**Considerate Programming Reducing the Maintenance Costs of Commercial Quality Code.**

**Complexity of Comprehension**

Do not add unnecessarily to the amount of information the reader needs for understanding a piece of code. Be considerate of the reader! Except for loops, the progression of execution should be generally forward in the code. Hopscotching back and forth complicates understanding. Rather than putting an error handler near the end of a subroutine, the programmer has placed it at the point of first usage. Subsequent users of that handler have to branch back to get to it. In large routines, this technique, used many times, can lead to convoluted and confusing code. A better choice would be to gather all error handlers towards the bottom of the routine.

**Performance Coding**

Is coding for performance really that good of an idea? It depends. Unfortunately, on current machines, performance and clarity have come into conflict like never before. With pipelining, instructions can be processed much faster than registers can be loaded or copied. Consequently, a delay needs to occur between the loading of a register and the use of that register. So in order to achieve the most efficient use of machine cycles, useful but unrelated instructions need to be inserted, if possible, so as to space out the loading of and the usage of a register.

So the question arises, what's more important, Clarity or efficiency? Again, it depends. The first question to ask: Is your code compute bound or I/O bound. If compute bound, then efficiency probably does matter more than clarity. If I/O bound, then it probably does not.

The next question to ask: How much real world time is a particular code change going to save? Are you pouring out sweat and working hours and hours to squeeze performance out of initialization code? Is your successor pouring out sweat and working hours and hours to understand that code? Congratulations, you've just saved about a microsecond per year of processing time.

Other questions: How long does it take your program to run? How often is it going to be run? Are there real world dollars involved? If your program is going to be executed once a day, and it only takes 10 seconds to execute, what's the point in making it run in 2 seconds? (Real dollars? Well, then maybe.)

If you're writing code that's in critical loops in a lengthy compute bound process, or if you're writing macros that are going to be used to an unknown degree in unforseen programs, then sure, coding for efficiency is probably the overriding aesthetic. Otherwise, the development and maintenance costs probably aren't worth it. You have to use good judgement.

Also, keep in mind that the paradigm of efficiency has changed considerably over the years, and probably will change again. For example: Once upon a time, efficient dataset I/O was important. But then one vendor wrote chained channel programs for tape access that were so efficient, that they monopolized the channels and locked out all other devices. I suppose they were very proud of how “good” their channel programs were until their customers forced them to throttle back. ... And now, there's caching. That goes a long way towards reducing the importance of efficient user program coding.

Remember, technology changes. How long will it be before pipelining techniques change or are replaced by something new and not yet thought of by you and me?

**Sanity Checks**

A sanity check is any programmatic action, taken at assembly time or at execution time, that proves an assertion relevant to the structure of the program, the logic of the program, the intent of the code, or the current environment. Sanity Checks are a very important part of documentation. When troubleshooting, the presence of sanity checks can save a lot of time and effort by guiding the programmer away from irrelevant inquiries.

**Assembly Time Sanity Checks**

At Assembly time, a sanity check is any usage that leads to a syntax error, should a required condition not be met. A lot of sanity checks can be accomplished simply by proper usage of symbols.

But there are many situations that arise in which the assembler has no direct ability to detect a problem. For example, if you're constructing a message into a buffer, there is a need to insure that the message does not overflow the buffer. If the message is built from a series of constants and fixed length elements, then there is no easy way for the assembler to directly check for overflow.

However, there is an easy way to create an assembler check that is sensitive to the overflow. Simply use the length of the message and the length of the buffer to concoct a constant whose value will be syntactically invalid if (and only if) the message is too long. If you make it a zero-length constant, then it can even be placed inline at the point in the code where the tested condition actually matters.

**Execution Time Sanity Checks**

At execution time, sanity checks can range from being corrective error handlers to abortive program check traps. Corrective error handlers, of course, generally are pretty complex, and certainly must be written for any error condition that has any likelihood of occurring. Abortive traps, on the other hand, are trivially easy to construct, but should only be used for conditions that you truly believe should never occur. (Of course, you may be wrong, in which case you're going to be real glad that you had the trap!)

Typically, an abortive trap is an instruction that will cause an immediate program check in the event that some condition occurs that really really shouldn't occur. The condition is considered either to be impossible or to be so unlikely that the effort of writing a handler is not worth taking. Such traps also can be used to guard against future code changes that might be incompatible with current logic.

Macro to generate conditional, abortive traps that can have explanatory messages associated with them. The macro is named #DIE, and it generates what I call a “DEAD-trap”. Depending upon the operands given, the DEAD-trap can be generated either in-line (with a conditional skip around it) or out-of-line as a literal operand of the generated branching instruction. If you want the DEAD-trap to include a special message, then you can provide that message, in a quoted string, as the macro's last operand; otherwise, a default message will be generated that will be a unique 4-digit number.

Its operands and usage are fully documented by internal commentary. Abortive traps are wonderfully useful devices. Here are some examples of the situations in which they can be used.

DEAD-Trap Sanity Check Example: Recursion Underflow Trap I'll use DEAD-traps to catch illogical conditions early before they have a chance to wreck havoc or to hide themselves behind million-instruction histories. Stack underflow is an example of an illogical condition. If a program is written correctly, then underflow should NEVER occur. Nonetheless, it is both a safe and comforting thing to make sure. A DEAD-trap comes in handy here.

**Summing Up**

Documentation. Symbol usage. Sanity checks. Clarity of thought and style. It may take extra development time, but if you can afford the time, the long-term costs will be reduced, and the life and usefulness of the product will be extended. Paranoia! In programming, that's a good thing.

routine – последовательность команд

error handler – обработчик ошибок

pipelining – конвейерная обработка

channel program - программа канала (управляет работой канала ввода-вывода)

sanity check – контроль работоспособности, корректности программы (на уровне здравого смысла)

assembly time – время ассемблирования (программы) (Lingvo Computer)

underflow – исчезновение (значащих) разрядов, потеря значимости; опустошение (напр., буфера данных при обмене)