Mélanges CRAPEL n° 27

THAT'S NOT ENGLISH,

THAT'S COMPUTING!

Harvey MOULDEN
CRAPEL, Université Nancy 2

Résumé

Cet article rend compte d'une tentative d'amélioration de la capacité d'étudiants français en informatique à comprendre des documents techniques en anglais. Il s'agit de les encourager à approfondir leur réflexion durant leur lecture. L'auteur propose d'ajouter aux activités de compréhension traditionnelles, centrées sur la langue, de tâches au cours desquelles on demande aux étudiants d'explorer à la fois le contenu technique explicite et implicite.

This paper will present an account of how and why its author failed for several years in succession (1993-1997) to significantly improve the ability of French learners of English to deal with technical reading comprehension problems associated with passive reading, implicit information and unclear writing. Despite the disappointing results in the present case, we shall be suggesting that these kinds of problems are important and might profitably be addressed more often in both mother tongue and foreign language reading comprehension courses.

The learners involved were University students enrolled in a three year course of study for a Master's level qualification in computing. Their English courses took up two hours a week of a heavy and (in Computing particularly) demanding workload.

The stimulus for beginning this work was the observation that about 40% of these students were failing, year in year out, to understand parts of texts in English about Computing whose technical content was well within the reach of their technical knowledge and which they could often correctly translate into French. When questioned closely to see if they had grasped the meaning of these passages, they were either unable to supply answers or gave answers that were erroneous.

For instance, an authentic text on data compression given in class contained the passage below.

In addition to compressing, on-the-fly compressors are more efficient in their use of disk space. DOS allocates space to files in clusters, where each cluster is composed of two to 32 512 byte sectors. A cluster is the minimum allocation unit - no matter how little data is in a file, each file must be made up of a whole number of sectors. Typical hard drives have 4-KB clusters, which means that DOS wastes 3 KB of space in storing a 1-KB file. Since all of the compressor's interaction with DOS is in a single normal file, the compressor can allocate space on a thriftier sector basis.

The last sentence posed the students translation problems in predictable places (choice of meaning for " *since* ", interpretation of the noun phrase " *a thriftier sector basis*")

and might well have figured in a traditional " text plus questions " exercise as the basis for a question such as:

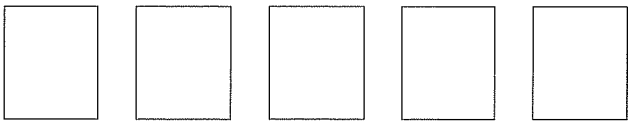
" What is the effect of DOS's interaction with the compressor in a single normal file? "

designed to see what sort of interpretations or translations the students would come up with for " *on a thriftier sector basis* ". Once acceptable translations had been established, the students would be asked to explain exactly why the fact that interaction with DOS in a single normal file meant that disk space was going to be used more efficiently. Now this is a question which gets to the heart of the understanding of the sentence in a way the simple " translation " question does not. For the answer to the question is not given explicitly in the text. The author relies on the reader to use the information given in the preceding 3 sentences to work this out for himself or herself. One chain of reasoning might go something like this:

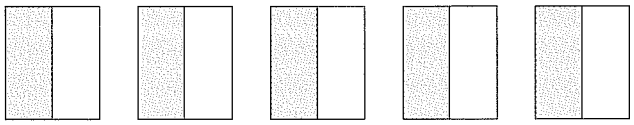
- Compression will normally involve an appreciable number of files.
- " *the compressor's interaction with DOS is in a single normal file* " must mean that all these files when compressed will be concatenated, head to tail with no space wasted between successive files, into a single file. (This is probably the key step in the process of understanding.)
- The only disk space wastage incurred when storing compressed files will be due to the length of the single file containing them not being equal to a whole number multiple of 4KB, i.e. will never be more than just under 4KB.
- Which is very little compared with often wasting 1, 2 or 3 KB for each compressed file recorded to disk if you treat them separately.

The better students, when prompted, managed to produce diagrammatic illustrations of the situation such as that below:

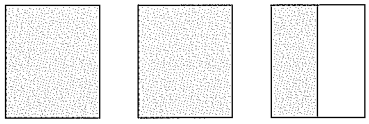
5 empty 4KB sectors:



5 two kilobyte files occupy 20 kilobytes of disk space when the uncompressed files are recorded separately by DOS:



The same 5 two kilobyte files after concatenation occupy only 12 kilobytes of disk space when recorded in a single normal file by the compressor's interaction with DOS (and will actually occupy even less space when compressed):



Many of the students were incapable of " filling in " the meaning of the text in this way[1], despite the fact that their previous courses in Computer Architecture had familiarised them with the way data is recorded on hard disks. In general, whenever a text demanded this kind of effort of the reader, the less able students fell prey to misunderstandings or understood nothing at all. Worse, they also failed to understand texts which, apparently, were free of implicit content that was difficult to understand. Why was this?

The kind of answers students often gave when probed on their understanding of a text strongly suggested that

1 The brain-work involved here is surely rather greater than that demanded in the general run of reading comprehension questions where correct inferences have to be drawn. Compare the above example with one given in the introduction to an exercise designed to prepare the reader for inferential questions given in the TOEFL (Broukal and Nolan Woods, 1991).

In 1896 the fabulously wealthy John D. Rockefeller declared that the great university he had founded was the best investment he had ever made in his life.
What can be inferred from the sentence?
John D. Rockefeller was richer than the vast majority of his fellow Americans.
Rockefeller was delighted with the university he had founded.
Rockefeller had made other investments in his life.

they had not been thinking very much about what they had been reading. Indeed, these answers were sometimes, from a technical point of view, so startlingly wrong as to make one wonder how certain students had managed to get so far in their chosen course of studies. What seemed to be happening was that, faced with a text in English and left to their own devices, many students would tunnel through it translating it sentence by sentence (or, worse, word by word), forgetting each sentence as soon as it was " done ", taking little close interest in what the text was saying and, above all, rarely asking themselves questions and calling on textual clues and their own technical knowledge when a sentence didn't make much sense. When this happened students would say things like:

" Je comprends les mots mais je ne comprends pas ce qu'il y a dedans. "

I understand the words (in this sentence) but I don't know what it means.

Informal discussion with individual students revealed again and again that their previous training on computer literature in English during the preceding one or two years had been of the form " sentence by sentence translation round the class ". If this approach to reading comprehension had gone no further than to demand a correct translation, it may bear some responsibility for the students' translation reflex and their lack of curiosity about the texts they were now being offered. Exercises of the " text plus questions " type were sometimes mentioned, but it sounded as though most of the texts were not very technical and that the questions were usually on explicit content.

It was decided, therefore, to look for and devise activities aimed at:

- persuading the students that translating correctly is not necessarily the same thing as understanding
- getting them into the habit of thinking more about what they were reading by asking themselves questions as they read and then to use their specialist knowledge to find answers to these questions.

The hope was that this more thoughtful, less translation-oriented approach to reading would help them not only to better understand fairly straightforward, fairly clearly-

written texts but also to cope with passages where the author, consciously or not, leaves it to the reader to complete his or her meaning.

Below we will list brief descriptions of the kinds of activities which were used. The activities ran alongside reading comprehension work of a more traditional sort (reading for gist, selective reading, technical vocabulary exercises, exercises on noun phrases, modals, linkwords etc.) as well as work on listening, speaking and writing.

ACTIVITIES TO DEMONSTRATE THAT TRANSLATING IS NOT NECESSARILY UNDERSTANDING.

The students were given a short text in French to read and asked to imagine that they had produced it themselves by translating from English or some other foreign language. One text used was that below, taken from the works of a twentieth century French philosopher and offered, without comment, as a space-filler for the edification of the public in the French national daily, *Le Monde*.

> *Les trois contresens sur le désir sont : le mettre en rapport avec le manque ou la loi; avec une réalité naturelle ou spontanée; avec le plaisir, ou même et surtout la fête. Le désir est toujours agencé, machiné, sur un plan d'immanence ou de composition, qui doit lui-même être construit en même temps que le désir agencé et machiné. Nous ne voulons pas dire seulement que le désir est historiquement déterminé. La détermination historique fait appel à une instance structurale qui jouerait le rôle de loi, ou bien de cause, d'où le désir naîtrait. Tandis que le désir est l'opérateur effectif, qui se confond chaque fois avec les variables d'un agencement. Ce n'est pas le manque ni la privation qui donne du désir: on ne manque que par rapport à un agencement dont on est exclu, mais on ne désire qu'en fonction d'un agencement où l'on est inclus (fût-ce une association de brigandage ou de révolte).*
>
> G. Deleuze: Dialogues

The students were then asked whether they had understood the text. Nobody ever did, so it would then be suggested that flawless translation is not necessarily

accompanied by understanding The students would usually object that they could not be expected to understand texts on subjects about which they knew nothing at all.

In this case they were asked to translate a short and easily translatable text on a computing subject with which they were not familiar but which explained the subject using concepts they were familiar with. They were then given a few searching questions to answer concerning the text they had successfully translated. Advantage of the ensuing perplexity was taken to make the point that this time they were no longer on unfamiliar ground, had translated correctly but had nevertheless not understood everything. If they then objected that they couldn't be expected to understand computing topics they had not yet been taught in class (and they often did), they were asked what sort of texts in English they thought they would meet during their future career. Texts about things they already knew all about or texts about things they didn't know all about? The latter usually. And would they have to translate them and then hand them in to be marked? Or would they have to use them to extract the information they needed to do a job properly?

ACTIVITIES TO GET STUDENTS TO THINK MORE WHILE READING BY ASKING THEMSELVES QUESTIONS

The students were shown how they could generate questions on a short text by asking " what? ", " why? ", " when ? " and " how? ". For example, the passage below

> *(1) Every time you use your hard disc, it gets a little slower. (2) Because DOS was written when capacity was more important than speed, it tries to pack files on disk as conservatively as possible, dividing them into clusters and squeezing them as tightly as it can. (3) But organising the clusters in this way means that DOS has to search all over the disk for errant clusters, and also means easier file recovery if you delete a file by mistake.*

might raise the following questions:

Sentence 1: What gets slower? Does the disk go round slower or does reading from and writing to the disc slow down? Why is there a slowing down?
Sentence 2: What does capacity mean exactly in this context? What does *as conservatively as possible* mean? How can you *pack files on disk as conservatively as possible*? What does *squeezing* mean exactly?
Sentence 3: What are ***errant*** *clusters?* Why does this organisation mean you have to search all over the disk for errant clusters? Why does it lead to easier file recovery if you delete a file by mistake?

The students were asked to suggest answers to these questions.

Some of the questions are easy (the first two concerning sentence 1 for example). But others are more difficult. In particular, the last 3 questions on sentence 2. Here, by talking about pack(ing) *files on disk as conservatively as possible and squeezing them as tightly as it can* the author sacrifices clarity to raciness of style. In fact, this sentence is so imprecise that anyone unfamiliar with the subject is probably not going to be able to understand the next sentence (*But organising the clusters in this way means that...*) If one does not already have, from previous knowledge, a clear mental picture of the way data is recorded to disk, one is going to be hard put to it to see why DOS has to search all over the disk for errant clusters and why you get easier file recovery if you delete a file by mistake.

So the point of getting learners to ask themselves questions like this was to ensure that they would get involved with what the text was saying and mobilise any knowledge available to them which might help them to understand what they were reading. The learners were encouraged to apply the questioning approach above whenever detailed comprehension was needed. It was suggested to them that they underline sentences where difficulties subsisted and that, as they read on, they return from time to time to these sentences to see if new information could throw any light on their meaning. When faced with difficulties, the learners could also scan ahead in the text for iconic clues or for repetitions of poorly understood words to see if fresh contexts provided further clues.

Asking people to call upon subject knowledge to aid comprehension is all very well, but sometimes the students did not possess the necessary knowledge. In this case, they were referred to a second text which supplied the information needed in a more explicit form. " Text-branching " of this sort, if pursued further, could lead to the original text disappearing from view, so to speak. This need not necessarily be a problem, though, within a larger time-scale than that of an exercise intended to be finished before the end of a class. In fact, leaping back and forth between texts was encouraged in another long-term exercise intended to develop curiosity during reading. Projects were given where the students had to individually research computing subjects on the Internet. The research was carried out by first of all assembling a relevant introductory text or collection of texts and then acquiring thoroughgoing knowledge of their content by branching out towards fresh texts when concepts mentioned in the starting texts were unfamiliar or incompletely or poorly explained.

The learners were given practice in thinking about what they were reading via teacher-set questions and tasks that probed to what extent a passage had been understood in a more searching way than the traditional type of comprehension question which merely requires location and translation of explicit information. When the students got into difficulty, they were counselled individually on what other questions they could ask themselves and how they could reason their way to a plausible solution of the problem. Below is a list of the types of tasks used. Some of them (explaining meanings of words, diagram drawing,) are familiar. Others may be more original, particularly as regards the technicality of the thinking they require. The text extracts used were authentic and were mostly drawn from *Byte*.

Say exactly what a word or term means in context.

For example, in the text " *On the fly compression* " (Appendix 1) the students were asked to explain what they understood the word " invisible " to mean in the paragraph below.

LZ and similar techniques make for fast compression and decompression, although decompression is usually

somewhat faster. Both attributes serve on-the-fly compressors, since speed is critical for invisible operation and read accesses are often far more common than writes.

A common, if understandable reaction, was to take the word at its face value and say: " invisible " means " can't be seen " and leave it at that, no questions asked. When pressed to explain further, students would say that " visible operation " was what you could see happening on the computer screen whereas " invisible operation " was what happened out of sight inside the computer. They were less forthcoming when asked to explain why they thought the text was saying that speed was so important for what happened inside the computer but not for what happened on-screen. In fact, " invisible operation " here, with all the insistence on speed in the paragraph, presumably means that compression and decompression take place quickly enough for their occurrence to introduce no observable delay in functioning. If the students had asked themselves why " speed is critical for invisible operation " from a file user's point of view they might have come to the same conclusion.

Assess the technical precision or appropriateness of a word or statement.

It sometimes happens that the author of a technical text says something which does not seem to be totally consistent with what s/he has said so far. This provides another opportunity for testing whether a learner has thoroughly understood the drift of the preceding text and, if not, to show him or her how to improve understanding via a more thoughtful approach. For example, in the text " *Interleaving: delivering the data on time.* " (Appendix 2) the following statement occurs in paragraph 3

If instead of following one another, sectors with successive numbers have one or more other sectors between them, the next sector will be approaching the disk drive head just when the controller is ready for it.

One of the questions posed concerning this text (see Appendix 2 for others) was: " Do you think the word " just " is appropriate here? Justify your answer. " This question was given because the information given up to this point in the text could lead the reader to wonder how it is that the other

tasks needing to be done always get done just in time for the next sector to be read. This seems too good to be true. It seems more reasonable to suspect that sometimes these tasks must get done well before the arrival of the next sector, sometimes just before and sometimes after (or even well after) the arrival of the next sector. In fact, the text confirms this suspicion in the next paragraph when it raises the possibility of one interleaved sector not being enough. Nevertheless, many students – including some who had taken the teacher's advice to read all the text before answering any questions - did not hesitate to affirm that the word " just " was, indeed, perfectly appropriate but they offered little in the way of justification other than an unquestioning faith in modern technology and the pronouncements of journalists. When this happened the students were asked to think about the following questions:

- Given what the text has said about the various tasks which have to be done other than reading disk sectors, what conditions would have to be fulfilled in order that, invariably, the next sector would be approaching the disk drive head just when the controller was ready for it?
- Given the variety and nature of these extra tasks, are these conditions likely to be met with every time?

Fill in missing details in a process description.

A task of this sort was provided by the text " *Speed reading: choosing between software and hardware caches* " (Appendix 3) where, in the last paragraph (below) , the term *check to the contents of the cache* is sufficiently lacking in explicitness as to puzzle some students when they are asked to say what it is, exactly, that is checked.

With this, the most frequently accessed pieces of data, up to the capacity of the cache, are stored. When the cache is full and some other piece of data not presently in the cache is loaded, then the least accessed of the data already in the cache is flushed and replaced by the new data. The down side of this is that the machine incurs the overhead of having to check to the contents of the cache with every transaction. But as the data read from the cache is delivered at an exponentially higher rate, the overall result is better throughput. As disk caches will always cache the directory, substantial performance increases can be gained in locating a given file alone

In these cases, the students concerned were invited to look for answers to the following questions :

What exactly is a transaction ?

What do you need to do at a detailed step by step level in order to carry out the process described in the sentence " *When the cache is full ...* " ? Is there any checking involved here ?

while bearing in mind what they had understood so far and, in particular, what they had understood of the four preceding sentences. For a potential difficulty here for the reader is that the text talks about flushing the least accessed data when the cache is full and space is needed for new data but does not go so far into detail as to say that the least accessed data has to be identified before it can be flushed. This is an understandable omission which the author, had he wished to be perfectly clear, might have repaired in the next sentence by saying that this is what happens when the machine " *checks to the contents of the cache* ". As the text stands, the reader has to deduce this for him or herself.

Fill in missing links in a cause-effect chain.

We have already seen one example of this kind of task at the start of this paper. Another one is provided by the text Spin Doctoring (Appendix 4) where, just before the end we have :

> *Server administrators typically strive to put data that belongs together in the same area on a hard drive to reduce the effective seek times of drive access. Thus, their access time is especially sensitive to changes in latency:*

The *Thus* at the start of the second sentence gives the impression that everybody should grasp, almost without having to think about it, why it is that " *their access time is especially sensitive to changes in latency* ". In fact, here, a very small tree of a word is hiding a sizeable wood of cogitation which may involve some or all of the steps below.

- Going back in the text to refresh one's memory on the meaning of " *seek time* ".
- Going back in the text to refresh one's memory on the definitions of " *access time* " and " *latency* ".
- Realising that the key to understanding the assertion made is in the access time equation given earlier in the text.
- Thinking about the access time equation and having the mathematical perception needed to see that access

time is going to vary more sharply with changes of latency if seek time is small.

Draw a diagram.

Asking the students to represent their understanding of part of a text in the form of a diagram revealed problems that mere translation often hides. The following task, given with the text *How interrupts work.* (Appendix 5) appeared feasible enough but frequently gave trouble because it requires more than a passive reading of the passage concerned.

> Paragraphs 6, 7, 8. Draw a diagram showing: the flags register, processor, IRQ, Software, interrupt enable flag, INTR, interrupting device, interrupt input and how they are connected.

To do the diagram, one needs to pull together information from the foregoing paragraphs (that the interrupting device and the processor are at the opposite ends of the system; what the IRQ is and does) and then, by a careful reading of paragraphs 6, 7 and 8 to visualise the way the remaining items fit into the picture.

Carry out a calculation.

This is another effective way of seeing whether part or all of a text has been really understood or not. The text *Wait States* (Appendix 6) was given, for example, along with the following task:

> Some years back, a computer manufacturer included the following details in an advertisement for his latest machine: 80386X, 33 MHz, 8 megabytes 0 wait state RAM and, further on, 80 ns memory. In fact, the information given contains a lie. Where is it? Why can this statement only be a lie? (Hint: you will need to do a small calculation.)

The text *Wait States* has to be followed attentively if the reader is to be in a position to perceive that the dubious information must lie in the claim of 0 wait state with 80 ns memory and then to investigate what is wrong by calculating the clock period of the computer concerned. This reveals that the latter is less than the 80 ns memory access time and hence, that the 0 wait state claim is impossible.

This particular question had the happy advantage of appealing to the sleuthing instinct of some of the students.

No great ingenuity was required to devise it, since the germ of the question was contained in the continuation of the text, which the students did not see. Finding other questions of this sort was sometimes facilitated by texts introducing illustrative calculations which could be handed on to the students to do by doctoring the texts so that the calculations did not appear. However, this is a slightly dirty trick to play since it renders the texts more difficult to understand than their authors intended. Other examples of this kind of task may be found in questions 3 and 4 of Appendix 2 and in question 8 of Appendix 8.

Write a computer instruction or short program using extracts from a programming manual.

Of all the activities given, this one and that below were certainly those which were most relevant for the future computer programmers who received them. Fortunately, the abundance of teaching literature available concerning the principal computer languages aids in the preparation of exercises, since these works often contain worked examples and problems with solutions which can be given accompanied by the relevant manual pages. An example is provided in Appendix 7 where the pages of the *IRIX 5.2 manual* corresponding to the *csplit* and *ed* commands are shown. The students were provided with these pages together with the following programming task to carry out.

> Use the IRIX manual extract provided (ANNEXE) to carry out the programming exercise below.
>
> /usr/dict/mots is a file containing a list of French words in dictionary order, i.e. starting:
>
> ABACA
>
> ABACULE
>
> ABAISSABLE
>
> and continuing line by line to the end of the file at:
>
> ZYMOTECHNIE
>
> ZYMOTIQUE
>
> ZYTHUM
>
> Write the instruction which splits the file into two halves, the first containing all words up to (but not including) ELLE, the

second containing the rest. Warning: don't forget that the word ELLE is contained in certain other words (BELLE, BELLE-MERE, BIELLE, CRECELLE etc.)

The *csplit* command was something the students had never seen before, but the task given was not expected to give them too much trouble since they were already using the manual in English regularly as part of their studies in programming. In fact, nearly half the students came to grief on it when tested under examination conditions. The pages given are, it has to be admitted, pretty heavy going for weaker students. They contain vocabulary problems in places, a sticky passage which doesn't need to be understood to solve the problem given, and a fair amount of digesting and selecting to do. The extraordinary answers given by some students suggested that they had either been overwhelmed by the text and had given up trying or hadn't even bothered to look at it.

Predict the effect of executing a computer instruction or short program.

This activity is the inverse of that described above. Once again, the students are supplied with manual extracts, but this time they get ready-made computer instructions or bits of programs and have to say what they think these will do when they run. For instance, the IRIX 5.2 manual pages in Appendix 7 were given with the question:

What will be the effect of running the following instruction ?

csplit –k prog.c '%main (%' '/^}/+1' {20}

The question now is: what sort of results were achieved by these " get them thinking " activities? Unfortunately, the answer is " pretty disappointing ones. ".

It was possible, via individual monitoring of students during regular interviews and common entry/exit tests to get an idea of whether any progress had been made. For the great majority of students no perceptible change occurred. Only 10% of the students showed signs of engaging more actively and effectively with the texts and achieving significantly better results in the tests. Very often, the students who improved were those who had failed end-of-year exams the year before and were now going round the course for a second time and trying a lot harder.

So what went wrong? Well, lots of different things went wrong, but there was a root cause.

Most teachers, no doubt, have their limitations and failings and the author is generally the first to find fault with himself in this field, but the basic problem here does seem to have been that the overwhelming majority of the students involved were just not interested in doing English or any other foreign language. This was confirmed when other language teachers arrived on the scene, for until then, the author had been the only language teacher in the establishment and these students his only students. Before their arrival, he could be held to be the principal cause of student dissatisfaction and work-shyness. The English and German teachers who joined him towards the end of the work described above, despite running " straight " courses, soon complained of their new students, declaring them to be the most dismayingly unmotivated, intractable and unrewarding they had ever had. A new English teacher, freshly arrived from teaching youngsters in secondary school found no difference between the behaviour of these 20 - 23 year olds and her previous charges. Any remaining doubts that anyone might have entertained on this subject were, hopefully, removed recently when a student spokesman informed the head of department that the students didn't give a damn for foreign languages (" *Les langues, on s'en fout.* ").

Under these conditions, any kind of innovation which - as this one did - demanded both intellectual effort and a substantial change in ingrained classroom habits was probably going to meet learner resistance. And so it proved.

The majority of the students concerned by this work complained of the difficulty of the exercises and avoided doing them whenever they could. They often pointed out - despite frequent justificatory input from the teacher - that this sort of work wasn't English, it was Computing (hence the title of this paper - a quote from a student who came to see why she had had a very good mark for an almost impeccable translation of a text and a very bad one for her answers to follow-up questions designed to test whether the text had been understood: *Ça, c'est pas de l'Anglais. C'est de l'Informatique*). Worse marks than usual were a frequent cause of friction with students used to being marked on their talent as translators.

Irritation over the need to think Computing in the English class was compounded by their conviction that they really had no problems at all in reading, that the exercises were a waste of time and that, instead, the teacher should have been giving them what they were used to being given: grammar exercises to do and debates where everybody was made to talk English. Above all, no Computing in English. A place for everything and everything in its place. (A short calculation designed to check understanding of part of a text was rejected by a glowering student who declared: " I don't do Maths in an English class.") Irritation was also provoked by the fact that an English teacher allowed himself a modest amount of expertise in Computing. This surfaced in the form of " testing " (sly catch questions or the making of idiotic technical pronouncements with a straight face) and one or two direct demands for credentials (" What qualifications have you got in Computing anyway? ").

Naturally enough, the students did not keep their disgruntlement to themselves but vented it regularly in the direction of the teachers entrusted with the smooth running of the department. This led to much explaining viva voce and in writing on the part of the author over a period of 5 years. To little effect though. The students were right and the English teacher was wrong. The official " advice " from above (admittedly from colleagues with much other work to do and who were doing a harassing job on a voluntary basis) was to keep away from Computing and give the students what they were used to : grammar, vocabulary and pronunciation (for this had been good enough for the colleagues when they had been taught English). The students didn't go to English classes to think; they had plenty of that to do elsewhere. Two or three colleagues in Computing or Mathematics did, however, approve of the author's efforts and one publicly deplored the lack of support given by the other teachers. Another, shortly before extricating himself from the department confided that the frustration which had provoked his departure coincided with that of the author.

Working conditions of this sort[2] are hardly an ideal ter-

2 One or two similar experiences have been reported (personal communications) by language teachers seeking to innovate in other French higher educational departments where languages are ancillary subjects.

rain for testing novel approaches and, in the present case, did little to bolster the innovator's self-esteem or procure him refreshing sleep at night, so the exercises described above were, in the end, to all intents and purposes phased out and replaced by less controversial activities. There would seem to be little hope for the innovator who finds him or herself faced with both unmotivated students and an antagonistic departmental ethos. A close-knit, determined team of teachers could probably survive the latter (it is harder to sideline several like-minded teachers than one enthusiastic but misguided eccentric) but slim indeed are their chances of making much headway with students who are a) not interested and b) do not even (now) need to get a good mark in English to get their diploma.

The reader may be thinking, " Well, yes. Poor chap. Very sad and all that. But ... for heaven's sake. Is there all that much of a problem? Surely written ESP English is clear enough for the most part, isn't it? O.K It sounds as though the students really were pretty awful. But fancy expecting them to keep fiddling about trying to answer smarty-pants questions on bits of texts they probably wouldn't have understood any better in their own language. Isn't there enough to do just getting learners to read English about as well as they do their own language, let alone having to become a computer expert as well? "

Well, one thing at a time. Firstly, is there all that much of a problem? This question did occasionally cross the author's mind when skimming fruitlessly through text after text in search of tricky passages for a new exercise. But actually the students turned out to be better than him at the job, discovering problems in places where he had seen none. And curiously enough, it was only when the work described in this paper had already been abandoned for several years (and that student levels had sunk even further) that the teachers of Computing could be heard talking more frequently about manuals both French and English and saying things like:

" They (the students) get into terrible trouble with the error messages because they don't go down deep enough."

" The trouble is that they (the students) don't see the

implicit content. "

And the teachers had problems sometimes too. In both English and French.

" If they'd set out to write it so that nobody could understand it they wouldn't have written it any differently to this. "

" There's a lot of implicit stuff in it. "

" Oh my goodness, yes. Manuals are really heavy going."

All this happened in one small part of a university in north-eastern France, but there seems no reason to think that things are so much different elsewhere. Indeed, at the same time that the author was doing battle with his students and superiors, in the United States a Professor of Chemistry was writing to the Royal Society of Chemistry's Education Division in London to say that:

> *" Among undergraduate teachers a frequently heard lament is, 'the bright ones are brighter than ever but so many of the rest seem to be unable to read'. Probably what is meant here is that the situation is not so much that they are unable to read as that they seem unable to understand what they read. " (Markham. 1995)*

" Ha! " interrupts the now triumphant reader. " French Computing teachers struggling with manuals in French? American students not understanding their Chemistry books? There. That proves it. This isn't a foreign language problem at all. It's a universal mother-tongue -reading-for-special-purposes problem. No business of a foreign language teacher at all. You don't expect us all to become experts in Computing, Chemistry, Finance, Law and everything else just so we can neglect our responsibilities as English teachers and drive our learners (and ourselves) potty, do you? "

Well, no. Of course not[3]. But if there **is** a universal mother tongue reading for special purposes problem that spills over into foreign language reading for special purposes, **who** is going to do something about it ? Do foreign language teachers have to wait until mother tongue tea-

3 Although I feel it is not unreasonable to ask a language teacher to get to know something about his or her students' main subject if these are the only students s/he teaches.

chers do the job for them? Why shouldn't they have a go themselves? But maybe people already are doing something about the problem and I've been too busy in my little corner to notice. A quick survey suggests, however, that nothing on a massive scale is happening. In the field of English for Computing, as we have seen, many of the French students in the author's classroom a few years back seemed to have previously approached reading on computing subjects mainly via translation. Past papers for the French BTS examination in English in computer-related disciplines at that time (J.F.Dreyfus. 1995) also feature translation and summary writing of moderately technical or adapted texts. A cursory look at the reading comprehension sections of a few English for Computing course books from the late 80's to the mid 90's (see bibliography) reveals a majority of non-authentic texts at a very basic technical level with questions and tasks which do not appear to test much more than superficial and piecemeal understanding of explicit information. A recent update on the writer's students' previous work in reading technical texts showed that translation and summary writing were still alive and well and were accompanied by vocabulary trawling and answering questions on texts at both general public and specialist level. The questions posed, according to the students, called for knowledge of English but rarely of Computing. Implicit information is often mentioned in the literature as a problem in reading comprehension, but if the TOEFL questions on it are typical of what the problem is generally perceived to be (see footnote pp 2-3) then, so far, learner-readers have only been asked to recover relatively easily accessible implicit information compared with that in the example given at the start of this paper (p.1).

It seems possible, then, that more might be done in the field of LSP reading to equip learners with strategies not only for penetrating authentic technical writing rendered difficult by authorly haste, but also to help them with texts which are relatively clearly written but need sustained short range attention and appropriate navigational and selection techniques if they are to be understood and used effectively in professional life. Despite his failure to make much headway with the activities described in this paper, the author is reluctant to abandon the idea that this approach

is on the right track and that better results might be obtained with more enthusiastic learners and more supportive subject specialists.

However, if anybody else envisages work of this kind on authentic specialised texts, it is, in the author's opinion, vital for him or her to acquire some knowledge of the subject. Apart from the fact that learner confidence in the teacher may be eroded or utterly destroyed when s/he is caught out in gross technical error, subject familiarity facilitates the identification of likely problem passages in a text, helps with explaining things and, above all, is of crucial importance when it comes to appealing to (or completing) a learner's technical knowledge in order to guide his or her thought processes when difficulties arise. Without this last kind of help from the teacher, the learner is probably not going to get very far with learning to think himself or herself out of the kind of tight textual corners where purely linguistic input from the teacher is not sufficient.

Obviously, the vast majority of LSP teachers are going to have neither the inclination nor the time to swot up Medicine or Forestry or Avionics or whatever, let alone all three or more if they have learners specialising in a variety of disciplines. Nevertheless, valuable work using activities of the kind indicated in this paper might, surely, be done using texts dealing with subjects with which both student and teacher are familiar. It would then be necessary to investigate to what extent any skills acquired by learners in this way transfer to their reading of texts in their own field of specialisation.

A final idea. The sudden (but not universal) interest aroused in students when specialised technical documents of use to them in an imminent or just beginning programming project were presented in the author's English classroom suggests an avenue which might be explored where a sympathetic and co-operative teaching environment exists: language teachers working in collaboration with the specialist teachers. These latter are the people who understand the guts of the texts better than we do and who are taken seriously by students for whom languages and language teachers are a bit of a joke and for whom the main discipline is the important stuff that gets you more marks

and pots of money afterwards. In the field of Computer Programming, for instance, ad hoc help on comprehension problems with manuals in English (when these are the only source of information available) could be integrated with practical programming work with computing teacher or language teacher, or both, on hand as helpers. In this way, the students would get more realistic and motivating exposure to English reading than in the English classroom and could get help in understanding the language better, rather than just being fed the answers by brighter students (which is what happens at present). Indeed, to encourage the weaker-in-English students to rely more on their own efforts, examination questions in programming might even involve comprehension of a programming manual in English where no French translation of the manual exists. This kind of ploy, of course, is perilously likely to call forth protests in the form of the back-to front version of this paper's title, i.e. *That's not computing, that's* ***English***. One of the author's colleagues in Computing actually took (spontaneously and to his everlasting credit) the risk of including in a programming test, a final exercise involving reference to an unseen extract from the English-only manual. Most of his students didn't even attempt to do the question. Some of these, presumably, couldn't do it because of the English. The others, little doubt, **wouldn't** do it because of the English. And this sort of nonsense will go on as long as there is not more real, nitty-gritty speciality in the teaching of LSP for reading, and as long as learners and teachers alike insist on classroom separation of foreign language and speciality skills which need to be practiced together if they are to be used together successfully in understanding foreign language specialist literature.

BIBLIOGRAPHIE

BROOKS, M and LAGOUTTE, F (1993). *English for the computer world.* Paris : Belin. 176p.

BROUKAL and WOODS, N (1991). *NTC's preparation for the TOEFL.* Lincolnwood : NTC. 228p..

DREYFUS, J.-F (1995). *BTS Anglais, tertiaires et industriels. Sujets corrigés.* Paris : Nathan.

GLENDINNING, E.-H and McEWAN (1989). *English in computing.* Edinburgh : Nelson.

HICK (1991). *English for Information Systems.* Hemel Hempstead : Prentice Hall.

MARKHAM, J.-J (1995). *Royal society of chemistry's education division annual report.* Cambridge.

WALKER, T (1989). *Computer science.* Londres : Cassell.

Appendix 1 : On-the-Fly Compression

Data compression works by translating a representation of data from one set of symbols to another, more concise series of symbols. On-the-fly compressors use a variety of lossless compression algorithms. and most manufacturers are tight-lipped about the exact techniques. However, the most common algorithms for general lossless compression are variations on dictionary-based schemes such as LZ (Lempel-Ziv) and its patented cousin LZW (Lempel-Ziv-Welch). For example, Stacker uses a compression algorithm that Stac Electronics calls LZS for "Lempel-Ziv-Stacker."

Dictionary-based compressors use symbols to represent recurring strings in the uncompressed input. An encoding dictionary maps these symbols to the strings they represent. With most dictionary algorithms, the decompressor can completely reconstruct the encoding dictionary from the compressed data stream - the compressor doesn't need to explicitly include a decoding table.

LZ and sirnilar techniques make for fast compression and decompression, although decompression is usually somewhat faster. Both attributes serve on-the-fly compressors, since speed is critical for invisible operation and read accesses are often far more common than writes.

Like all lossless compression, on-the-fly compression works by removing redundancy in the source data; therefore, it's highly dependent on the input data type. Source data with a high degree of redundancy (e.g. bit maps or mostlv empty databases) compress verv well, while more random data (e.g. executable binaries or precompressed archives) don't compress well at all. Text files usually land somewhere in thc middle.

Byte

Appendix 2 : Interleaving: delivering the data on time

1 Each circular track of a hard disk is divided into sectors - arcs of the circle that contain equal portions of the data stored on that track. You may well ask, "Why don't they make the entire track one huge sector?" The answer is

that the disk drive controller must always read or write whole sectors at a time. Having only one sector per track would mean that every read or write would require as much as two revolutions of the disk; up to one revolution to get to the beginning of the track and another full revolution to read it. (The designers of the Commodore Amiga, incidentally, tried to implement this approach with floppy disks, but they added a special trick. The unique Amiga disk drive controller can start a read or write operation at any point on the track - something no other controller I know of can do. This sets the time for every read or write to exactly one revolution of the disk. Alas, the latency is still a bit long, causing the Amiga floppy discs to exhibit lackluster performance except on large files.

2 Each track of a standard IBM PC hard disk contains 17 sectors of 512 bytes each. The outermost ring in figure A shows the most obvious arrangement of the sectors. They're placed in ascending order around the track, from 1 through 17. (This is called 1-to-1 interleave.) In practice, however, this might not be the most efficient arrangement. Often, disc drive controllers, disk I/O routines. and the host systems they run in require time between accesses to successive sectors. They may use this time to transfer data to and from memory, acquire control of the system bus, set up direct-memory-access channels, or allow other I/O to take place. If the time required for these tasks is too long, the controller may find that the next sector it wants is already under the disk drive head - or past it - by the time everything is ready.

3 Interleaving solves this problem. If instead of following one another, sectors with successive numbers have one or more other sectors between them, the next sector will be approaching the disk drive head just when the controller is ready for it. The second ring from the outside in figure A shows an example of 2-to-1 interleave, in which sectors with successive numbers always have one other sector between them. The order becomes 1, 10, 2, 11, 3, 12, 4, 13, 5, 14, 6, 15, 7, 16, 8, 17, 9.

4 If the system can keep up with it, 1-to-1 interleave will generally provide the best performance. But there are severe performance penalties if the interleave factor is too low.

The controller will "miss" each sector - possibly by only a few hundred microseconds and will have to wait until it comes around again. If the interleave factor is set one notch too high (say 3-to-1 instead of 2-to-1) the penalty isn't nearly as bad.

5 The optimum interleave may be different even for two operating systems on the same machine. On my 8-MHz AT clone - not a particularly fast machine by today's standards - DOS works best at a 1-to-1 interleave. OS/2, however, likes a 2-to-1 interleave; the intervening sector gives the system time to handle interrupts and switch in and out of protected mode as needed.

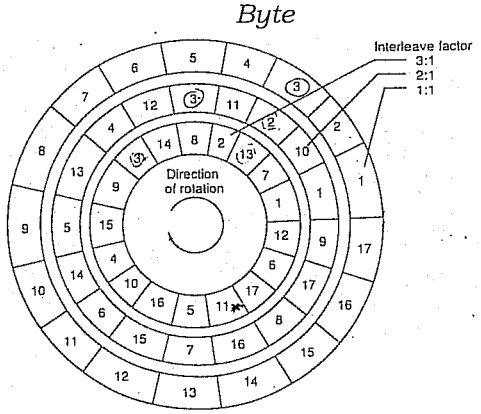


Figure A : Arrangements of sectors for different interleave factors. At one-to-one interleave, the sectors are numbered in sequence ; at 2-to-1, sectors with consecutive numbers are separated by one other sector, ans so on.

Interleaving: delivering the data on time

Questions.

1. 4ème phrase du texte: quels sont les temps maximum et minimum de lecture d'un secteur dans le cas d'un disque du type dont il est question dans cette phrase?

2. La 2ème phrase du 3ème paragraphe affirme que: *If instead of following one another, sectors with successive numbers have one or more other sectors between them, the next sector will be approaching the disk drive head just when*

the controller is ready for it. Comment comprenez-vous *the next sector*? Est-ce que l'emploi du mot *just* dans cette phrase vous paraît entièrement approprie? Expliquez votre réponse a cette dernière question.

3. Voir le 4ème paragraphe. Soit un IBM PC dont le disque dur a un " *interleave factor* " de 2:1 seulement alors qu'il aurait fallu un " interleave factor " de 3:1 Calculez le temps "perdu" (en secondes) lors de la lecture de toute une piste. On suppose que la lecture commence au début du secteur I et que le disque tourne a 3600 tours/minute. Montrez clairement comment vous avez obtenu votre réponse.

4. Maintenant (même machine que ci-dessus), calculez le temps "perdu" en secondes en lisant toute une piste quand 1' "interleave factor " est de 3:1 alors que 2:1 aurait suffi. Montrez clairement comment vous avez obtenu votre réponse.

Appendix 3 : Speed reading: choosing between software and hardware caches

I've read several articles recently about optimising caches and they all seem to offer conflicting advice. As I understand it, my PC has two hardware caches and possibly two software caches. The Intel 25MHlz 486SX chip has a 128K memory cache. The Promise DC-4030VL card has a further 2Mb of disk cache. MS-DOS gives me SMARTDRV, and now I read that my Windows for Workgroups gives me something called VCACHE! Are all these caches doing the same job? Should I be using them all on one system? Is there a quick way to determine the optimum settings? Do I need them at all? Are other software caches any better than those supplied by Microsoft?

Malcolm Surgenor Falkirk, Scotland

> Probably the first thing we need to address is what a cache actually does, regardless of whether it's software or hardware based. In essence. a cache is a location in the computer's memory where small chunks of data are stored ready for retrievaL The idea is that these bits of memory can be accessed faster than the main store, be that a disk or RAM.

The most common software disk cache currently available, especially on Windows machines, is SMARTDRV.EXE. Windows for Workgroups comes with an updated version called VCACHE. SMARTDRVand VCACHE perform exactly the same function, but the former is compatible with both DOS and Windows, so needs to sit in DOS memory (around 28K of it). VCACHE, on the other hand, is a Windows VxD driver and requires no DOS memory, but will only work for DOS applications if they're running in the Windows environment. It's also worth noting that VCACHE can be used with the standard 3.1 edition of Windows.

In a software disk cache, information on the disk is stored in RAM, where it can theoretically be retrieved at around 100,000 times faster than from the disk. A cache is essentially a buffer between a slower device like a disk drive and faster one, such as the processor. As caches are typically smaller than the device they're caching—a typical SMARTDRVsetting would be 2Mb for a 240Mb hard disk— some logic is needed to establish what is stored in the cache. The most common method is to use the 'most frequently accessed' equation.

With this, the most frequently accessed pieces of data, up to the capacity of the cache, are stored. When the cache is full and some other piece of data not presently in the cache is loaded, then the least accessed of the data already in the cache is flushed and replaced by the new data. The down side of this is that the machine incurs the overhead of having to check to the contents of the cache with every transaction. But as the data read from the cache is delivered at an exponentially higher rate, the overall result is better throughput. As disk caches will always cache the directory, substantial performance increases can be gained in locating a given file alone

PC Magazine

Appendix 4 : Spin Doctoring

Access time is the most widely used indication of the speed of a hard drive. Access time is the sum of the *average seek time* - how long on average it takes the head to move to the correct track - and the *latency* - how long on average it takes the desired data on the correct track to move under

the head. (Advertisements often, deliberately or accidentally, confuse access time with seek time. They also usually quote the lower - faster - read-seek time rather than the higher - slower - write-seek time.)

The seek time depends on the size of the drive (e.g. 3.5 inch), the number of tracks per inch (tpi, which itself depends on such things as the size of the magnetic domains), and the speed and precision of the head actuators. The latency depends upon the spin rate: the rotational speed of the disk. Latency is half the time it takes for a complete rotation of the disk. The actual throughput also depends on the layout of the magnetic domains: You can pack more sectors in the tracks near the outer edge than on tracks nearer the center. This approach is referred to as *zone - bit recording.*

One way to improve the access time is to reduce this latency by speeding up the rotation of the disk (the spindle speed). Faster spin rates generally mean better performance. In the olden days (a few years ago), all desktop spindle speeds were the same: 3600 rpm. The resulting latency (time for half a rotation) was 8.3 milliseconds.

Top - speed hard drives for desktop PCs these days rotate at 5400 rpm, 50 percent faster, for a latency of 5.6 ms. Some current hard drives rotate at 4500 rpm, for a latency of 6.7 ms. Many hard drives for portable computers still use a rotation speed of 3600 rpm in order to consume less power.

Current hard drives for servers rotate at an even zippier 7200 rpm, twice as fast as the old brand. Their latency is 4.17 ms, half the old latency. This is especially significant for transaction - oriented servers, points out James Porter, president of Disk/ trend (Mountain View, CA). a company that monitors drive business and technology. Server administrators typically strive to put data that belongs together in the same area on a hard drive to reduce the effective seek times of drive access. Thus, their access time is especially sensitive to changes in latency. Reduce the latency significantly and you have a happy server administrator.

Byte

Appendix 5 : HOW INTERRUPTS WORK

Our daily lives are filled with asynchronous events that vie for our time, interrupting the orderly, sequential plan for the day. The telephone rings; there's a knock at the door; the baby cries for a diaper change. You can't predict their occurrence and plan them into your schedule, yet they must be accommodated. You could, of course, regularly check (or poll) for events—Is the phone ringing? Is someone at the door? Does the baby need to be changed?—but, clearly, that would be an inefficient use of time; it is better to let such asynchronous events capture your attention as they need to.

2 Similarly, your computer must respond to asynchronous events (e.g., keyboard presses, mouse movements, disk accesses, timer time-outs, and data communications). If the processor in your computer had to continually poll the various I/O devices, it would not be very efficient at doing the real work you ask of it. So to maintain efficient use of the processor's time, computers use interrupts to handle asynchronous events.

3 Like people, a processor executes instructions in a scheduled, sequential manner until an interrupt request (IRQ) occurs. When this happens, the processor drops what it's doing and services the interrupt, and then resumes sequential execution where it left off.

4 As for supporting interrupts, the conventional imple-mentation of today's PC systems is lacking in some areas, but certain key problems have been overcome in the newer EISA and Micro Channel expansion buses. Following is a detailed look at how PCs handle interrupts.

Interrupt Basics

5 There are three general types of interrupts that can occur in a PC: hardware interrupts, software interrupts, and processor exceptions. Hardware interrupts are the focus of this article, but I'll describe the others as well.

6 I/O devices electrically generate hardware interrupts to get the attention of the processor. The first PCs, of course, used Intel's 8088 processor, which has essentially the same functionality as the newer 286, 386, and 486 processors operating in real mode. All these processors have two pins that are used for interrupt purposes: INTR and nonmaskable interrupt (NMI).

Maskable Interrupts

7 INTR is the conventional interrupt input to the processor.

This interrupt input is maskable, meaning that it can be enabled or disabled under software control. An interrupt enable flag (IF) in the FLAGS register enables INTR interrupts when set and disables them when cleared. With interrupts enabled, when the INTR input goes high, the processor completes its current instruction and then responds to the IRQ with two successive interrupt acknowledge (INTA) cycles.

8 The first INTA cycle is essentially a dummy to ready the interrupting device for the second INTA cycle. During the second INTA cycle, then, the interrupting device must place an 8-bit interrupt-vector (sometimes called an interrupt-type) byte onto the data bus to further direct the processor's handling of the interrupt. In most systems, including PCs, a special IC called an interrupt controller interacts with the processor to place the interrupt vector on the data bus at the appropriate time.

9 When the processor receives the interrupt type from the interrupting device, it multiplies the value by 4 (by shifting it 2 bits to the left) to create an offset into the interrupt-vector table. This table—which contains 256 4-byte entries (1 KB total) starting at the very bottom of memory— holds the addresses of the service routines for the implemented interrupts. Note that a maximum of 256 distinct interrupts can be supported in this fashion.

10 The processor now retrieves the 4 bytes at the calculated offset in the interrupt-vector table to form a pointer to the interrupt-service routine; the pointer is in standard 80x86 segment:offset format. After pushing the FLAGS register onto the stack and clearing the IF bit in the FLAGS register, the processor begins to execute the ISR.

11 To keep problems from occurring after returning to the interrupted program, the ISR has to save any CPU registers that it uses and restore them when it is finished. An ISR generally terminates with an interrupt-return (IRET) instruction, which restores the FLAGS register from the stack (reenabling interrupts) and resumes program execution where it left off. The figure shows the steps that are involved in processing an interrupt after the interrupt-type byte has been received.

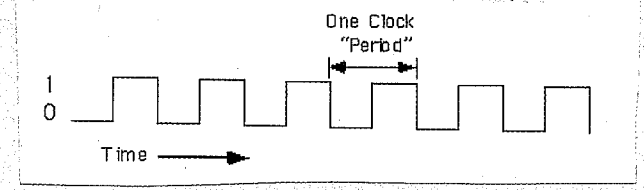
Byte

Appendix 6

Wait States

3.2.1 The System Clock

At the most basic level, the system clock handles all synchronization within a computer system. The system clock is an electrical signal on the control bus which alternates between zero and one at a periodic rate:



The frequency with which the system clock alternates between zero and one is the system clock frequency. The time it takes for the system clock to switch from zero to one and back to zero is the clock period. One full period is also called a clock cycle. On most modern systems, the system clock switches between zero and one at rates exceeding several million times per second. The clock frequency is simply the number of clock cycles which occur each second.

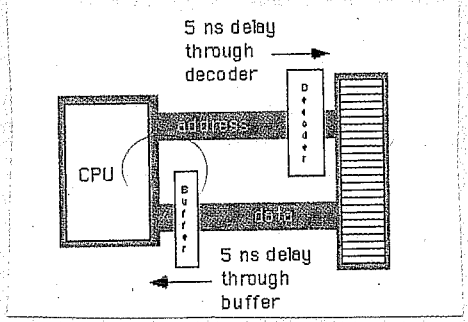
3.2.2 Memory Access and the System Clock

Memory access is probably the most common CPU activity. Memory access is definitely an operation synchronized around the system clock.

Memory access time is the amount of time between a memory operation request (read or write) and the time the memory operation completes. On a 5 MHz 8088/8086 CPU the memory access time is roughly 800 ns (nanoseconds). On a 50 MHz 80486, the memory access time is slightly less than 20 ns.

3.2.3 Wait States

A wait state is nothing more than an extra clock cycle to give some device time to complete an operation. For example, a 50 MHz 80486 system has a 20 ns clock period. This implies that you need 20 ns memory. In fact, the situation is worse than this. In most computer systems there is additional circuitry between the CPU and memory: decoding and buffering logic. This additional circuitry introduces additional delays into the system:

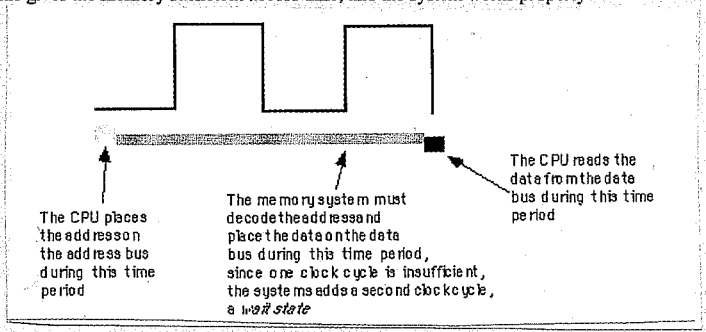


In this diagram, the system loses 10ns to buffering and decoding. So if the CPU needs the data back in 20 ns, the memory must respond in less than 10 ns.

You can actually buy 10ns memory. However, it is very expensive, bulky, consumes a lot of power, and generates a lot of heat. These are bad attributes. Supercomputers use this type of memory. However, supercomputers also cost millions of dollars, take up entire rooms, require special cooling, and have giant power supplies. Not the kind of stuff you want sitting on your desk.

If cost-effective memory won't work with a fast processor, how do companies manage to sell fast PCs? One part of the answer is the wait state. For example, if you have a 20 MHz processor with a memory cycle time of 50 ns and you lose 10 ns to buffering and decoding, you'll need 40 ns memory. What if you can only afford 80 ns memory in a 20 MHz system? Adding a wait state to extend the memory cycle to 100 ns (two clock cycles) will solve this problem. Subtracting 10ns for the decoding and buffering leaves 90 ns. Therefore, 80 ns memory will respond well before the CPU requires the data.

Almost every general purpose CPU in existence provides a signal on the control bus to allow the insertion of wait states. Generally, the decoding circuitry asserts this line to delay one additional clock period, if necessary. This gives the memory sufficient access time, and the system works properly



Sometimes a single wait state is not sufficient. Consider the 80486 running at 50 MHz. The normal memory cycle time is less than 20 ns. Therefore, less than 10 ns are available after subtracting decoding and buffering time. If you are using 60 ns memory in the system, adding a single wait state will not do the trick. Each wait state gives you 20 ns, so with a single wait state you would need 30 ns memory. To work with 60 ns memory you would need to add three wait states (zero wait states = 10 ns, one wait state = 30 ns, two wait states = 50 ns, and three wait states = 70 ns).

Needless to say, from the system performance point of view, wait states are not a good thing. While the CPU is waiting for data from memory it cannot operate on that data. Adding a single wait state to a memory cycle on an 80486 CPU doubles the amount of time required to access the data. This, in turn, halves the speed of the memory access. Running with a wait state on every memory access is almost like cutting the processor clock frequency in half. You're going to get a lot less work done in the same amount of time.

You've probably seen the ads. "80386DX, 33 MHz, 8 megabytes 0 wait state RAM... only $1,000!" If you look closely at the specs you'll notice that the manufacturer is using 80 ns memory. How can they build systems which run at 33 MHz and have zero wait states? Easy. They lie.

There is no way an 80386 can run at 33 MHz, executing an arbitrary program, without ever inserting a wait state. It is flat out impossible. However, it is quite possible to design a memory subsystem which under certain, special, circumstances manages to operate without wait states part of the time. Most marketing types figure if their system ever operates at zero wait states, they can make that claim in their literature. Indeed, most marketing types have no idea what a wait state is other than it's bad and having zero wait states is something to brag about.

However, we're not doomed to slow execution because of added wait states. There are several tricks hardware designers can play to achieve zero wait states most of the time. The most common of these is the use of cache (pronounced "cash") memory.

Appendix 7 : Man Page Interface for IRIX 5.2

The following is the man page for csplit(1):

csplit(1)

NAME

csplit - context split

SYNOPSIS

csplit [-s] [-k] [-f prefix] file argl [argn]

csplit(1)

DESCRIPTION

csplit reads file and separates it into n+1 sections, defined by the

arguments argl ... argn. By default the sections are placed in xxOOxxn

(n may not be greater than 99). These sections get the following pieces

of file:

00: From the start of file up to (but not including) the line referenced by argl.

01: From the line referenced by argl up to the line referenced by arg2.

n: From the line referenced by argn to the end of file.

If the file arqument is a -, then standard input is used.

csplit processes supplementary code set characters, and recognizes supplementary code set characters in the prefix given to the -f option (see below) according to the locale specified in the LC_CTYPE environment variable [see LANG on environ(5)]. In regular expressions, pattern searches are performed on characters, not bytes, as described on ed(1).

The options to csplit are:

—s

csplit normally prints the number of bytes in each file created. If the -s option is present, csplit suppresses the printing of all byte counts.

csplit normally removes created files if an error occurs.

If the -k option is present, csplit leaves previously created files intact.

-f prefix If the -f option is used, the created files are named prefixOO...prefixn. The default is xxOO...xxn. Supplementary code set characters may be used in prefix.

The arguments (argl...argn) to csplit can be a combination of the following:

/rexp/ A file is to be created for the section from the current line up to (but not including) the line containing the regular expression rexp. The line containing rexp becomes the current line. This argument may be followed by an optional + or - some number of lines (for example, /Page/-5). See ed(1) for a description of how to specify a regular expression.

%rexp% This argument is the same as /rexp/, except that no file is created for the section.

lnno A file is to be created from the current line up to (but not including) lnno. lnno becomes the current line.

{num}Repeat argument. This argument may follow any of the above arguments. If it follows a rexp type argument, that argument is applied num more times. If it follows lnno, the file will be split every lnno lines (num times) from that point.

Enclose all rexp type arguments that contain blanks or other characters meaningful to the shell in the appropriate quotes. Regular expressions may not contain embedded new-lines. csplit does not affect the original file; it is the user's responsibility to remove it if it is no loner wanted.

NAME

ed, red - text editor

SYNOPSIS

ed [-s] [-p string] [-x] [-C] [file]

red [-s] [-p string] [-x] [-C] [file]

DESCRIPTION

ed is the standard text editor. red is a restricted version of ed. If the file argument is given, ed simulates an e com-

mand (see below) on the named file; that is to say, the file is read into ed's buffer so that it can be edited. Both ed and red process supplementary code set characters in file, and recognize supplementary code set characters in the prompt string given to the -p option (see below) according to the locale specified in the LC_CTYPE environment variable (see LANG in environ(5)).

In regular expressions, pattern searches are performed on characters, not bytes, as described below.

-s Suppresses the printing of byte counts by e, r, and w commands, of diagnostics from e and q commands, and of the ! prompt after a !shell command.

-p Allows the user to specify a prompt string. The string can contain supplementary code set characters.

-x Encryption option; when used, ed simulates an X command and prompts the user for a key. This key is used to encrypt and decrypt text using the algorithm of crypt(1). The X command makes an educated guess to determine whether text read in is encrypted or not. The temporary buffer file is encrypted also, using a transformed version of the key typed in for the -x option. See crypt(1). Also, see the NOTES section at the end of this reference page.

-C Encryption option; the same as the -x option, except that ed simulates a C command. The C command is like the X command, except that all text read in is assumed to have been encrypted.

ed operates on a copy of the file it is editing; changes made to the copy have no effect on the file until a w (write) command is given. The copy of the text being edited resides in a temporary file called the buffer.

There is only one buffer.

red is a restricted version of ed. It allows only editing of files in the current directory. It prohibits executing shell commands via !shell command. Attempts to bypass these restrictions result in an error message (restricted shell).

Both ed and red support the fspec(4) formatting capability. After including a format specification as the first line of file and invoking ed with your terminal in stty -tabs or stty tab3 mode (see stty(1)), the specified tab stops are automatically used when scanning file. For example, if the first line of a file contained:

Page 1

ed(1) ed(1)

<:t5,10,15 s72:>

tab stops are set at columns 5, 10, and 15, and a maximum line length of 72 is imposed. When you are entering text into the file, this format is not in effect; instead, because of being in stty -tabs or stty tab3 mode, tabs are expanded to every eighth column.

Commands to ed have a simple and regular structure: zero, one, or two addresses followed by a single-character command, possibly followed by parameters to that command. These addresses specify one or more lines in the buffer. Every command that requires addresses has default addresses, so that the addresses can very often be omitted.

In general, only one command can appear on a line. Certain commands allow the input of text. This text is placed in the appropriate place in the buffer. While ed is accepting text, it is said to be in input mode. In this mode, no commands are recognized; all input is merely collected. Leave input mode by typing a period (.) at the beginning of a line, followed immediately by pressing RETURN.

ed supports a limited form of regular expression notation; regular expressions are used in addresses to specify lines and in some commands (for example, s) to specify portions of a line that are to be substituted. A regular expression specifies a set of character strings.

A member of this set of strings is said to be matched by the regular expression. The regular expressions allowed by ed are constructed as follows:

The following one-character regular expressions match a single character:

1.1 An ordinary character (not one of those discussed in 1.2 below) is a one-character regular expression that matches itself.

1.2 A backslash (\) followed by any special character is a one-character regular expression that matches the special character itself. The special characters are:

a. ., *, [, and \ (period, asterisk, left square bracket, and backslash, respectively), which are always special, except when they appear within square brackets ([]; see 1.4 below).

b. ^ (caret or circumflex), which is special at the beginning of a regular expression (see 4.1 and 4.3 below), or when it immediately follows the left of a pair of square brackets ([]) (see 1.4 below).

c. $ (dollar sign), which is special at the end of a regular expression (see 4.2 below).

d. The character that is special for that specific regular expression, that is used to bound (or delimit) a regular expression. (For example, see how slash (/) is used in the g

Page 2

ed(1) ed(1)

command, below.)

1.3 A period (.) is a one-character regular expression that matches any character, including supplementary code set characters, except newline.

1.4 A non-empty string of characters enclosed in square brackets ([]) is a one-character regular expression that matches one character, including supplementary code set characters, in that string. If, however, the first character of the string is a circumflex (^), the one-character regular expression matches any character, including supplementary code set characters, except newline and the remaining characters in the string. The ^ has this special meaning only if it occurs first in the string. The minus (-) can be used to indicate a range of consecutive characters, including supplementary code set characters; for example, [0-9] is equivalent to [0123456789]. Characters specifying the range must be from the same code set; when the characters are from different code sets, one of the characters specifying the range is matched. The - loses this special meaning if it occurs first (after an initial ^, if any) or last in the string. The right square bracket (]) does not terminate such a string when it is the first character within it (after an initial ^, if any); for example, []a-f] matches either a right square bracket (]) or one of the ASCII letters a through f inclusive. The four characters listed in 1.2.a above stand for themselves within such a string of characters.

The following rules can be used to construct regular expressions from one-character regular expressions:

2.1 A one-character regular expression is an regular expression that matches whatever the one-character regu-

lar expression matches.

2.2 A one-character regular expression followed by an asterisk (*) is a regular expression that matches zero or more occurrences of the one-character regular expression, which can be a supplementary code set character. If there is any choice, the longest leftmost string that permits a match is chosen.

2.3 A one-character regular expression followed by \{m\}, \{m,\}, or \{m,n\} is a regular expression that matches a range of occurrences of the one-character regular expression. The values of m and n must be non-negative integers less than 256; \{m\} matches exactly m occurrences; \{m,\} matches at least m occurrences; \{m,n\} matches any number of occurrences between m and n inclusive. Whenever a choice exists, the regular expression matches as many occurrences as possible.

2.4 The concatenation of regular expressions is an regular expression that matches the concatenation of the strings matched by each component of the regular expression.

Page 3

ed(1) ed(1)

2.5 A regular expression enclosed between the character sequences \(and \) defines a sub-expression that matches whatever the unadorned regular expression matches. Inside a sub-expression the anchor characters ((^) and ($)) have no special meaning and match their respective literal characters.

2.6 The expression \n matches the same string of characters as was matched by an expression enclosed between \(and \) earlier in the same regular expression. Here n is a digit; the sub-expression specified is that beginning with the n-th occurrence of \(counting from the left. For example, the expression ^\(.*\)\1$ matches a line consisting of two repeated appearances of the same string.

A regular expression can be constrained to match words.

3.1 \< constrains a regular expression to match the beginning of a string or to follow a character that is not a digit, underscore, or letter. The first character matching the regular expression must be a digit, underscore, or letter.

3.2 \> constrains a regular expression to match the end of a string or to precede a character that is not a digit, underscore, or letter.

A regular expression can be constrained to match only an initial segment or final segment of a line (or both).

4.1 A circumflex (^) at the beginning of a regular expression constrains that regular expression to match an initial segment of a line.

4.2 A dollar sign ($) at the end of an entire regular expression constrains that regular expression to match a final segment of a line.

4.3 The construction ^regular expression$ constrains the regular expression to match the entire line.

The null regular expression (for example, //) is equivalent to the last regular expression encountered.