, and register variables equal speed. Consequently, it is often advantageous to copy data into a local variable prior to using it. Typically this is most advantageous when the variable is to be used within a loop. Thus, the overhead of copying is offset by speedy reuse of the copied data. This optimization is particularly useful if class members are used in a tight loop. Delphi tends to load the pointer / class member just prior to its use *within* the loop, adding a lot of unnecessary overhead.

There is one exception to this rule: arrays with elements of a simple type. If you have an array of constant size and constant data, making it global will save a register during calculations. Since saving a single register is not worth a lot, this should only be used for constant structures (conversion or transformation tables) where having a global structure makes some sense to begin with.

Keep the number of parameters low

Small but heavily used routines should not have more than three parameters, as that is the maximum that can be passed by register. By following this rule you maximize the use of registers and give the Delphi optimizer a better chance to improve your code. Note that class methods have a hidden parameter `Self` that is always passed implicitly so for these only two parameters are left.

Do not use nested routines

Nested routines (routines within other routines; also known as "local procedures") require some special stack manipulation so that the variables of the outer routine can be seen by the inner routine. This results in a good bit of overhead. Instead of nesting, move the procedure to the unit scoping level and pass the necessary variables - if necessary by reference (use the `var` keyword) - or make the variable global at the unit scope.

Pointer variables

A valuable technique is to take advantage of pointers. A lot of programmers shy away from pointers due to the potential for access violations, memory leaks and other low-level problems. However, pointers are a valuable tool in optimizing code in Delphi. Fortunately, this does not mean you have to convert all your data references to pointers. What it does mean is that you should take advantage of pointers as temporary references into your data. These temporary variables will typically be optimized into register variables. Consequently, you really are *not* adding any machine code, you are only providing a clue for the compiler that it needs to hold on to the intermediate address. You can use pointers much the same way as you would use a `with` statement. That is, use them to simplify complicated or redundant addressing in complex data structures. In the case of `with` statements, this is exactly what the compiler does internally. For example:

```
with Structure1.Structure2[i] do
begin
  ...
end;
```

at the compiler-level becomes:

```
InnerStructure := Structure1.Structure2[i];  // if it is a class
InnerStructure := @Structure1.Structure2[i];  // some other type
begin
  ...  // references to InnerStructure
end;
```

Linked lists vs. Arrays

Finding the trade-off between linked lists and arrays is a classic design problem. On older computers (Pentium and before) integer multiplication was a slow operation. Since multiplication is the key to accessing arrays this shifted the performance balance towards Linked lists in some cases. Is it random access or sequential access? Obviously, if you truly need random access then an array is the way to go for anything more than about 5 elements (this is a rule of thumb based on experimentation). For sequential access or quasi-sequential access the short answer is that arrays are better for simple element types and linked lists are better for larger types.

Multiplication on the Pentium II is now much much faster. Consequently, array access is always faster.

Types of Arrays

In Delphi, arrays come in many flavors: Static, Dynamic, Pointer and Open. Static arrays are the classic Pascal array type (A: array[0..100] of Byte). Dynamic arrays are the new array type introduced with Delphi 4 (A: array of Byte). Pointer arrays are simply pointers to Static arrays, however, the actual number of elements may not match that of the array boundaries. Finally Open arrays look like dynamic arrays but are exclusively used as parameters of routines. The underlying implementation of all these arrays varies quite substantially. From an efficiency viewpoint Static and Pointer arrays are the best choice, followed by Open, then Dynamic arrays. However, Static arrays are often too inflexible, and Pointer arrays can create painful management issues. Fortunately, the various types are convertible. For arrays without a fixed size, the current best choice is to *manage* them as Dynamic arrays, and convert them to Pointer arrays as needed.

Dynamic arrays are much like huge strings (AnsiStrings) in that the variable is actually a pointer to the array's first element. Thus, converting a Dynamic array to a pointer arrays is simply an assignment. While the length (size) of the Dynamic array is stored just before the first element, using `High` or `Length` on a Dynamic array generates a function call rather than some compiler "magic" to extract the length - in fact, `High` calls `Length`. Consequently, do not repeatedly get the size of the array via these functions. Get it once (in the routine) and save it.

Both Dynamic and Open arrays incur a fair amount of overhead when used as parameters unless you use the `const` or `var` modifiers. Note that, just like class parameters, a `const` modifier on a Dynamic array only prevents you from changing the *entire array*, not from modifying its *content*.

Let's finish with an example for converting the various array types:

```
type
  TDoubleArray = array of Double;
  TStaticDoubleArray = array[0..0] of Double;
  PDoubleArray =^ TStaticDoubleArray;

function Sum(const X: TDoubleArray): Double;
var
  P: PDoubleArray;
  i: Integer;
begin
  P := Pointer(X);
  Result:=0;
  for i := 0 to Length(X)-1 do
    Result := Result + P[i];
end;
```

Exceptions

Do not use exceptions just to jump out of a bit of code, or as a catch-all on input errors. They add overhead with both the try..finally block and with throwing the exception itself. Use the break, continue, or exit statements to do unusual flow control, and validate inputs (like pointers) as early as possible, but outside any loop.

Use type-casting rather than `absolute`

A technique sometimes used to avoid typecasting is to "overlay" a variable with another of a different type by using the `absolute` keyword. However, this prevents the variable from becoming a "fast" register variable. It is better to type-cast and save the original variable into a new variable. For example:

```
procedure DoSomething(s: PChar);
var
  ByteArray: PByteArray absolute s;
begin
  ...
```

should be changed to or written as:

```
procedure DoSomething(s: PChar);
var
  ByteArray: PByteArray;
begin
  ByteArray := PByteArray(s);
  ...
```

Working with Sets

There are two compiler magic functions called Include and exclude, that are quite efficient for adding and subtracting single elements form sets. Thus, you should use these instead of " `s:=s+[a];`" sort of statement. In fact, it is efficient enough that a small number of repeated uses of Include or exclude can still be better than the above notation.

Pentium II specific bottlenecks

It has occurred to me that while many of the techniques presented here are based upon how Pentium II processors bottleneck, I have never actually stated how this works. The long and detailed version can be found in Intel's documentation and in Agner Fogs Pentium Optimization Guide. Here I present a quickie version slanted towards Delphi's compiler output. Having a general understanding of this process will may help you decide what needed optimizing and what does not.

First off, the Pentium II is a superscalar pipelined processor with out-of-order execution capabililities. Basically that means that each instruction gets "executed" in steps and can march along one of a few different channels. Specifically, each instruction has to be loaded, executed and retired with the out-of-order buffer acting as a sort of "waiting room" between the load and execution steps. This seems simple enough, but the complications start to build once the multiple channels part is added in, because not all channels can handle all instructions. There are 3 loading channels, one of which can take anything while the other two can only handle "simple" instructions. There are 5 execution channels (called ports by Intel), one is general purpose integer, another is general purpose integer plus floating point, the third handles address math, and the fourth and fifth load and store data. The retiring step also has 3 channels. There is also the issue of latency. That is, many instructions take longer than 1 cycle to execute.

So what does all this mean? Well, it means you can bottleneck in a whole bunch of different ways. Basically, any channel of any step can be a bottleneck if too many instructions that require that specific unit are encountered. Thus, while the CPU can theoretically process 3 instructions per cycle, it may be limited to 2 or 1 or even less due to the mix of instructions currently being executed. The out-of-order "waiting room" helps with this situation by allowing instructions not dependant on a currently executing operation to go around any that may be waiting for a specific port and execute on a different port. This helps with the small bottlenecks where there is a temporary backup on a given port. However, it does nothing for the large scale backups. For instance, executing a large series of floating point operations, say a loop around a complex math expression, will typically be constrained by the limitation of there being only one FP capable port. Thus, the throughput drops to 1 instruction/cycle. Now the loop also has some other overhead instuctions associated with it (incrementing the loop variable and jumping) These instructions incur essentially zero cycles since they can be fitted in around the backed up FP port.

In Pascal terms, this means that any tight, repetative operation will probably be constrained by only one aspect of the operation. It might be Floating point as described above, or it might be integer math or memory addressing. In any case, the only optimizations that will have any impact are those that go after that main aspect. Pruning the ancillary code back will have no effect.

Inside the `For` statement

Implementing `For` statements is one of the more complex jobs the compiler has to deal with. This is in part, because the compiler goes to great pain to avoid integer multiplication, which was quite slow on CPUs before the Pentium II. Thus, `For` loops are deconstructed into pseudocode that looks something like this:

Original Loop:

```
For i:=m to n do
      A[i]:=A[i]+B[i];
```

Becomes:

```
PA:=@A[m];
PB:=@B[m];
counter=
m-n+1; ifcounter>0 then
  repeat
        PA^:=PA^+PB^
        inc(PA);
        inc(PB);
        dec(Counter);
      until counter=0;
```

There are other configurations, but this is the most common, and it is the one that causes problems. The problem stems from the fact that the variable `i` appears nowhere in the deconstructed version. However, when stepping through this code in the debugger, watching `i` will show the value of the new variable most similar to `i`, which is `counter`. This has sent many a programmer into fits of hysteria thinking that their loop is being executed backwards. It is not. The debugger is merely misinforming you.

This example also illustrates the substantial overhead associated with for loops. Notice that three variables need to be incremented on each iteration, and that there is a fair amount of initialization code. In some cases, this overhead is lessened. For instance if `m` and `n` were compile-time constants, Or if m=0, and A and B were already pointers that were not used again in the code, then overhead code would be reduced.

Interfaces

This is just a beginning pass at the performance implications of using interfaces. Basicaly, an interface is implemented as a cross between a string and a class. Interfaces, like strings are reference counted which means every time you create or copy one, and every time an interface variable goes out of scope there is some overhead. Thus, treat interface variables more like you would string variables than object variables. (Pass by `const` where possible, watch out for using too many temp variables, etc.) Internally, interfaces behave something like an object with all virtual methods only worse. There are in fact two layers of indirection. So treat them accordingly.

Another note on the subject of interfaces:

Hallvard Vassbotn: For any interfaced global variable, the compiler automatically adds another global variable that contains the address of this first variable. When external units access the variable, it is used through the implicit pointer variable. The reason for this is to support packages and is done even if you are not using packages. (See article in The Delphi Magazine issue 43 for more details.)

Optimization Techniques

Keep an open mind

Optimization is best approached as a top down issue. The most powerful concept in optimization can be stated as "If it takes too long to figure out the answer, then change the question." The best improvements in performance will always come from changes made at the design and algorithmic level. By the time you get down to coding specifics, your options are quite limited. Unfortunately, like the rest of design, it is rather difficult to break down this high-level optimization into a nice set of rules. Nonetheless, the first thing to do if performance needs improvement, is to look at the complete problem at hand, starting optimization at the top and then working down.

Time your code

Timing code is generally called "profiling". If you want to improve the performance of your code, you first need to know precisely what that performance *is*. Additionally, you need to re-measure with each change you apply to your code. Do not spend a single second twiddling code to improve performance until you have analytically determined exactly where the application is spending its time. I cannot emphasize this enough.

Code alignment

Be aware that the exact positioning of your code and its layout in the executable module can affect its timing. The reason for this is that there are penalties for jumping to "sub-optimal" address alignments. There is very little you can do to influence this alignment (this is a linker task), except to be aware of it. While it is possible to insert spacing code into a unit, there is on guarantee that your alignment efforts will be permanently rewarded since 32 bit Delphi aligns only on 4 byte (DWord) boundaries. Thus, the next change to any routine in any unit above the current routine may shift your code. This problem can result in speed penalties as great as 30% in tight loops. However, in more typical loops the problem is substantially less. Also note that this can make timing code and therefore optimization difficult since seemingly innocuous shifting in the code can affect the performance. Consequently, if you make a change that you "know" should increase performance but it does not, it may be the shifts in code alignment are hiding improvements in your code.

Utilize the CPU window

You do not need to be an assembler programmer to take advantage of the CPU window. At the very least it will give you an idea of the underlying complexity involved in each statement. Very often you can estimate the effectiveness of a particular optimization technique by simply counting the number of instructions produced for a given operation. For instance, many references to the ebp register (as in `mov eax,[ebp-$04]`) within a loop are an indication that variables are continually reloaded. These reloadings are unnecessary and thus are a prime target for optimization.

In Delphi 4.0, the CPU window is readily accessed from the main menu. However, both version 2.0 and 3.0 of Delphi also have hidden CPU windows. To allow access to these you need to add an entry in the registry, using the registry editor "RegEdit.exe":

```
[HKEY_CURRENT_USER\Software\Borland\Delphi\2.0\Debugging]
"EnableCPU"="1"

[HKEY_CURRENT_USER\Software\Borland\Delphi\3.0\Debugging]
"EnableCPU"="1"
```

Unroll small loops

Loop unrolling is a classic optimization technique, and it can be easily done in Delphi. However, it is only worth doing on fairly *small* loops. Unrolling essentially consists of doing what was originally multiple iteration operations within a single pass through the unrolled code. This reduces the relative loop overhead. Since the branch prediction mechanism on Pentium II CPUs does not perform well (read: causes penalties) on very tight loops, unrolling might be beneficial there, too.

In Delphi, the best way to unroll is usually with a `while` loop. For example:

```
i := 0;
while i < Count do
begin
  Data[i] := Data[i] + 1;
  Inc(i);
end;
```

becomes:

```
i := 0;
while i < Count do
begin
  Data[i] := Data[i] + 1;
  Data[i+1] := Data[i+1] + 1;
  Inc(i, 2);
end;
```

The downside to loop unrolling is that you have to worry about what happens when `Count` is not divisible by the factor two. You typically handle it this way:

```
i := 0;
if Odd(Count) then
begin
  Data[i] := Data[i] + 1;
  Inc(i);
end;

while i < Count do
begin
  Data[i] := Data[i] + 1;
  Data[i+1] := Data[i+1] + 1;
  Inc(i, 2);
end;
```

You can unroll by whatever factor you want. However, the diminishing marginal return on unrolling by progressively larger values coupled with the growing complexity of the code makes unrolling by more than a factor of 4 rather uncommon.

Eliminate conditionals within loops

It is common for there to be `if` statements within a loop with the conditionals for the statement based on the loop index. Frequently, these can be removed from the loop by unrolling the loop or splitting the loop into two loops. An example of the former would be when statements must be executed every other iteration. An example of the latter would be when statements are executed on a specific iteration.

Reduce the number of looping conditionals

A common coding structure is to loop while some condition is true and while the loop index is less than some value. If the loop is small - and it often only consists of incrementing the loop index - then the bulk of the loop's execution time is spent evaluating the loop conditionals. It is sometimes possible to reduce the number of these conditionals by making one condition happen when the other would have. Take the example of scanning for the occurrence of a specific character in a string:

```
i := 1;
l := Length(s);
while ((i <= l) and (s[i] <> c)) do
  Inc(i);
...
```

The two conditionals can be combined by placing the desired character in the last position in the string:

```
i := 1;
l := Length(s);
```

```
lastc := s[l];
s[l] := c;
while s[i] <> c do
  Inc(i);
s[l] := lastc;
...
```

This results in nearly a 2x speed improvement. This optimization requires some forethought to ensure that there is an empty space available at the end of the data. Strings and PChars always have the null at the end that can be used. Also this technique, may cause undesired side-effects in a multi-threaded environment due to changing the data being scanned. This technique also works well with unrolling, as it often simplifies the problems associated with needing partial iterations. See `FindMax` as an additional example of this technique.

Make the default path, the "no-jump" path

This technique dates way back. However, there is still a glimmer of truth in it. The original reason (jumping took a lot of time) really isn't the problem now. The problem now is more related to code alignment and branch prediction. On the alignment side, if you do not jump than there is no problem with alignment. On the branch prediction side, a branch is not even considered for prediction until it is actually taken once.

Take advantage of break, exit and continue

These flow control statements are often derided as being "bad programming". However, they do have their place, especially in performance optimizing code. The need for these statements typically arises when some condition is not determined until the *middle* of a loop. Often they are avoided by adding boolean variables and additional conditionals to transfer control. However, these additions cost execution time, and can often make the code look more, rather than less, complex.

Resorting to assembler

Do not attempt to use assembler to improve performance on Pentium II CPUs. This is somewhat controversial, but it is a pretty good rule of thumb. The out-of-order execution capabilities of Pentium II class CPUs pretty much eliminate any advantage you might gain by recoding your algorithm in assembler. In several tests, I have found that assembler vs. optimally coded Pascal rarely exceeds a 10% difference. There are always exceptions to this rule (for instance this Alpha Blending code) and even times where arguably assembler is cleaner than Pascal, but the point is that you should not just jump to assembler when code seems too slow.

On the other hand, Pentium CPUs can often benefit from assembler coding. Improvement factors of around two are not uncommon. However, optimal coding for the Pentium processor can easily result in rather non-optimal code on other processors. This applies to floating point code in particular. If you choose to pursue this path then study Agner Fog's assembler optimization manual carefully.

`For` versus `While` loops

Loops with a pre-determined number of iterations can be implemented either with a `For` loop or a `While` loop. Typically, a `For` loop would be chosen. However, the underlying implementation of the `For` loop is not as efficient as that of the `While` loop in some instances (See Inside the `For` statement). If the contents of the loop involve no arrays or only single-dimensional arrays with elements of sizes 1, 2, 4, or 8 bytes, the code generated for a `While` loop will be more efficient and cleaner than for the comparable `For` loop. On the other hand, multi-dimensional arrays or arrays with elements of other sizes as those listed above are better handled by `For` loops. It is often possible to convert one of the latter into the former, typically by using pointers. This approach is likely to increase the efficiency of the code.

Additionally, using a `While` loop appears to reduce the complexity factor. Thus it may be possible to trade `For` loop usage against splitting up a routine. An example of this is the row reduction step of Gauss Elimination. The optimal configuration with `For` loops is to factor out the two innermost loops into a separate routine. With `While` loops however, all three loops can be kept together.

Of course there has to be an exception. If the index is never used within the loop then `For` is usually a better choice. Also give `For` as shot if both loop bounds are compile-time constants.

Note that for a `While` loop to be as efficient as possible, the loop condition should be as simple as possible. This means that, unlike `For` loops, you need to move any calculation of the iteration count out of the `While` statement and into a temporary variable.

Large Memory requirement problems

On Pentium II class CPUs it is often the case that cache or memory bottlenecks are the main optimization problem, especially if the data set being manipulated is large. If this is the case, then an entirely different strategy is in order. Focus on reducing the memory requirements and on reducing the number of passes through the data. In other words, pack it tight and do as much as possible with it before moving on. This may be at odds with some of the other suggestions presented here, so it is necessary to determine which factor is more rate limiting by experimenting with different implementations and profiling these. A good indication that a cache and/or memory bottleneck is dominating is when apparent improvement in the code being executed does not increase the performance.

`Case` statement optimization

`Case` statements are implemented as follows: First,the compiler sorts the list of enumerated values and ranges. This means that the placement individual cases within the `Case` statement is irrelevant. Next, The compiler uses a sort of binary comparison tree strategy along with jump tables to test the cases. The decision between jump table and comparison tree is based on the "density" of the enumerated cases. If the density is sufficiently high a jump table will be generated. If the density is too low then the list will be split approximately in half (with ranges counting as 1 element in the list rather than as the number of values spanned). The process then starts over on each of the sub-branches. That is, density check then either generate jump table or split. This continues until all the cases are handled.

So what optimization possibilities exist? Basically, it boils down to this. `Case` statements are quite well optimized, but not perfect. The "splits" on the binary comparison tree can come at awkward places. Consequently, if you have groups of consecutive values interspersed with gaps, it is better to make each range of consecutive values its own `Case` statement and then make an overall `Case` statement with the underlying ranges each being a single case. This works because ranges won't be split but sequential cases will be. The impact of this is that it tends to create jump tables covering each entire sub-range. An Example:

Before:

```
Case x of
  100 :DoSomething1;
  101 :DoSomething2;
  102 :DoSomething3;
  103 :DoSomething4;
  104 :DoSomething5;
  105 :DoSomething6;
  106 :DoSomething7;
  107 :DoSomething8;
  200 :DoSomething9;
  201 :DoSomething10;
  202 :DoSomething11;
  203 :DoSomething12;
  204 :DoSomething13;
  205 :DoSomething14;
  206 :DoSomething15;
  207 :DoSomething16;
  208 :DoSomething17;
  209 :DoSomething18;
  210 :DoSomething19;
end;
```

After:

```
Case x of
  100..107 :
        case x of
      100 :DoSomething1;
      101 :DoSomething2;
```

```
        102 :DoSomething3;
        103 :DoSomething4;
        104 :DoSomething5;
        105 :DoSomething6;
        106 :DoSomething7;
        107 :DoSomething8;
          end;
        200..210 :
          case x of
        200 :DoSomething9;
        201 :DoSomething10;
        202 :DoSomething11;
        203 :DoSomething12;
        204 :DoSomething13;
        205 :DoSomething14;
        206 :DoSomething15;
        207 :DoSomething16;
        208 :DoSomething17;
        209 :DoSomething18;
        210 :DoSomething19;
      end;
  end;
```

Also, `Case` statements do not have any provision for weighting by frequency of execution. If you know that some cases are more likely to be executed than others. You can use this information to speed the execution. The way to achieve this is to cascade `if` and `Case` statements to prioritize the search order. An Example:

Before:

```
Case x of
  100 :DoSomething1;
  101 :DoSomethingFrequently2;
  102 :DoSomething3;
  103 :DoSomething4;
  104 :DoSomething5;
  105 :DoSomething6;
  106 :DoSomething7;
  107 :DoSomething8;
end;
```

After:

```
if x=101 then
  DoSomethingFrequently2
else
  Case x of
    100 :DoSomething1;
    102 :DoSomething3;
    103 :DoSomething4;
    104 :DoSomething5;
    105 :DoSomething6;
    106 :DoSomething7;
    107 :DoSomething8;
  end;
```

Moving and zeroing memory

The built-in methods supplied with Delphi for moving memory and filling it with zeros are `Move` and `FillChar` respectively. These routines are based around the `rep movsd` and `rep stosd` assembler instructions, which are fairly efficient. However, there is some extra cleanup code associated with each of the routines that can reduce their efficiency, especially when working on smaller amounts of memory. Additionally, there are special data alignment considerations on Pentium II CPU's that can have a substantial effect.

The first pass solution to these issues is simply to use a plain loop to do the task. This is especially effective if the data elements being handled are 32 or 64 bit or the structure involved is only partially zeroed/moved (e.g. sub-section of a matrix). However, the loop approach is less effective for arrays of large records or for smaller elements like byte or word. You can unroll the loop and use typecasting to further improve the situation but this complicates the code substantially and only results in a small improvement.

At this point it is time to start looking at a specialized routine. The first issue is the boundary size of the structure. Working with 32bit quantities is always the fastest approach. However, if the structure is not evenly divisible by 4 bytes this can be a problem. The solution used by `Move` and `FillChar` is to round down the size to the nearest dword (4 byte) boundary, then copy the

remainder separately. As was mentioned, this extra overhead can be costly on smaller structures. However, many structure are in fact evenly divisible even if they do not at first appear to be. All memory allocations are rounded up to the next dword boundary. Thus a string of 2 characters is really 4 bytes long. It is usually faster to copy or zero this extra data than to avoid it. Obviously care must be taken when using this shortcut and it should be well documented.

Dealing with the data alignment issue is more complicated, and more relevant only on larger structures. I will skip the description and simply show the code:

```
procedure ZeroMem(A:PDataArray; N:integer);
var
  i,c:integer;
  B:PDataArray;
begin
  B:=Pointer((integer(A)+15) and $FFFFFFF0);
  c:=integer(@A[N])-integer(B);
  fillChar(A^,N*SizeOf(TData)-c,#0);
  fillChar(B^,c,#0);
end;
```

This will align on a 16 byte boundary by skipping over some of the data. Of course the skipped part must be properly dealt with, thus the two calls to fillChar. Obviously, this is not *the* fastest approach since you have now got the overhead of fillchar*2, but it does illustrate the technique. For maximum speed, this is one of those cases where you have to resort to assembler.

```
procedure ZeroMem32(P:Pointer;Size:integer);
// Size=number of dword elements to fill
// assumes that Size>4
asm
  push edi
  mov ecx,edx
  xor edx,edx
  mov dword ptr [eax],edx
  mov dword ptr [eax+4],edx
  mov dword ptr [eax+8],edx
  mov dword ptr [eax+12],edx
  mov edx,eax
  add edx,15
  and edx,-16
  mov edi,edx
  sub edx,eax
  shr edx,2
  sub ecx,edx
  xor eax,eax
  rep stosd
  pop edi
end;
```

The move version is very similar. The Dest pointer is the one aligned:

```
procedure MoveMem32(Src,Dest:Pointer;Size:integer);
// Size=number of dword elements to fill
// assumes that Size>4
asm
  push edi
  push esi
  push ebx
  mov ebx,[eax]
  mov [eax],ebx
  mov ebx,[eax+4]
  mov [eax+4],ebx
  mov ebx,[eax+8]
  mov [eax+8],ebx
  mov ebx,[eax+12]
  mov [eax+12],ebx
  mov ebx,edx
  add ebx,15
  and ebx,-16
  mov edi,ebx
  sub ebx,edx
  shr ebx,2
  sub ecx,ebx
  lea esi,[eax+4*ebx]
  rep movsd
  pop ebx
  pop esi
  pop edi
end;
```

Global Data revisited

There is a case where using a global structure is especially advantageous, namely two-dimensional (or rather double-indexed) arrays with elements of a simple type and where access is non-sequential for both indices. By making the data structure global, both indices can be applied simultaneously to access the structure, thereby avoiding additional instructions to combine the indices.

`While` loops revisited

An additional technique that can be applied to `While` loops operating on arrays that saves on CPU registers is to shift around all the references to the index variable so that you can count from a negative number toward zero. This frees the register that would have been needed to hold the iteration count. For example:

```
i := 0;
while i < Count do
begin
  Data[i] := Data[i] + 1;
  Inc(i);
end;
```

becomes:

```
type
  TRef = array[0..0] of TheSameThingAsData;
  PRef = ^TRef;
var
  Ref: PRef;
...
  Ref := @Data[Count];
  i := -Count;  // Assign NEGATIVE count here
  while i < 0 do  // and count UP to zero
  begin
    Ref[i] := Ref[i] + 1;
    Inc(i);
  end;
```

Pointer variables revisited

In addition to the reduced dereferencing discussed above, using a pointer variable can also serve to increase the "priority" of an already existing pointer variable. In the example shown below, taken from a Sub-String replacement routine, the PChar variable `pSub1` was being reloaded within the loop (this could be seen in the CPU Window). By assigning it to `pTemp` and then using `pTemp` within the loop, the loading was shifted outside the loop, saving instruction cycles.

```
pTemp := pSub1;  // increases "priority" of pSub1
while iStr[k] = pTemp[k] do
  Inc(k);
```

Avoid checking methods pointers with `assigned`

Checking method pointers for nil is a common operation typically associated with calling events. Unfortunately, `if assigned(FSomeEvent) then ...` does a 16bit compare of the high word of the code address for the method pointer. This is rather odd and completely unnecessary, and I can only guess that it is some sort of holdover from 16bit Delphi 1. The workaround is to check the code address directly (`if assigned(TMethod(FSomeEvent).code) then ...`. This is a bit ugly and so you may only want to follow it in particularly time critical sections.

Controlling the size of enumerated types

If you use enumerated types (such as TSuits= (Diamonds,Hearts,Clubs,Spades)) include the `{$MinEnumSize 4}` (or `{$Z4}`) directive to force all enum variables to be 32bit. If you have compatibility issues you can simply turn it on for the type declarations of interest. For instance:

```
type
{$Z4}
  TSuits=(hearts,clubs,diamonds,spades);
{$Z1}
```

Utilization of this directive is especially effective for enumerated types greater

than 256 elements. These result in word sized variables which are quite slow.

Virtual methods

It should not be surprising that virtual methods incur more overhead than static methods. Calling a virtual method requires two pointer dereferences and an indirect call which, the method has a couple of parameters approximately doubles the total call overhead. However, there is the potential for much more severe penalties. The indirect call can suffer from what amounts to branch misprediction which is a fairly stiff penalty on Pentium II processors. The penalty is incured each time the target of the call changes. Thus, calling virtual method within a loop where the method might change on every iteration could see a substantial number of penalties. The workaround is essentially to sort the method calls.

Example:

```
  TBaseClass=class
  public
    procedure VMethod; virtual;
    procedure SMethod;
  end;

  TDerivedClass=class(TBaseClass)
    procedure VMethod; override;
  end;

  TDerived2Class=class(TBaseClass)
    procedure VMethod; override;
  end;

implementation

type
  TArray=array[0..100] of TBaseClass;

procedure DoStuff;
var
  b: integer;
  j: integer;
  A:TArray;
begin
  A[0]:=TBaseClass.Create;
  b:=0;
  for j := 1 to 99 do
  begin
    b:=(1+random(2)+b) mod 3;  // mix em up
    case b of
      0: A[j]:=TBaseClass.Create;
      1: A[j]:=TDerivedClass.Create;
      2: A[j]:=TDerived2Class.Create;
    end;
  end;
  for j := 0 to 99 do
        A[j].VMethod;
  for j := 0 to 99 do
        A[j].SMethod;
end;
```

Sorting the calls is somewhat complicated, an example is shown below:

```
Type
  TSomeVirtualMethod=procedure of object;
  TSomeMethodArray=array[0..100] of TSomeVirtualMethod;

var
  SomeMethodArray:TSomeMethodArray;

//    Initialization pass
  for i:=0 to Count-1 do
    SomeMethodArray[i]:=Item[i].SomeVirtualMethod;


//  Do something passes
  for i:=0 to Count-1 do
    SomeMethodArray[i];
```

This, by itself, saves an underwhelming 1 clock cycle per call, but you can sort the array by the code that's called (using TMethod) to minimize the oh so painful branch prediction failure that can really dominate this kind of method calling. Additionally, if the base class method is some sort of do nothing routine it could be eliminated from the procedure list entirely.

```
// initialization pass

for i:=0 to Count-1 do
begin
  Hold:=ClassArray[i].SomeVirtualMethod;
  if TMethod(Hold).Code<>@TBaseClass.SomeVirtualMethod then
  begin
    j:=0;
    while (j>ArrayCount) and (longint(TMethod(Hold).Code)<SomeMethodArray[j]) do
      inc(j);
    for k:=ArrayCount-1 to j do
      SomeMethodArray[k+1]:=SomeMethodArray[k];
    SomeMethodArray[j]:=Hold;
    inc(ArrayCount);
  end;
end;
```

This obviously isn't for the faint of heart, and is only useful in certain situations, but it could be a big time saver in cases where there are many objects, but only a few versions of the method and the method is relatively small or empty.

Integer Optimization Guidelines

Contents

Style Guidelines

Use 32 bit variables whenever possible

In 32 bit code, such as generated by Delphi 2 and later, things just get better when the values being manipulated have a size of 32 bits. 16 bit variables (Word, ShortInt, WideChar) are especially slow as they require the processor to temporarily slip into 16 bit mode to work with them. This can double the time it takes to work with these values. 8 bit variables (Byte, SmallInt, Char) are not as bad, especially, if you do not mix their usage with 32 bit values. However, they can still cause the inclusion of additional instructions in order to zero out the rest of the 32 bit register.

If you must use a smaller type for compatibility, convert it to 32 bit as soon as possible, and back to the smaller size (if necessary) just prior when it is needed. You do this simply by assigning it to a 32 variable.

Avoid ordinal subranges

One of the advantages of Pascal has traditionally been its strong typing, and so the ability to create special subrange types and enumerations has been part of this. Unfortunately, subranges and enumerations can cause trouble when attempting to optimize for performance. The problem lies in the fact that the underlying variable type choosen hold a subrange or enumeration variable is based on the size of the subrange. For example, enumerations with less than 256 elements or subranges with boundary values ranging between 0 and 255 will be stored as byte. This can lead to trouble in that the underlying variable size may not be handled as efficiently. For instance, consider the following subrange:

```
type
  TYear=1900-2000;
```

Variables of type TYear will saved as 16bit quantities. As already discussed, 16bit variables are particularly slow.

Optimization Techniques

Play around with adding temporary variables to split up complex expressions

Typically, cramming everything into a single expression is the best way to optimize, but not always. At some point, the expression will become so complex that the compiler will be forced to break it up on its own. But frequently you can do a better job than the compiler. Try it!

Integer multiplication

Prior to the Pentium II, integer multiplication was quite expensive. With the arrival of the Pentium II however, integer multiplication has dropped down to the same one-cycle execution time as most other instructions. Additionally, the compiler will avoid doing multiplication by a constant if the same can be accomplished by utilizing addition, shifting and the lea instruction (mentioned below). Thus, you need to take your target processor into account when choosing whether to use multiplication or use some other equivalent method.

Comparison of a variable against multiple ordinal constants

This topic sounds heavy but only boils down to statements like these:

```
if (x > = 0) and (x < = 10) then
```

```
  DoSomething;

if (((c > = 'a') and (c < = 'z')) or
    ((c > = '0') and (c < = '9'))) then
  DoSomething;
```

In each case, there is only a single variable and it is compared to multiple constants. It may be slightly more efficient and arguably clearer when the above code is expressed as:

```
if x in [0..10] then
  DoSomething;

if c in ['0'..'9', 'a'..'z'] then
  DoSomething;
```

The improvement in efficiency depends upon the likelihood of, for instance x bein within the range versus being out of the range. If it is more likely to be within the range then the set notation is better. Efficiency of the set notation increases as the number of sub-ranges increases. However, there is an inherent limitation on sets that limit them to 256 elements. This restricts this usage to values between 0 and 255 for integer types. For the full range of integers you can use this notation:

```
case x of
  0..10: DoSomething;
end;

case c of
  'a'..'z',
  '0'..'9' : DoSomething;
end;
```

which produces equivalent code, but is not as elegant.

Advanced note: This operation may use an additional CPU register.

movzx vs xor/mov

A common requirement is to load values smaller than 32 bits in to a register. Since they do not overwrite the entire register it is necessary to zero out the register first. Alternatively, you can use the built in instruction movzx (move with zero extend). On Pentiums and before this instruction was slower than using xor reg,reg/mov reg,{value}. However, The PII has streamlined this instruction so that now it is prefered over the xor/mov combination. Note that the compiler chooses between these two options based on a set of rules that is apparently fairly complicated, as I have yet to figure them out.

Utilizing the LEA assembler instruction

There is an assembler instruction called LEA (**L**oad **E**ffective **A**ddress) that can do a couple operations at once. The only way to consciously take advantage of this instruction in Delphi is with array notation. For it to be fully effective, the array variable itself must be used again after the desired "trick" location. For example the following snippet is from a routine that calculates the length of a Pchar (StrLen) string (i.e. the position of the first #0 character). A total of *four* characters are processed at a time. Notice the usage of q in the calculation of r2.

```
function StrLenPas(tStr: PChar): integer;
var
  p: ^Cardinal;
  q: PChar;
  bytes, r1, r2: Cardinal;
begin
...
  q := PChar(p^); // load 4 characters into q
  r2 := Cardinal(@q[-$01010101]); // subtract 1 from each char (utilizing LEA)
  r1 := Cardinal(q) xor $80808080; // check top bit (q must be used again)
  bytes := r1 and r2; // distinguish between chars>127 and zero.
  inc(p);
...
end;
```

Performance of large integer types

If you need to work with integers larger than can fit in longint, you have a couple of options (`int64`, `comp`, `double`, and `extended`). Three of these types are actually floating point types. Consequently they are not completely

interchangeable. Only the relatively new `int64` is completely handled as an integer. `comp` is sort of a hybrid in that it is stored as an 8 byte integer (the same as `int64`) but all operations are performed as floating point. Borland has officially designated `comp` as obsolete, and instead favors `int64`. However, as can be seen in the following table, `comp` enjoys a substantial performance lead in some circumstances. `Extended` and `double` can also be used to operate on large intergers although care must be taken to ensure that the lack of periodic rounding doesn't accumulate to the point of changing the answer. Shown below are the measured CPU Pentium II cycles for each operation with random values in the range (0 < x < 2^63). "Ovhd" refers to the overhed associated with making a function call and assignment with this type. `LongInt` is included for comparison purposes only.

	ovhd	add	mult	div
Longint	2	1	1	4.7
Comp	40	4.3	4.4	34
int64	19	2.6	26.2	804
double	25	3.1	1.3	35.8
extended	43	4.1	3.2	34.4

Note: that the out-of-order execution capabilities of the Pentium II make precise timing measurements nearly impossible for individual operations. Consequently, the cycle counts shown above should only be considered approximate.

Aside from the horrid division performance for `int64` there is no obvious best choice. Of the three Floating point based types `double` is best but it actually has slightly fewer digits (15 vs 19). `int64` is better for addition than `comp` but worse otherwise.

So what to do. Well the best answer is to blend a bit. Use `int64` as the base type. Division is easily handled by using `trunc(Int64A/Int64B)` instead of `Int64A div Int64B`. Getting the best performance is somewhat more complicated. Since `comp` and `int64` have the same format, converting between formats is free. Using this you can force what would have been an integer based multiplication to a floating point one. This is shown below:

```
var
  A,B,C,D:int64;
  CA:comp absolute A;
  CC:comp absolute C;
begin
      // Result:=A*B*C*D;  // Original expression
      Result:=round(CA*B*C*D);  // blended version
    end;
```

The usage of `CA` above forces the calculation to be done in floating point. Note that only one floating point type is needed to force the entire term into floating point. However, if there are multiple terms they each need a floating point variable: `Result:=round(CA*B+CC*D)`. Also note that `round` is used instead of `trunc` as was used in division. while it would be possible to convert back into integer within an expression, it will typically not result in a speed increase unless two or three additions can be done in for each `round`.

String Optimization Guidelines

Contents

Style Guidelines

Use Hyperstring instead of "rolling your own"

There is no point in reinventing the wheel. The HyperString freeware library addresses the shortcomings and inefficiencies of the native AnsiString function set included with Delphi. If you are basically doing "normal" sorts of string operations but need more speed you should start here first.

Do not double-initialize strings

The default string type, AnsiString, is automatically initialized to be empty upon creation. Consequently, there is no need to initialize it a second time. For instance the code `s := '';` is redundant below:

```
procedure GreatestOnEarth;
var
  S: string; // a long string, not short!
begin
  S := '';
  ...
end;
```

Note that this does not extend to functions that return a string since the behavior of the `result` variable in this case is better characterized as a passed `var` parameter than a local variable.

Use `SetLength` to preallocate longstrings (AnsiStrings) whereever possible.

Dynamic allocation makes AnsiStrings very powerful. Unfortunately, it is quite easy to abuse this power. A typical situation looks something like this:

```
S2 := '';
for I := 2 to length(S1) do
  S2 := S2 + S1[I];
```

Ignoring the fact that `Delete` could be used for this, the problem here is that memory for the `S2` string may need to be re-allocated repeatedly inside the loop. This takes time. A simple and potentially much more efficient alternative is this:

```
setlength(S2, length(S1) - 1);
for I := 2 to length(S1) do
  S2[I-1] := S1[I];
```

Here, memory for `S2` is allocated only once, prior to the loop.

This sort of "memory manager abuse" is common with AnsiStrings only because re-allocation is automatic and thus easily ignored. With `PChar` and manual allocation, the programmer is made painfully aware of the problem with this coding style. The older Pascal style strings avoided this problem entirely by using static allocation.

Thread safety of strings and dynamic arrays - Applies to: Version 5+ and CPU's before Pentium III and Athlon

The thread safety of strings and dynamic arrays has been improved by preventing reference count problems. Previously, Reference counts were read altered then saved resulting in the potential for another reference on another thread to read or write in between those operations. This has been fixed by directly altering the reference count and locking that single instruction to prevent preemption. Everything has a price unfortunately. The `lock` CPU

instruction prefix used to achieve this thread safety is quite expensive on Pentium II processors. My measure of the effect of this change is an additional 28 cycles per refcount adjustment, which in a worst case scenario can result in a 2x *decrease* in performance. Real world reports have placed the impact in the 0 to 20% range.

Reverting back to version 4 longstring behavior

It is possible to undo the change to longstring behavior described above. You can even make longstrings even faster before. To do this you need to make some changes in system.pas and recompile it.

The easiest way to recompile system.pas is to use the make utility and the makefile located in the /source/Rtl directory. *Copy the source to a new directory!* You don't want to replace the originals. The make also expects certain other subdirectoies such as lib and bin to be present. Make sure your new location has these as well. Also you will need TASM as there are many external asm files that need compiling as well.

The changes that need to be made are fairly simple. First, you need to get rid of all the `lock` prefixes. I prefer to do a global replace of 'lock' with '{lock}'. This will return strings and dynamic arrays to the Pre Version 5 performance levels. To go beyond that you need to eliminate two `xchg` instructions. These instructions have implicit `lock`prefixes. The original code is shown below:

```
procedure _LStrAsg{var dest: AnsiString; source: AnsiString};
...
@@2:    XCHG    EDX,[EAX]
...

procedure       _LStrLAsg{var dest: AnsiString; source: AnsiString};
...
        XCHG    EDX,[EAX]                       { fetch str                   }
...
```

In both cases you can replace the XCHG instruction by using three move instructions and ecx as a temp register:

```
procedure _LStrAsg{var dest: AnsiString; source: AnsiString};
...
@@2: {   XCHG    EDX,[EAX]}
        mov ecx,[eax]
                mov [eax],edx
                mov edx,ecx
...

procedure       _LStrLAsg{var dest: AnsiString; source: AnsiString};
...
        {XCHG    EDX,[EAX]}                       { fetch str                   }
        mov ecx,[eax]
                mov [eax],edx
                mov edx,ecx
...
```

The above changes will result in string assignments executing about 6 times faster than they do in Version 5. (2 times faster than Version 4).

Avoid using ShortStrings - Applies to: Version 5+

Presumably in an effort to phase out all the old shortstring methods and maintain only one set of string routines, shortstrings are converted to longstrings prior to many manipulations. This effectively makes these shortstring operations much slower.

Avoid using `Copy` to create dynamic string temporaries.

This also relates to memory manager abuse. A typical situation looks something like this:

```
if Copy(S1,23,64) = Copy(S2,15,64) then
  ...
```

Once again, the problem here is memory allocation for the string temporaries which takes time. It is unfortunate but the native AnsiString functions offer little alternative other than something like this:

```
I:=1;
Flag := False;
```

```
repeat
  Flag := S1[I+22] <> S2[I+14];
  Inc(I);
until Flag or (I>64);
if Not Flag then
  ...
```

Use longstrings (AnsiString) exclusively and cast to `PChar` when necessary.

Popular myth has it that AnsiString is somehow inherently less efficient. This stems from poor coding practices, memory manager abuse and lack of native support functions as described above. Once an AnsiString has been dynamically allocated, it is just like any other string; a linear series of bytes in memory, and no more or less efficient. With adequate support functions and proper coding, the performance difference with AnsiString is negligible.

Prefer `Delete` over `Copy` to remove from the end of the string

`copy` will always copy the entire string. However, `delete` will just cut off the end of the current one.

Change: `AString :=copy(AString, 1, length(AString)-10);`
To: `Delete(AString, length(AString)-10, 10);`

Concatenating Strings

The best way to concatenate strings is also the simplest. `s1:=s2+s3+s4;` will produce the best results regardless of the number of strings or whether they are compile-time constants or not. Note: In D2 when combining compile-time constants the `s1:=Format([%s%s],s2,s3)` approach may be faster.

Casting to `PChar`

Essentially there are 3 ways to convert a string to a pchar: typecast as pchar, take the address of the first character, and typecast the string to a generic pointer. Each of these does different things. Taking the address of the first character (i.e. `p:=@s[1];`) will force a call to UniqueString to ensure that the pchar returned points to a unique string only referenced by s in the above example. Typecasting a string to a PChar returns the address of the first character or if the string was empty it returns the address of a null. Thus the pchar is guarenteed to be non-nil. The simplest is casting as a generic pointer (i.e. `p:=pointer(s);`). This is also the fastest, as there is no hidden function call.

Floating Point Optimization Guidelines

Style Guidelines

Do not use `extended` unless absolutely necessary

While the FPU performs calculations internally in extended (80 bit) precision, it does not load and store in this format very efficiently. Consequently, using the `extended` type can double the overall execution time of the simpler arithmetic operations (+,-,*). This is not due to additional time for actually performing the operation, but rather due to the extra time needed to load and store these values. Additionally, the size of `extended` type variables is awkward (10 bytes, 12 bytes with doubleword alignment), leading to in increased likelihood for the variable to straddle a cache line which causes a performance loss. Finally, in compiler versions 2 through 4 local `extended` type variables are aligned in the *last* 10 bytes of the 12 bytes (3 dwords) allocated for them, instead of the first 10. This means that they *always* are misaligned as local variables. This has been fixed in Version 5 and does not apply to compiler generated temporary variables in any version.

Avoid mixing floating point types

The basic problem is that you will force an unnecessary type "conversion" step in two cases: 1) assigning one variable to another, and 2) Passing a variable as a parameter. In these, two instances a variable will have to be loaded on to the FP stack and saved as the new type, rather than simply copied. This can take 3 or 4 times a long.

Strive to have one function call in each assignment expression

The floating point unit's register stack is only eight entries deep. Consequently, to prevent the stack from overflowing, function calls from within an expression require that the register stack be unloaded prior to making the call. The only exception is that the first function call in an expression is free from this unloading because it can be called just after its arguments are determined, but before the rest of expression is evaluated. Delphi unloads the stack by saving any stored values to temporary (and invisible) `extended` variables. As was already noted, `extended` is bad, so you should make your own temporary variables and break up expressions so that only one function call is made per variable assignment. This rule also covers compiler "magic" functions found in the SYSTEM unit, like `Abs` and `Sqr`. It does not include "nested" calls. That is, function calls contained in the parameter expression another function call. Since floating point parameters are always passed on the stack each parameter expression represents a separate expression.

Floating point constants

Floating point constants must be saved in the executable as a specific type (i.e. single, double or extended). Basically, whole number constants are saved as single and fractional numbers are saved as extended. As already mentioned, using `extended` incurs a high cost, so you should force the constants to be of a given size (`single` or `double`) by making them *typed* constants. Note that this does not increase the overall executable size since the value had to be included in the binary anyway. For example:

```
const
  e: Double = 2.71828; // Euler constant
begin
  ...
  SomeVariable := e*sqr(r);
```

```
...
```

will be both faster (use of `double`) and smaller (`double` only requiring 8 bytes) than the equivalent routine using the `extended` type. Note, though, that a typed "constant" can be written to with the $J+ directive.

Also, the compiler will combine constants at compile time if possible. If the operation between two constants has a higher precedence than any operation involving those constants and any variable or variable expressions then the constants will be "folded" together. Additionally, in Delphi 2 and 3 division by a constant would always be converted into multiplication by its reciprocal. Unfortunately, this was eliminated in version 4. So as an example, in D2 and D3 the statement:

```
fp:=fp*3*4/5+3*4/2;
```

will actually be calculated as:

```
fp:=fp*3*4*0.2+6
```

In D4 the same statement will actually be calculated as:

```
fp:=fp*3*4/5+6
```

You can get better constant folding by placing the constants in front of any variables:

```
fp:=3*4/5*fp+3*4/2;
```

Will actually be calculated as:

```
fp:=2.4*fp+6
```

Set the control word precision to the appropriate level

Floating point division and square root instructions can take a substantial amount of time. However, you can save some of that time if you do not need *maximum* accuracy. You can modify the level of accuracy by changing the FPU's control word. The default accuracy, as initialized by the Delphi runtime library, is the slowest, but most precise one (i.e. `extended`). Delphi supports direct modification of the FPU's control word with the `Set8087CW` procedure and the global variable `Default8087CW`. Use the following lines to set the control word to different precision levels:

```
Single:   Set8087CW(Default8087CW and $FCFF);
Double:   Set8087CW((Default8087CW and $FCFF) or $0200);
Extended: Set8087CW(Default8087CW or $0300);
```

Note that changing this control word only changes the execution time of division and, in the case of Pentium II and Pentium III processors, square roots.

As of version 6 this has gotten easier as you can simply call the `SetPrecisionMode()` with the proper precision level constant (`pmSingle`,`pmDouble`, or `pmExtended`).

Prefer `Round` over `Trunc`

`Trunc` reads and sets the FPU control word, which is very costly. The `Round` function, on the other hand, does not do this and therefore is about 2.5 times faster on a Pentium II.

Favor procedures with `var` parameters over functions

This is an overhead management issue and hence comes into play more with small functions where overhead is a greater percentage of the total processing time. For example changing:

```
function Calc(a: Double): Double;
begin
  result := a*1.1;
end;
```

to:

```
procedure Calc(var Result; a: Double);
```

```
begin
  Result := a*1.1;
end;
```

cuts execution time in half (on a Pentium II). This is especially true if you are mainly passing a value around (including simple assignment) rather than actually using it. For instance:

```
function SetValue(NewValue: Double): Double;
begin
  Result := Value;
  Value := NewValue;
end;
```

results in a function composed almost entirely of overhead.

The downside of this technique is that you need to use `var` instead of `const` for parameters that are not supposed to change, because `const` does not really do anything on floating point parameters except to force a compile-time check that the parameter indeed is not changed.

Trapping Floating Point Exceptions

FP exceptions (such as divide by zero) aren't actually triggered when the error occurs. Instead they are delayed until the next floating point instruction. Presumably this implementation was used to allow for testing and handling of the error locally. However, it can have the rather odd effect of making the wrong code look guilty if and when the exception is finally triggered. The solution to this is to stick a `wait` or `FWait` instruction in to force the exception. This just what the compiler does after each and every floating point statement. Of course executing all those waits can be costly, so in a hand written floating point assembly routine you may want to simply stick one in right at the end of the routine once you have it debugged. This keeps the cost low, but still ensures that any exception generated still points to at least the proper routine.

Of course, every rule needs an exception. One example where this is not the case is Windows 95 (!) and this code (From Stefan Hoffmeister's FPU Demo):

```
x := -1;
asm
    fld x
    // Generate an IEEE invalid operation:
    //   sqrt(-1)
    fsqrt


    fwait
end;
```

Under NT, this (correctly) raises an FP exception. Not so on Win95. Jam in an FXAM before the FWAIT in Win95 - and get the exception. Thank you, Microsoft.

Optimization Techniques

You need to do your own floating point optimization

Delphi does no floating point optimization. You are going to get exactly what you ask for. Thus, do not assume things like common expressions are going to be combined. You need to do all this yourself.

Make great effort to reduce the number of divisions

Division is very expensive, taking about 20-40 times as long as multiplication, addition, or subtraction. Move divisions outside of loops whenever possible. Do not forget to convert a division by a constant into the corresponding multiplication with its reciprocal.

How to avoid floating point checks for zero

Under certain circumstances it can be beneficial avoid a direct comparison to check for a zero in a floating point variable and instead utilize typecasting to test the underlying representation of the variable. This is because floating point comparisons require a true floating point based

zero check by taking advantage of the way zero is stored. Considering substantially reduced readability of this technique it should be used sparingly.

To check a `single` variable for zero use: `DWord(Pointer(SomeSingleVar)) shl 1 = 0`

Checking a `double` variable is more complicated:

```
type
  PDoubleData=^TDoubleData
  TDoubleData=record lo,hi:DWord end;

// two possible ways

var
  DoubleData:PDoubleData;
...
  DoubleData:=@SomeDoubleVar;
  if (DoubleData.hi shl 1 ) + DoubleData.Lo = 0 then
...

// or

var
  DoubleData:TDoubleData absolute SomeDoubleVar;
...
  if (DoubleData.hi shl 1 ) + DoubleData.Lo = 0 then
...
```

The above techniques can shave about 30-40% off the comparison time on a Pentium II.