La résistance

In 1992, the power of C++ dominated the known programming world. All the world? Not quite. For in a small Gallic village... Willie Watts talks to chief Eiffel Druid Bertrand Meyer.

Can you give us a sketch history of yourself and Interactive Software Engineering?

I am originally a pure product of the best that the French system has to offer in terms of general, scientific and engineering education. This was completed by an MS in computer science at Stanford in the US.

At Stanford I didn't have enough time to learn much in detail, but I learned what was important and what was not. So that after that I was able to read a lot and learn by myself. I was already working in industry, but at least what I learnt what was important from Knuth, McCarthy and a few people like that. I also learnt about object-oriented technology at Stanford. I was fortunate enough to run into a description of Simula and it struck me right away as the way to program.

After that I came back to France, and went to work for a company called 'EDF', which is the equivalent of what the Central Electricity Generating Board used to be in Britain. I was head of a group which was in charge of software engineering, programming education, software engineering tools, libraries and also links with research in software engineering.

I had a kind of dual role - partly operational (very operational, actually), but also at the same time research. The non-academic side involved, among other things, setting up a training programme on modern software methodology, which opened my eyes to the reality of industrial software development. At the same time I was quite active in research in programming methodology. I wrote a book called *Méthodes de Programmation*, ie 'Programming Methods', which was very important for me and was also quite influential in France. It was used in the main educational institutions, and it put me in contact with professors.

Bertrand Meyer and Eiffel

With the current plethora of object-oriented languages and tools, and C++ apparently set to conquer the Universe, the reader might reasonably ask why *.EXE* chooses to devote so much space to the inventor of yet another language - a language which is still comparatively unknown in the wider programming community.

The answer, in a nutshell, is Meyer's book *Object-Oriented Software Construction* (pub. Prentice-Hall, ISBN 0-13-629049-3). This book, I feel I can state with little danger of effective contradiction, contains the most lucid description of object-oriented techniques that you will find; whatever their opinion of the Eiffel language, few of Meyer's opponents will deny that he can write like an angel.

The book has acted as a superb advertisement for Eiffel. Interest in the language has spread from the academic world, and is now beginning to appear in the commercial mainstream. At the same time, control of Eiffel has been passed from Meyer's company Interactive Software Engineering to a non-profit consortium, and alternative (non-ISE) implementations of the language have begun to appear, notably SIG's implementation for DOS. Eiffel is fast becoming a practical possibility for ordinary projects.

Eiffel is different. It is not just another Algol variant with a few bolted-on object extensions. Programmers who track current thinking should at least to know about the ideas in Eiffel; this interview may serve as a taster for further reading. ***WRW.***

Santa Barbara

I had this dual career for about eight or nine years. In 1983, I decided to develop the academic side of me a little further, so I took what I thought would be a year's sabbatical at the University of California Santa Barbara (in the end I stayed four years). There I taught much of the basic software curriculum.

Now, in the data structures and algorithms course that I taught it was the department policy that the professor should use C. This was strange, because most of the other courses enjoyed a lot of academic freedom. But C was a particular requirement because they were using this course for two purposes: 1) the official one - to teach data structures and algorithms - and 2) to serve as a filter to separate the men from the boys.

You're saying the course was made deliberately difficult?

Oh yes. Deliberately puzzling. I hated it. I should say that I didn't know C very well at the time, so I had to learn it - and it was horrible. I wanted to teach data structures, algorithms, and systematic approaches to programming problems. Instead of that, I found myself trying to help students debug programs using pointers, finding incorrect memory references and so on. I decided that I could never do that again.

So in '85 I started Interactive Software Engineering with a few colleagues in order to follow up certain aspects of our research. The work in the company soon became more interesting, from a purely scientific viewpoint, than the work we were doing in the University, because I could use whatever I found scientifically valuable as opposed to bowing to the student pressures to use C.

When did Eiffel appear?

When we started the company we really wanted to develop some CASE tools - in particular a structural editor, now the Archi-Text product. But I also wanted to use proper techniques - I didn't want to be too far from what I was trying to teach. So we looked around for a decent object-oriented environment and we just couldn't find one. Smalltalk was attractive, but it was too far from the mainstream. C++ was available (as was Objective C) but we really didn't discuss it. It was not up to our standards.

I very quickly wrote a specification for a modern version of Simula, with some simplifications and some extensions. I also a wrote a basic version of the libraries. Libraries are the key to the success of object-oriented technology in Eiffel. If it can be defined by just one sentence, Eiffel is a language to write the Eiffel libraries - that is to say to write the best possible, reusable industrial-quality software components that we could think of.

I did that in just a week or so, and that was it. At first, we didn't really think about it as being a tool for others. It was only months later that we realised that we had something that no-one else really had: a complete, statically-typed, efficiently implemented language that could actually provide an object-oriented solution to mainstream industry. It's only then that we started thinking about marketing the product.

How was Eiffel first implemented? Was it one of those languages which was always implemented in itself?

No. We didn't have a bootstrap strategy initially, which may have been a mistake. We just used the obvious solution, that is to say to write it in C. And I should say we were sorry for that decision for a long time. It is only in Version 3 that we have managed to write the entire system in Eiffel. With our portability requirements, C was the almost inevitable choice.

What's in a name?

Why is it called 'Eiffel'?

The question should be, 'How could it be called anything else?' It was almost inevitable. First there's this habit of naming languages after people: Pascal, Turing and many others. We decided, for once, to take not a scientist but an engineer, and a really great engineer. Second reason: the method promotes bottom-up software construction. If you think of the shape of the Eiffel Tower, and you try to build it top-down... Third, and probably the important: if you look at that structure, it's an extremely elegant, powerful, reusable, extensible structure. However it is built of a few small, simple parts which you combine, which is exactly the same idea as object-oriented software construction.

The opposition

How would you characterise the difference between Eiffel and C++?

There is a difference in philosophy and there are technical differences. The difference in philosophy comes from a different view of what 'compatibility' means. I don't think there is any disagreement with respect to the necessity of reusing existing software, particularly C software. The divergence resides in how it should be done.

The C++ (or Objective C, or Turbo Pascal) approach is that you should have compatibility at the language level. So you take C and you add things to it. The Eiffel view is that compatibility with existing software is not an excuse for polluting the language. The language can be C, which is a certain technology, or it can be object-oriented, which is completely different.

You don't make a device that is both a diode and a transistor. If you have an existing machine that uses diodes then, fine - you want to keep it. You could put in wires to the transistors and have communication links and so on, but you don't try to transform a diode into a transistor incrementally. This is the Eiffel position. You choose between the Eiffel, object-oriented world and the C world, but you should keep the two separate; because otherwise you risk losing on both counts.

You lose the simplicity of C: you lose the ability to implement it efficiently, the ease of writing it and the ease of teaching the language, which are the three major advantages of C. On the object-oriented side, you again lose a lot. You lose simplicity - because the object-oriented paradigm has to be combined with totally incompatible concepts. You lose the ease of teaching the object-oriented paradigm. And you lose some of the most important advantages of object-orientedness. For example, it is quite impossible to have garbage collection in C. You lose typing - you cannot have both the C type system and an object-oriented type system. You lose the ability to do things like exception handling properly, as you must take into account all kinds of bizarre side-effects.

So you actually lose the most important benefits, the real breakthroughs, the quantum leaps that you can get from object-oriented technology, and all because of this stupid requirement of remaining compatible with something that has nothing to do with object-oriented technology. The technology is too good, too important, too potentially beneficial to damage it because of concerns that may appear valid in 1986, or 1989, or 1992, but will totally disappear from the scene if the technology becomes successful.

As for technical differences, there's a whole list. Type-checking; assertions; genericity (particularly constrained genericity, which is the only way to get safe genericity); exception handling (which has been proposed for C++, but Eiffel has them now),

the assignment attempt (the ability to force a type on a variable) which is absolutely essential; and there are the persistence facilities. Going a little bit beyond the language, the presence of standard libraries is, I think, a really strong plus for the Eiffel approach. In C++ there are all kinds of competing libraries but nothing has emerged as a real standard, in part because the language does not support the tools (such as genericity) for building libraries.

Template faults

With regard to genericity, what is wrong with templates as implemented in C++?

To start off with, templates are only now getting into the language. But they are only an emulation of genericity. In particular they are not closely connected to the type system, and there is no support for constrained genericity. It is very important to be able to accept actual generic parameters only if they are descendants of a certain class. For example: I want to have vectors of something of type `T`, and I want to be able to add two vectors, so objects of type `T` must have the + operation applicable to them. It means you can then have a vector of integers, but you cannot have a vector of nuclear submarines if a nuclear submarine doesn't have a + operation.

Templates cannot support this because they are just a kind of macro. I see templates as making more official what people had been doing manually in C++ so far, which is using the pre-processor to generate variations of a class.

But templates do give you some type protection... You're saying that's not enough?

Yes, I am. Type protection in C++ is in any case always problematical. As long as you can have those casts, as long as you can convert from any pointer type to any other pointer type, what type we talk about becomes pretty meaningless.

In a recent interview with .EXE, Bjarne Stroustrup said that he didn't think that it was necessary to add Eiffel-style support for assertions to C++. He thought it sufficient that one could acquire add-on specialist tools to do the job, and that he did not believe in cramming too many features into a language. Is that not a fair comment: if you need assertions, you should buy a separate tool?

The comment 'I don't believe in cramming too many features into a language' is a fair comment. I would say that about C++. I don't think it is proper to have, for example, function pointers and dynamic binding in the same language, because it is confusing. If you don't believe in crowding too many features into a language, then you wouldn't produce C++, because C++ is exactly that. It's taking C, which already had its share of language features, and adding more, including some which are redundant and incompatible.

This business of function pointers is typical. If you use an object-oriented language like Smalltalk or Eiffel, then to obtain automatic selection from various operations at run-time you use dynamic binding. If you use C, you can emulate this in a rather unpleasant low-level fashion by having arrays (or other data structures) of function pointers. Now this is another way of doing things; it's less nice, but it works. What I don't think is proper is the C++ approach in which you have both mechanisms. Programmers have to choose all the time between the one way of doing things and the other, which means a lot of confusion and complexity.

I decided that I should learn up about C++ after all, so I went and read the Ellis and Stroustrup book *(C++ Annotated Reference Manual)* from cover to cover. I was horrified to see how many criticisms there are in that book of C and C++. There are comments like, 'the array facility of C, and hence of C++, is brain-damaged.' You read this and you say, 'What? You're designing the language, and now your telling us that something is still wrong?'

There are comments like this on many other aspects. There are comments like, 'This is available, but don't use it.' I don't think that is good language design. If you produce a language design book, you should be proud of it and there shouldn't be any dark corners. I think I can say that about Eiffel. I'm not saying that Eiffel is perfect, but I cannot point to any construct in Eiffel for which there is a comment of the form 'Don't use it'.

Returning to your question: I don't think that assertions should be separate from the language. Assertions are absolutely essential to object-oriented design. This was something which was mentioned in *Object-oriented Software Construction*, but clearly

A Few Eiffel Terms

Assertions - C programmers will be familiar with the macros in ASSERT.H, which allow the programmer to assert that a condition is true. If it is not, the program aborts with an error message. An example use is to protect a routine that calculates square roots of positive reals from being called with a negative parameter.

Although Eiffel's assertions are superficially similar, they are fully integrated into the language (for example, they have their own set of inheritance rules) and are much more powerful. There are various kinds. *Preconditions* work like the square root example above: 'before you call this routine, *this* condition must be true.' *Postconditions* say what must have happened on the completion of a routine. A simple example is a routine which adds an item to a list: its postcondition might state (among other things) that the list is non-empty on return. *Invariants* are attached to classes rather than routines; these make statements about a class's data that are always true. For example, if the list class has a Boolean field `empty` and an integer field `count_of_items`, then a reasonable invariant is `empty = (count_of_items = 0)`.

As well as their debugging use, assertions are used in documentation of classes, and to enforce 'Design by Contract'.

Assignment attempt - this is Eiffel's mechanism for identifying object types at run-time. Using a special assignment symbol `?=`, the programmer is allowed to attempt to assign to an known object from one of unknown type (perhaps unknown because it has just been loaded from a database). If the type system prohibits the assignment, the target object becomes void; if it is allowable, a normal assignment takes place. This system fulfils the same purpose as Turbo Pascal's `TypeOf` function and C++'s proposed `ptr_cast()` and `ref_cast()` 'safe cast' extensions.

Genericity - Generic classes are 'typeless' container classes which are assigned a type when instantiated. For example, one could build an all-purpose `list` type, then use it to create a list of numbers, a list of windows, a list of addresses. Please see Meyer's comments in the text for an explanation of *constrained genericity.*

Type-checking - Eiffel is a very strongly typed language. In particular, *all* type-casting (which Meyer describes as 'a sordid back-alley deal') is forbidden.

not enough. I've written more about it since then. It's this whole idea of Design By Contract. That, for me, gave the theory behind object-oriented software construction. It's not just that you have classes and inheritance and so on; it's that you build software in such a way that it's made of pieces that communicate with each other on a basis of well-defined obligations and benefits.

I have tried to explain Design by Contract in a chapter with that title, part of a collective book that has just appeared, *Advances in Object-Oriented Software Engineering*. It's this idea that software is a combination of various pieces which communicate with each other not on the basis of pre-defined assumptions, but on the basis of proper and precise definitions of what each one of them expects from the other and must provide. This is what justifies the idea of preconditions, postconditions and invariants, and I don't know how to teach object-oriented programming without them. When I teach object-oriented techniques, I spend anywhere between one third and one half of the presentation on assertions - especially in connection with inheritance. I don't think anyone can understand inheritance properly without introducing assertions.

I also don't think you can understand the notion of class without the idea of the invariant, which expresses the integrity constraints on a certain data type independently of how the data type is implemented. This is where I disagree with Bjarne Stroustrup. I don't think assertions are 'fairly useful'.

Be assertive

Do you think people who use Eiffel always use the assertion mechanism?

People who use Eiffel well use them a lot. Even people who don't include assertions in their own software, because they haven't yet understood the power of this notion, benefit from them anyway.

The practice of software development in Eiffel is pretty much based on libraries. When you switch to Eiffel, you don't necessarily see as the major change a change in language or in method; what you see is a way to start working at a higher level of abstraction by using libraries. Now, these basic libraries are fully loaded with assertions. The documentation is essentially based on assertions, and their use is based mainly on assertions too. So even if somebody is only starting to work with Eiffel, and has is not yet putting assertions in his own software, he is going to benefit from them anyway.

It is true, however, as you implicitly suggested, that some people who start with Eiffel - especially if they come from something else, like C++ - don't necessarily put assertions to their full use. But they usually start using them after some experience.

Another point about assertions: when reading your book** Object-Oriented Software Construction **I found that it was not always obvious to me, as a programmer rather than a mathematician, why particular assertions were applied - particularly in the case of the invariant. Is it just that I am not smart enough, or have you found this to be a more general difficulty?

In my experience, and the experience of people working with me - both developers in our company and users - it is true that you don't necessarily get the invariant right first time. As you start improving a class, adding things to it, and understanding it better, this process is pretty much embodied in writing the invariant and improving the invariant. The more you understand what a class is

BRENT Communications and Telemetry Consultants

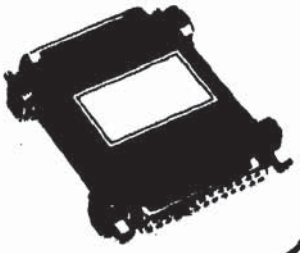
BEWARE THE PIRATE'S PATCH

You sell your software. You don't give it away. It needs the kind of protection that only a top quality UN-PATCHABLE dongle affords, but you don't want to pay the Earth for it, and you want to be sure that you'll not be making mistakes in incorporating it into your code.

The MAXPRO system is for you. There are microprocessor based units at realisitc prices which take care of complete .EXE files without access to source code. Set stop dates, tamper detection and many other facilities on a menu-driven front end. Encrypt in just moments. MAXPRO even copes with such as Clipper, QB & Clarion files with internal overlays. Neat trick.

For additional information contact us at

Brent Communications
Unit 2
Dragon Industrial Estate
Harrogate HG1 5DN
Tel: (0423) 566972
Fax: (0423) 501442



CIRCLE NO. 946

SOFTWARE SECURITY MODULES

Hardware devices (dongles) are a recognised and proven means of protecting software from unauthorised use and piracy.

Our range of devices offers some of the most robust and troublefree solutions around. All units are cascadeable and can be uniquely coded for each customer. Features include:

* *PC/non-PC.*
* *RS232/Printer.*
* *Internal memory (some units)*
* *Software drivers supplied.*
* *Minimum 2 year guarantee.*
* *From £14 to £50*

CTL

Control Telemetry of London
11 Canfield Place, London NW 3BT
Tel: 071 328 1155 Fax: 071 328 9149

CIRCLE NO. 921

about, the more you are able to express invariant clauses and; when you add a clause to your invariant, then you gain something in understanding what the class is about.

This doesn't necessarily come right the first time - but proper software design doesn't come right first time anyway. The process of improving the software and the process of writing and improving the invariant go together. It's not surprising that these things should appear a little difficult at first - software design *is* difficult. But invariants, and assertions in general, enable you to get to the heart of the matter.

Garbage

Can we talk about memory management. Why do you use garbage collection, instead of 'manual' heap managing systems?

In the manual for Eiffel-S, which is the SIG implementation for DOS, they have a very nice analogy. They say something like: 'An object-oriented program without a garbage collector is like a pressure cooker without a valve. You don't know exactly when it is going to happen, but you know that sooner or later it is going to explode'.

An object-oriented program generally creates a lot of objects. A lot of them are going to become unreachable. You can of course make sure that not too much of that happens - having a garbage collector is not an excuse for generating tons and tons of garbage, you still have to be careful - but if you start managing these things yourself it's dangerous and it's tedious.

It's dangerous because you always run the risk of 'freeing' (in the sense of C's `free()`) an object which in fact is still needed. This is the source of some of the worst bugs that exist in C programming. It's a very serious problem, because the consequence of the error is usually quite remote from the source of the error. Usually when you think you can free an object and you are wrong - there is still some reference pointing to it - you use that reference much later in the program, so that tracing back the cause of the error may be extremely difficult.

As for the tedious part: if you do your storage management manually, you end up polluting your code. If you want to do manual reclamation, you have to write a lot of recursive `free` procedures. As long as you have the kind of complicated data structures that are possible in object-oriented programming, it's not enough to free one object; you have to follow the pointers. You end up being the garbage collector yourself. Programmers have better things to do with their time.

The Ellis and Stroustrup book is the most damaging criticism of C++ that I know

It seems to me that if you don't have a garbage collector, you lose many of the major benefits of object-orientation. Personally I wouldn't write an object-oriented program in an environment in which I didn't have a garbage collector.

C programmers are hostile to garbage collectors because of the time overhead...

That's just because they don't know about modern garbage collection technology. In version 3 of our implementation, we estimate that the overhead of garbage collection - the difference between running a program without garbage collection or running it with garbage collection - is about 20%. But this is not the real overhead, because if you didn't have the garbage collector, you would have some overhead due to manual reclamation anyway. With version 3 - version 2.3 was not as good in this respect - we do not expect anybody to run an Eiffel program without garbage collection enabled. But you can still switch it off if you want.

On the street

How many implementations of Eiffel are there at the moment?

Among the ones I know about are: there's of course ours - Interactive Software Engineering's implementation of Eiffel; there's an implementation by SIG Computers of Germany, which is essentially a DOS and OS/2 implementation, although there's also a UNIX version of it; there's a company in the US called Power Solutions that is about to release its implementation; and there's a GNU version that is in the making but is not released yet. Oh, and I hear that there's some people called Nexnix Ltd in Brighton who are developing a compiler.

By the way, the ISE implementation will support DOS/Windows some time later this year.

What are the main features of version 3 of the Eiffel language and environment?

Probably the most important thing is this 'melting ice' idea, which is an attempt to get the best of both the interpreted world and the compiled world. If you look at why some people use environments such as LISP systems and Smalltalk systems, once you have removed the superfluous arguments, it boils down to just one, practical, serious idea. With an incremental environment such as Smalltalk - I want to say 'interpretative environment' but that's not quite accurate - you can get a very quick turn-around.

Until now, with statically typed, compiled, object-oriented languages you have had to go through a fairly classical edit-compile-link-execute cycle. This means in a big system, even for a small change, there's a fairly long wait. This is an issue we have been grappling with for a long time, because there is absolutely no philosophical reason why you should have to choose between a quick turn-around and static typing.

Static typing is good. It gives you safety, because you are able to catch errors much earlier, and it gives you efficiency, because it make it possible to generate much, much better code. And there's no reason why these goals should be incompatible with a very, very quick turn-around.

This is essentially what this melting-ice technology of Version 3 achieves. You can have extremely quick re-execution after a change, even though you retain the static typing. The idea is very simple. Whatever you change inside the normal development cycle is going to be interpreted, so that it's extremely fast to see the results of a change after you have made it. The interpreter doesn't have any significant negative effect on performance, because typically you're dealing with a big system and after a change only a small part of it will have been affected, so most of it will still be compiled. We think that this is a major advantage that will make all the difference in the world.

Another aspect to version 3 is the presence of the library called EiffelVision. This is a graphical user-interface library supporting various tool kits - to start with Open Look and Motif - with complete source code compatibility. So people can write the best

into user interface applications in terms of high-level concepts like menus, windows and so on, without being concerned with the details of Motif etc. Then they can just port their application to various graphical platforms without making source changes. Eventually this will also apply to Windows and Presentation Manager.

Also important in version 3 is the availability of the standardised relational database interface. This follows the same principles as our GUI library; that is to say you program in terms of the SQL model and then you go to some other RDBMS without source code changes. This is very important for big projects in commercial areas. Most big relational systems currently use C, so it is very important to be able to access traditional data quite easily and to map objects onto relational data.

Looking forward

A recent .EXE Survey showed that about half our readers used C or C++ as their primary development language, about 10% use Pascal, and no other single language collected more than about 5%. There were no Eiffel users. What is your prediction for the result of next year's survey?

Let me consider two years from now, when Eiffel will have had a chance to make an impact on the DOS world. I think that people coming from languages like Pascal and Modula-2 will very naturally migrate to Eiffel. They'll find themselves on safe ground with strong typing, general software engineering concerns and so on. I would see them migrating *en masse* to Eiffel.

I think a proportion of people programming in C will migrate to Eiffel as well. Those are the people who, again, have serious software engineering concerns, and want to be able to guarantee the quality of the code they produce. I don't know how much that is - it's a certain proportion of people writing in C today.

As for people using C++ today; they're still to a large extent the *avant garde*, the early pioneers. I would say that once these people have understood the benefits of object orientation and the limitations of C++, they will look for something more serious. I don't think there is much competition to Eiffel.

So without making any too wild predictions, I think that, if you repeat your survey two years from now, from 15% to 25% will be using Eiffel.

And C++?

I think that five years from now, no-one will be using C++.

This despite the fact that 80% of our C users are looking at C++?

That's all they have heard about. To a certain extent you cannot expect anything else. Suppose somebody has been raised to the Stalinist creed, who has only ever read political literature favouring Marxism and Leninsm. If you ask him, 'What is the next thing?', he's not going to answer 'Democracy and the free market.'

EXE

Many thanks to Dr Meyer for taking the time to give this interview. To find out more about Eiffel, you should contact Caroline Browne at Applied Logic Distribution (081 780 2324).

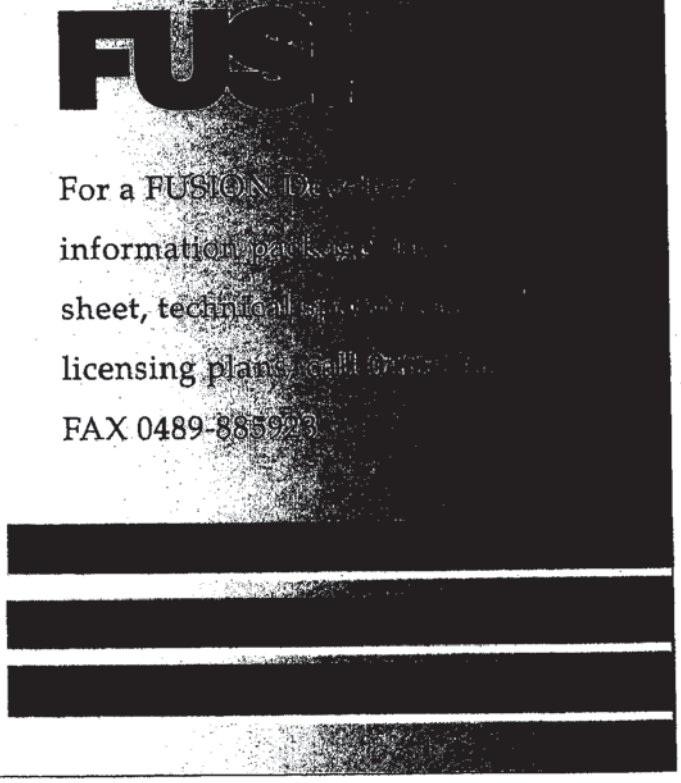
Looking to add TCP/IP network access to your system designs?

Now you can incorporate the industry standard TCP/IP protocol suite in your system designs with *FUSION Developer's Kit.*

Designed for the OEM and systems integrator, FUSION Developer's Kit provides the full TCP/IP protocol suite including TELNET (virtual terminal), FTP (File Transfer Protocol), and R-Commands.

FUSION Developer's Kit also has a flexible C-source code architecture, making it processor and operating system independent.

Currently used in hundreds of process control, embedded systems, and end-user designs, FUSION Developer's Kit from Network Research comes with full support and porting services.



Network Research

CIRCLE NO. 923