TO: DECWEST

FROM: Don MacLaren -- 29-Mar-1985

SUBJECT: Systems Programming Language

This is another try at stimulating some discussion of the Systems Programming Language Question. As a starting point, I provide my some of my own opinions about functional advantages and disadvantages of VAXELN Pascal. Note that these opinions are uncorrupted by personal experience with the language -- I did all my VAXELN work in PL/I and MACRO.

1 WHY THE QUESTION?

If we are going to do the system software for a new architecture, we will, deliberately or accidentally, design a systems programming language for it. If the system designers leave this to chance, odd things can happen. I can argue the latter point at length. Maybe it's enough to examine what happened with VAX/VMS. The VMS systems programming language turned out to be the union of MACRO, BLISS, and SDL. The official edict was to use BLISS if feasible. The system designers used Macro, and this was also the publication language, e.g. in the System Services manual. At one time (and maybe yet) the best definition of most RMS features was in the PL/I User's Guide, PL/I being one of the several high-level languages customers prefer to use for systems programming.

Starting soon to think about the language will increase its final quality, even if the language is only a minor variation on an existing language. We should determine what language capabilities are desirable and then choose or design the language -- subject of course to the constraints of schedule and compiler practicality.

Our experience with VAXELN shows that expressing the system in the language from the beginning can improve both.

2 STRENGTHS OF VAXELN PASCAL

As the systems programming language for VAXELN, the most striking feature of EPascal is its integration with the system. It may not be possible to acheive quite this this effect in the more general context of a complete operating system, but it's certainly worth trying.

2.1 Completeness

Everything can be done in the language (well, almost); MACRO is not needed.

2.2 Types And Type Checking

This is the central feature of Pascal. EPascal also applies the notion of compile-time checking to some tricky systems programming things, e.g. what's allowed in an interrupt service routine.

2.3 Flexible Types And Dynamically Sized Data Items

2.4 Strings In The Language

2.5 Type Escapes

These features, especially type casting, lack elegance but they are important for two reasons.

- Systems programming at times requires redescribing data, e.g. to do pointer arithmetic or to get at the parts of a floating point number.
- The language is open ended in regards data structures. One can manipulate data whose structure can't be described within the language's type structure.

2.6 Inline Routines

2.7 Argument List Notation

The capabilities for keyword notation, optional arguments, and variable-length argument lists seem especially significant for the system services typically found in operating systems.

2.8 Modules

There are some questionable details in the EPascal treatment of modules. However the language does provide an explicit form of module that blends with the system treatment

- of source files and separate compilation
- of object modules and linking
- of debugging

3 WEAKNESSES OF VAXELN PASCAL

Considered as a language for a new architecture and operating system, VAXELN Pascal has some functional weaknesses.

3.1 Missing Data Types And Instructions

EPascal is complete for VAXELN, but it doesn't support all VAX data types (e.g. decimal) and all useful instructions, e.g. EMUL.

How EPascal relates to the new architecture's instruction set remains to be seen.

3.2 I/O

Pascal's treatment of files is unrelated to the system's treatment of files (any system). The result is obscurity, inefficiency, and runtime library code that's irrelevant for systems programming.

The text i/o capabilities are primitive.

3.3 Inter-Language Data Structure Definition

Whatever language we choose for the system, many customers will program in one or more other languages. A functional equivalent of SDL is needed. Shouldn't this be part of the systems language?

This is more a question of compiler capability than language. A variant of EPascal front end could be the shell for an SDL-like utility. However the language capabilities should support this usage, and this at least requires review of the language. For example, structures likely to be acessed by other languages should be simple and follow appropriate naming conventions. Does the language help with this?

3.4 Foreign Routine Interfaces

The EPascal features for specifying parameters and calling conventions don't encompass all the conventions used in other VAX languages, e.g. descriptors as used in the VMS languages. A more comprehensive treatment is important for the new system, especially if we accept the idea that the systems language encompasses the SDL function.

This point depends in part on the assumption that many customers will work with multiple languages -- at least the systems language plus their own favorite. In this situation they always end up having to write some routines to brdige the gaps.

Note the implication that routines violating the system's conventions will be written in the systems language. A compiler option can produce discoraging messages for these.

3.5 The Mysterious Linker

Although EPascal was intended to be complete for creating a program (as opposed to building a whole system), the capabilities of the linker aren't reflected in EPascal. Things like shareable images are hard to explain in EPascal terms.

3.6 Inefficient Constructions

There are rough spots in EPascal that promote the generation of inefficient code, e.g.

- value parameters
- sets
- the way functions specify the returned value

In comparison with MACRO, EPascal and other high level languages lose a lot in cases where two or more distinct data nodes are being manipulated, e.g. when setting bits in one node while testing the contents of another. The problem is that the compiler can't tell that the nodes are distinct. Each assignment goes through to storage, even if several bits are being set in the same word. This would look even worse on a RISC machine.

Problems like this are inevitable when using a high-level language, but why accept future code inefficiency that can be avoided by design effort now?