From Punched Cards To Flat Screens

A Technical Autobiography

Philip Hazel

From Punched Cards To Flat Screens

Author: Philip Hazel

Revision 0.03   11 August 2017

Contents

Preface

When I retired at the end of September 2007, I knew I would be expected to make a speech at the retirement lunch. Looking back over the 40 years that I had been part of the computing community in Cambridge, it struck me again just how much change there had been, and also how many different areas of computing I had worked in. For the younger colleagues who listened to my speech, the early years must seem like ancient history.

A few minutes is a short time in which to summarize almost half a century, and afterwards I decided to write a longer account. This memoir is the result. It is a set of personal recollections of computation and computers and things that happened in connection with them, even if only loosely. I have not attempted to include detailed descriptions of the machines' hardware, as I was never a hardware person. Most of the technical detail is about software – after all, I was primarily a software developer – and inevitably it covers only the software that I wrote or used or had some dealings with.

This is definitely *not* a history of the University of Cambridge Computing Service. When I first joined this fledgling organization, many of the staff were software developers, and I knew more or less what everybody was doing, but as we got bigger I knew less and less about areas other than my own specialities. By the time I retired, I think I was probably the only member of the Service who was still developing software full time.

2017: *The first version of this memoir was published online in December, 2009. In 2017, ten years after I retired, I have revisited it and added a few further comments in paragraphs like this. The orginal text remains unaltered, apart from some minor text corrections.*

Acknowledgements

I received many useful comments on the first draft of this document from Chris Cheney, Tony Stoneley, Richard Hallas, Richard Parkins, Jeffrey Friedl, Peter Linington, John Line, and Mike Challis. They corrected some of my errors and mis-rememberings, and reminded me of things I had forgotten. David Hartley put me in touch with the ICT 1301 resurrection project at the Computer Conservation Society. Judith Hazel, my wife, cast her copy-editor's eye over the text for typos and other infelicities. Any errors that remain are, of course, my own.

Illustration credits

I am grateful to the following people and organizations for granting permission to include their photographs in this book: Jo Edkins for the log tables, from her website **http://gwydir.demon.co.uk/jo/index.htm**; Derek Ross for the slide rule, from his web site **http://www.antiquark.com/sliderule/sim**; John Wolff for the Facit mechanical calculator, from his web site, *which is now (2017)* ***http://www.johnwolff.id.au/calculators/***; The ICT 1301 resurrection project of the Computer Conservation Society for the ICT 1301 (their web site is **http://ict1301.co.uk/1301ccsx.htm**); The Computer Laboratory, University of Cambridge, for the photograph of the Titan in its original room from **http://www.cl.cam.ac.uk/Relics/archive_photos.html**, reproduced by permission under the Creative Commons licence; A *Flickr* user called Twylo for the Flexowriter, which is also reproduced under the Creative Commons licence; Richard Hallas for the Acorn desktop.

Mike Challis kindly sent me scanned images of his copy of System Planning Document 203. I found the photos of the lineprinter and the keypunch in web pages whose authors told me that they were ‘grabbed from the web’ without any record of their attribution. I have not been able to find their origins.

The remaining photographs were taken by myself.

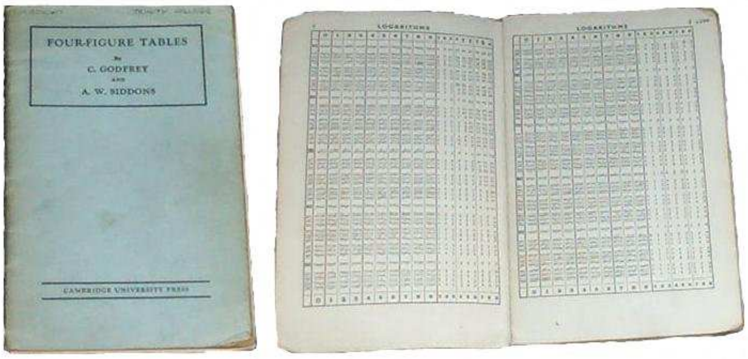
1. Early days

It is said that anything invented before you are three years old is a normal part of life; anything invented before you are thirty-three is new and exciting, and you can probably make a career in it. Subsequent inventions are just new-fangled fripperies that you can well do without. For me, computers fit neatly into the second category. They were being invented soon after I was born, though I did not become really aware of them until near the end of my under-graduate career at the University of Cape Town (UCT).

My family had emigrated from England to South Africa after the second world war, ending up in the small town of Kuruman in the Northern Cape, where my father worked in the offices of an asbestos-mining company. At the age of eight, I was sent to boarding school in Cape Town, a 24-hour bus and train journey each way. This was not so bad compared to the multi-day train journey the children from what was then Northern Rhodesia (now Zambia) had to endure. Some of them used to get off the train and hitch-hike for a few hours, rejoining the train later, just to relieve the boredom. I made the round trip four times a year for ten school years, then twice a year for the four years I was at UCT.

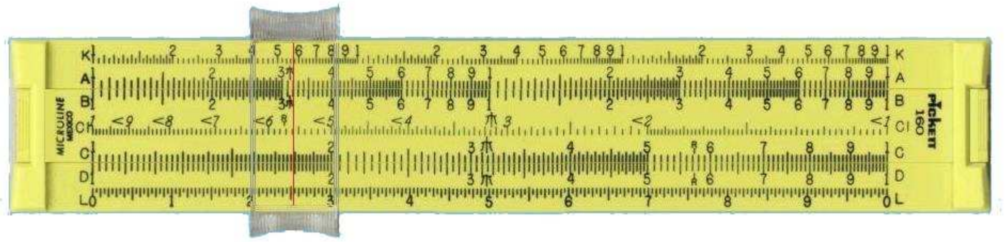
There were, of course, no digital calculators in the 1950s. In primary school we learned to do arithmetic by hand, including long division and such esoteric processes as extracting square roots. All the problems in the arithmetic text books were set with ‘easy’ numbers so that they could conveniently be worked by hand. South Africa’s currency at that time was still pounds, shillings, and pence (decimalization did not happen until February 1961, ten years to the day before Britain decimalized its currency), so we spent much time adding and subtracting in bases 12 and 20. My father could add up a column of pounds, shillings, and pence in his head, something I never managed to emulate.

On arrival at secondary school we were issued with books of four-figure mathematical tables of trigonometric functions and logarithms, commonly called ‘log tables’. We spent many lessons learning how to do non-exact arithmetic using these tables, and soon the arithmetic problems we were set could no longer be done entirely by hand.



Four-figure log tables

After I left school and went to UCT, I had to learn how to use a slide rule, a device that is now so unknown that I recently saw ‘slide rule’ printed in quotation marks. A slide rule is in effect a hardware implementation of log tables, but it is only as accurate as your eyesight and finger control. Fortunately, I never had to do much serious calculation with a slide rule.



A slide rule

In 1962/3 I spent what would now be called a ‘gap year’ as an exchange student at Towson High School, in Baltimore, USA. Because of the mis-alignment of academic years in the northern and southern hemispheres, I had to split my first year at UCT, completing the first half (January to June) before going to the States, and the second half after coming back. Returning to school after six months as an undergraduate was a bit weird, but I enjoyed the whole experience very much. None of the courses I took were for examination or any kind of credit, and I studied no mathematics or physics at all. However, to fill in an empty slot in my timetable I took a course in touch typing, which, as it turned out, was highly relevant to my future career.

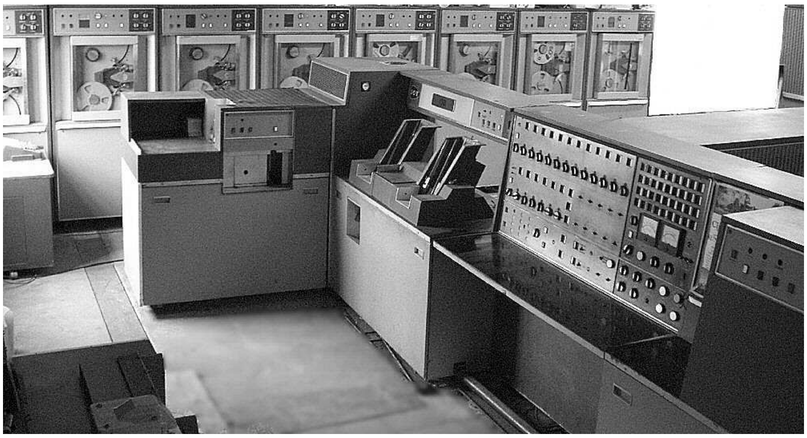
Back at UCT, in the longer vacations, I worked for my father’s mining company, sometimes in the electrical and mechanical workshops, and sometimes in the offices, where I first encountered mechanical calculators. Hand-cranked machines, such as those made by Facit and Brunsviga, were then widely used for repetitive calculation, even in the wilds of Africa. There was great excitement when the first electro-mechanical calculator, costing several hundred pounds (the price of a car in the 1960s), arrived. Not only could it do arithmetic without your having to turn a handle, but it also printed the calculation on a roll of paper as it worked, so you had a record of what had been done. I spent some time investigating all its features (which by today's standards were very primitive), but it was rather wasted on the permanent office staff, who mostly continued to do their sums the way they always had. Resistance to innovation is nothing new.

A Facit mechanical calculator

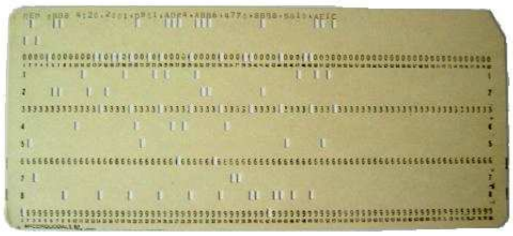
Towards the end of my third year at UCT there was great excitement in the applied mathematics department when the first computer arrived. This was an ICT 1301, to be housed in the department, but used by both academics and the administration. (ICT, *International Computers and Tabulators*, was a British company that later merged with some others to become *International Computers Limited* or ICL. It was subsequently bought by Fujitsu.)

One of the lecturers must have had some kind of training, because as soon as the machine was installed he ran a series of introductory programming lectures, open to all. I went along, and was introduced to Manchester Autocode, the programming language we were all to use. Looking back, I suppose it must have been possible to program the 1301 in some kind of assembler or machine code, but nothing was ever said to us about that kind of programming.

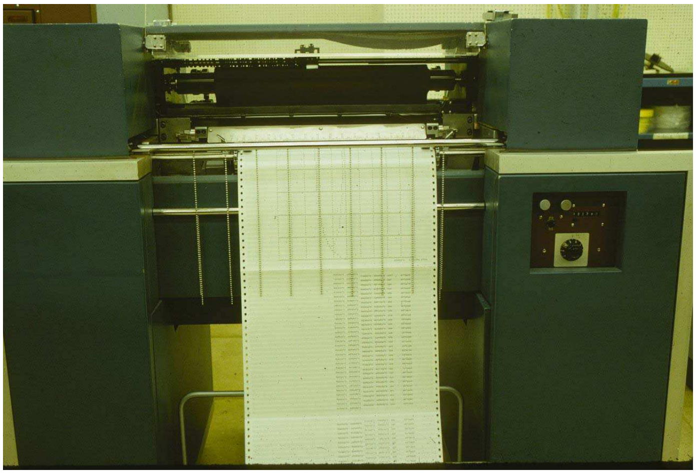
After the first lecture, I read the Autocode programming manual from cover to cover, and found that I easily understood all of it. At the time I thought nothing of this, but I subsequently discovered that reading computer manuals was not the way most people liked to learn. Later on I picked up many other programming languages just by reading the manuals and writing test programs.


An ICT 1301 computer at the CCS resurrection project in 2009

The ICT 1301 was a one-user-at-a-time computer, made of printed circuit boards with discrete transistors. It occupied quite a large room. Input was on punched cards, and output was printed by a lineprinter, on so-called ‘piano paper’, large sheets ruled with faint background lines and shading. These came joined together in a continuous run, with sprocket holes down the sides, and perforations so that you could tear off the individual sheets.


A punched card

Students could go along to test sessions and stand in line for their turn to use the machine, having first written a program by hand and then taken this to a keypunch machine to punch the cards. You put your cards in the hopper, pressed a sequence of buttons, and off it went. If you were lucky and your program worked, the output appeared on the printer; more often that not (at least at the start) all you got was an error message, which meant that you had to fix your program, return to the back of the queue, and hope to get another run before the end of the session. The more attempts it took to get a program to work, the more useful scrap paper one accumulated.


A lineprinter outputting on piano paper

In my third and final years at UCT, I wrote only a few exercises and test programs, but, unknown to me at the time, this laid the foundations for my future career. It might have been different: the mother of one of my erstwhile schoolfellows was an ornithologist who had amassed a lot of observational data, and he and I talked about what a computer could perhaps do for her, but having no experience of programming anything other than calculations, we did not have the vision to invent a database.

2. A research student at Cambridge

In January 1967 I arrived in Cambridge to study for a PhD in fluid dynamics, in the Department of Applied Mathematics and Theoretical Physics (DAMTP). My arrival was made easier by the fact that a good friend, Raymond Pollard, who had been at school and UCT with me, had done exactly the same thing a year earlier (he had forgone a ‘gap year’). We both became members of St John’s College. Raymond had applied there because it was his father’s College, and I followed his lead. A third member of our school year, Peredur Williams, followed a year or two later to study Astronomy, having done an MSc at UCT first. Three people from the same school year ending up at the same Cambridge College must be some kind of a record for our school in Cape Town, I feel.

I was financed by a Smuts Trust scholarship from South Africa, out of which I had to pay both my fees and my living expenses, and I was conscious of having to budget very carefully. For the first year I kept very detailed accounts, and I remember that the total cost was almost exactly £1000. Recently I saw an estimate of £20,000 for a non-EU research student’s annual cost. The factor of 20 increase correlates closely with the cost of a pint of beer, which has risen from around 2/6 (two shillings and six pence, one eighth of a pound, or 12.5p in new money) to around £2.50 in the intervening years.

2017: *Both the cost of beer and the cost of being a student have now of course increased even more.*

As soon as I arrived, I investigated the possibility of doing some undergraduate supervision to supplement my income, and was immediately snapped up for the Part II fluid dynamics course, for which supervisors were in short supply. This was a piece of good luck. The fluid dynamics lecturer at UCT had been at Cambridge, and had cribbed his examples and exercises from the Cambridge course. I had written his final exam only a couple of months before, so supervising that material was really easy. I also picked up some Part I supervising, and what with that and Harold Wilson’s devaluation of the pound in November 1967, I did not run short of money.

My own supervisor, Francis Bretherton, set me to work on a problem involving stratified shear flows, that is, fluid flows in which the density varies from top to bottom. The problem had practical relevance to both the ocean and the atmosphere. I spent some time settling into DAMTP, attending lectures, reading the background, and then struggling with the differential equations, trying to find a way of obtaining an analytical solution. In this I had no success. When some months had passed, Francis suggested that perhaps I should try solving the equations numerically, using the University’s then quite new Titan computer.

There were probably no more than four computers in the whole University at around that time. As well as the Titan, there was an IBM 1130 in the Engineering department, a small IBM 360 out at Astronomy, and a PDP-7 which was used for graphics research by the computer scientists, though the phrase ‘computer science’ was not yet in common use at Cambridge.

The Titan

I went along to the Mathematical Laboratory (forerunner of today’s Computer Laboratory, and always known as ‘the Maths Lab’), then housed in the old Anatomy building on Corn Exchange Street. I was sent to see Peter Hammersley, who was in charge of user accounts. It was really the wrong time of year to be starting. All the programming courses happened at the beginning of the academic year, and new users normally had to attend a course in order to learn how to use the machine. However, I was lucky. I told Peter that I knew Manchester Autocode, and that was good enough because of its similarity to Titan Autocode. I was sent away with an account number and an Autocode manual, and that was the start of the slippery slope down which I have been sliding ever since.

Learning to use the Titan

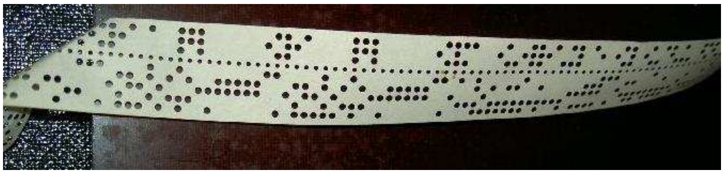
The Titan was a prototype for the Ferranti Atlas II, but only two others were ever commissioned. The operating system (then called the ‘supervisor’) was developed locally at Cambridge, and it supported multiprogramming. That meant that the system could run several different programs ‘simultaneously’, a relatively new concept at the time, and of course completely novel for me. The users were kept well away from the machine, which was run by operators who worked round the clock in three shifts, except at weekends.

When I became a user, a time-sharing online service was just coming into existence, though I was not aware of this till later. In those days, you had to show yourself to be a competent user of the offline (batch) service before you were allowed to go online. (Several years later, when new users were first allowed to use the IBM mainframe online service without having served an offline ‘apprenticeship’, it seemed rather scandalous to the old hands.)

Input to the Titan was mainly on punched paper tape. By the time I arrived, 7-track tape was the norm. This supplanted the 5-track tape that had previously also been used on the EDSAC computers, before my time. The great advantage of 7-track tape, apart from an increased character set, was that the typewriter-like *Flexowriters* that punched the tape also printed a transcript on a roll of paper as you operated their keys. The old 5-track punches (which I never used) just produced a tape, with no confirmation of what you had typed. You had to run 5-track tape through a separate machine to get a printout. The Titan could also read 8-track tape, which used the ASCII code that had recently been standardized.

A Flexowriter

At first, I thought that paper tape was an inferior input medium to the punched cards I had used at UCT. If you made a mistake, you had to make a whole new tape instead of just replacing one card in your deck. However, the compactness of paper tape soon won me over. A program that required a whole box of cards (somewhat bigger than a shoebox) could be put on a paper tape that would fit in a shirt pocket, and if you dropped it, it didn’t get shuffled. Furthermore, various aids were available to help correct mistakes.


Punched paper tape

If you noticed that you had pressed the wrong key while typing, you could press ‘backspace’ and ‘erase’ to remove the incorrect character. This did not actually backspace the tape; instead it punched special codes for ‘backspace’ and ‘erase’. When the tape was read in, a process called ‘line imaging’ took place, in which these codes were interpreted, leaving a cleaned-up line for subsequent processing.

Once you had your program on tape, you made another short tape called a ‘job description’, specifying how the program was to be processed; this was a crude forerunner of what today would probably be called a ‘script’. You put the tapes in a small plastic bag, together with a paper ticket containing your details, and hung the bag on a pegboard outside the computer room. Then you went away and did something else for a few hours.

The bags containing the paper tapes were taken off the ‘job queue’ by the computer operators and fed into the Titan, which stored the data on disc before running the job. The discs themselves were physically large, the platters being about two feet across, though they held very little data by today’s standards. This use of discs for ‘spooling’ input (and, similarly, output) meant that a running program could read and write data quickly, without having to wait for slow, human-operated peripherals, as was the case with other computers of that time. For the paper tape readers, ‘slow’ is a somewhat relative term, as they were capable of reading 1000 characters per second. Incidentally, the word ‘spooling’, derived from ‘simultaneous peripheral operations online’, was not in use in those days. It came in later with the IBM mainframe.

If you were lucky, your output was waiting when you returned to the Maths Lab, in your slot in the ‘output tanks’. These were long tanks which held a suspended file for each user, into which printout and your bag of input paper tapes could be placed. The Titan did not use ‘piano paper’, which was regarded with disdain; plain white was preferred. Also, a full set of both upper and lower case letters was available on the printer, in contrast to many lineprinters of that era, which had only upper case. (A reader of the first draft of this book remembered there being an ‘old’ printer that had only upper case, but I think this must have been superseded by the time I arrived.) The Titan also supported a pen plotter for producing graphical output, though I did not use that till later.

2017: *I used the pen plotter for some graphs for my thesis. When it came time to write up, I discovered that a one-inch margin was required on all pages, and my plots did not leave sufficient space on quarto-sized paper, which was still in common use. Luckily for me, I found out that the regulations had recently been changed to allow theses to be submitted on the then new standard A4 paper size, which was a bit wider, and this saved me from having to re-do all my graphs.*

Needless to say, after your first run of a program, the only printout was often a programming error message. When my officemate in DAMTP ran his first program, all he got back was a sheet of paper saying ‘invalid carriage control character’, which confused him totally because he couldn’t understand why an ‘invalid carriage’ (an early type of mobility scooter) was mentioned. The message was actually referring to an incorrect ‘carriage control character’, one of the special character codes that controlled the carriage of a mechanical output device. Obscure computer error messages have been around for a long time.

To make a revised paper tape you could use a Flexowriter to copy the old tape up to the point of error, type in the correction, and then copy the rest. However, an easier way was provided by the early text editors. By the time I arrived, a program called E3 was in general use. As part of the job description you could request the use of E3 and include simple editing commands such as ‘delete line 42’ or ‘insert these lines before line 99’. The edits were then applied to your program before it was obeyed. What is more, you could also request that a line-numbered printout and a new paper tape be produced automatically, so you had a fresh listing and up-to-date tape for the next run. Programming modifications were typically done by scribbling on the printout before creating the new editing tape.

With luck, a user could perhaps get three runs of a program in a working day, though not at busy times. One cause of turnaround delay was the fairly frequent hardware breakdowns suffered by the Titan. The engineering team was always on hand to mend it, but repairs could take several hours. The faster turnaround out of working hours was the main reason for the night-time user population at the Maths Lab. Friday night fish-and-chip suppers with the operators in the common room were a regular feature. When the tea ladies complained about grease on the counter, one enterprising user wrote a program called ‘tablecloth’, which printed a tasteful design of fishes on several sheets of lineprinter paper.

Another programming language

I was lucky in that the Autocode programs I developed for my initial research turned out well, and I was able to write up the results quite quickly. Then I moved on to other problems in the same general area, which required some new programs.

About this time, a FORTRAN compiler (written by John Larmouth) became available for use on the Titan. When the Titan software system was originally designed, the idea was that all user programs would be written in a single language, which would be suitable for every kind of problem. The language was called ‘Combined Programming Language’ (CPL), and it was being jointly developed by Cambridge and the University of London.

This was very ambitious for the machines of the time, and CPL was never completed, though its design influenced later languages, in particular, BCPL and C. There were several subsequent attempts to impose a ‘one language fits all’ philosophy on the world: IBM tried with PL/I in the 1970s, and Java was touted as a universal panacea in the 1990s. Needless to say, in reality, no single language ever suits all applications, or indeed all programmers.

When it became clear that CPL was not going to be available in time for use on the Titan, a rudimentary assembly language called IIT (‘Intermediate Input for Titan’) was made available, and a compiler for Autocode, based on EDSAC 2 Autocode, was hastily created. Meanwhile, out in the wider world, people realized the need for standardized languages that could be used on many different computers. Until that time, each make and even model of computer used its own language, so changing computers meant rewriting all your programs. In 1966, FORTRAN, which was designed for scientific calculations in the 1950s, was the first programming language to be formally standardized by what was then the American Standards Association (ASA), now the American National Standards Institute (ANSI).

I knew none of this at the time, but when a FORTRAN compiler was announced for the Titan, I took a look at the language and decided that it was more suitable for my research problems than Autocode. Before standardization, there were many dialects of FORTRAN, because each implementor added their own pet features to ‘improve’ the language. True to form, the first Titan compiler supported a local dialect called T3 FORTRAN. However, it was soon realized that support for the standard version would be necessary, and so the compiler was modified to support ASA FORTRAN, and T3 FORTRAN was phased out.

The rest of my PhD research was done in FORTRAN. Using it on the Titan involved learning a new scripting language called MLS (‘Mixed Language System’). This introduced me to the concept of independently compiling different modules, which could be written in different languages, and then linked together to form the final executable program.

Going online

After I had been using the Titan for about a year, I was able to get an account for timesharing access. The terminals were Teletype model 33s, running at 110 baud (bits per second), and printing onto rolls of paper. They were extremely noisy devices, hugely primitive and slow compared with what came later, but very exciting at the time. The only public terminals were in Room B in the Maths Lab, which became something of a social venue for the geeks of the day (and night), despite the continual clatter of the Teletypes. If you had a problem, you could just lean over and ask whoever was next to you for help.

A model 33 Teletype

When I first logged on to the Titan Multiple-Access System, the impression that the big expensive computer was sitting idle, waiting for me to type, was so strong that the session was rather manic. Even after learning that it was OK to sit and think for a bit, one could not be idle for too long in Room B, because there were often users waiting to use one of the eight terminals (called ‘consoles’ in those days).

Because of memory limitations, and the lack of paging or swapping, the online facilities that were available to normal users were quite restricted. The text editor, for example, could only move forwards through a document because it held only one line at a time in main memory. You could, however, ‘rewind’ to the start; this involved copying the rest of the document and then opening the output as a new input. (Incidentally, computer memory was usually called ‘store’ in the UK at that time, but the American usage seems to have taken over nowadays.)

The online editor, unimaginatively called ‘edit’, was written by Steve Bourne, later to be the author of the Bourne Shell for Unix. Editing was line-by-line; the current line was printed on the terminal, and the editor then responded to commands to modify the line, or to move forwards to another line by number or by context search, the latter meaning that you no longer needed to maintain an up-to-date line-numbered printout of your program.

Only a few privileged programs such as the text editor and the ‘command program’ (which nowadays would be called a ‘shell’) were permitted to interact directly with a user at a terminal. These programs were written in such a way that a single copy could be shared by all the users, so their memory usage was minimized. User programs had to read their input from, and write their output to, disc. In simple cases, output was written to a temporary file which was automatically copied to the terminal once the program had finished and relinquished its working memory. This ‘normal mode’ gave an illusion of interaction in cases where the input was short and could be completely provided at the start of execution.

Despite its limitations by today’s standards, Titan timesharing was a huge advance at the time. As soon as I started using it, my program development speeded up considerably, and I rapidly began to fill up my allotted amount of disc file storage. This meant that I had to start making archives on magnetic tape. The Titan’s tapes were one inch wide, and were pre-blocked, meaning that you could jump around on the tape and treat it rather like a very slow disc. In particular, it was possible to update earlier blocks without affecting later ones. This facility provided the mechanism for a tape archiving system (written by Robin Fairbairns) where an index to the tape’s contents was kept in specific blocks at the start of the tape. This made it easy to manage the files that were on the tape.

2017: *At weekends, the Titan was available for system development by ‘authorized users’. When a dedicated machine was not needed, they often allowed general access. This was another good time to get work done, but there were no operators to load and unload magnetic tapes. I managed to get myself trained as an authorized tape deck user so that I could do it myself.*

A different way of coping with limited file space was provided by Peter Linington (then a research student in Physics), who implemented *microlib*, which was a way of storing many small files in a single ‘real’ file – a precursor of today’s file archiving programs.

Some time in 1968 a Teletype connected to the Titan was installed in DAMTP. I think this was the first terminal that was situated outside the Maths Lab. As there were few other Titan online users in DAMTP at that time, I was able to monopolize it to quite a large extent.

The summer of ’68

In the middle of 1968 I spent some time at the Woods Hole Oceanographic Institution in Massachusetts, attending a summer school in geophysical fluid dynamics. I did not make much headway with a ‘summer problem’ that I was given to tackle, showing again that mathematical analysis was not my strong point. Still, there were interesting people there to talk to, and we did some educational experiments with fluids on a rotating turntable. (We also tried sitting on the table to see if one could feel the Coriolis force, but all that happened was that you became rather dizzy.)

Just up the road, at MIT in Boston, Richard Parkins, a Cambridge undergraduate whom I knew, was spending the summer in the computer department. MIT had pioneered timesharing with CTSS (the *Compatible TimeSharing System*) which, when he saw it demonstrated, had caused Maurice Wilkes to push for time-sharing on the Titan. We found out that there was a Teletype in Woods Hole that could be connected to MIT by a phone line, and slaved to a terminal there. By this means, Richard was able to give me a CTSS demonstration.

Yet more programming languages

Though my ‘day job’ programs were now written in FORTRAN, I picked up manuals for several other programming languages while hanging around the Maths Lab, and taught myself how to use them. The first of these was IIT, the original assembly language for the Titan. Instructions were written numerically, labels were just numbers, and there were none of the extended features such as macro support that one finds in today’s assemblers. For example, to load register 2 from a memory location labelled 9, then add 42, putting the result into register 3, one would write this:

```
101   2   0   (9)
121   3   2   42
```

The Titan had 128 index registers (called ‘B-lines’), and though some of them were reserved for special purposes (B127 contained the program counter, for example), there were so many available for general use that many programs could keep all their most important integer data in registers. (Other computers of the time had far fewer registers; the IBM 7090 had only 7.) Programs written in IIT ran faster than any written in a compiled language. It would be many years before compiler technology improved to the extent of rivalling hand-crafted assembler programs. I wrote some utility software in IIT, and also a few ‘fun’ programs. The first program of mine that was made publicly available was called *info*; it output various pieces of information about the system and the programming environment.

Frank King, then a research student in Engineering, had written a program called *MOO*, an implementation of the number guessing game ‘Bulls and Cows’, and Titan users strove to reach the top of its league table. As well as being an entertaining game, the program was a demonstration of the security features of the Titan; anybody could run the program, but only when it was running could the file that held the league table be updated. Inevitably, someone wrote a program to play the game, and this topped the league. I spent some time on a Christmas holiday back in South Africa trying to do the same, but although my program worked when I got back to Cambridge, its algorithms were poor and it did no better the human players. I do not suppose anybody else wrote IIT (on paper, of course) that far away from Cambridge.

When Martin Richards returned to Cambridge from MIT in 1968, he brought back his implementation of a compiler for BCPL, a small language that he had derived from CPL by cutting out all the ‘difficult’ features. An implementation for the Titan became available, and I started to play with the new toy. BCPL was the first block-structured language I had encountered. It was also free-format, a welcome escape from the rigours of FORTRAN. After a while I settled on a program layout that I liked, and I am still using it forty years later, though with different languages. Unfortunately, many later programmers adopted a different style, which makes my code hard to read for some people. Here is a typical example of my layout, taken from the PCRE library (which is written in C):

```
if (repeat_min == 0)
  {
  if (repeat_max == -1) *code++ = OP_STAR + repeat_type;
    else if (repeat_max == 1) *code++ = OP_QUERY + repeat_type;
  else
    {
    *code++ = OP_UPTO + repeat_type;
    PUT2INC(code, 0, repeat_max);
    }
  }
```

The ML/I language was a different kind of animal altogether. It was a macro processor written by Peter Brown, a precursor to later processors such as *m4*. I did not have occasion to use ML/I for any major projects, but it taught me a lot about naming concepts and in particular the difference between a literal name and a name that stood for something else. Suddenly the part of *Alice Through the Looking-glass* where the White Knight is explaining the difference between what the song 'is called' and what it 'is' made a lot more sense.

Other Titan interests

The Titan system was continuously developing, and I was keen to keep up with all the developments. By this time I was known in the computing community, and at some point I started doing duty as the 'programming adviser', who provided the HelpDesk service of that era. New facilities were regularly released, but already there was a body of users who were resistant to change, and who had to be helped. One big issue was the conflict in two different scripting systems. The MLS system, originally developed to manage FORTRAN, had a different philosophy to the command system used for interactive access. When a new 'command program' that incorporated both ways of working was implemented by Mike Guy, some users' scripts had to be rewritten for them, so that MLS could be retired.

One facility of the Titan that I never used was its data link to a PDP-7 computer with a cathode-ray tube display that was mainly used by the computer scientists for graphics research. The PDP-7 was made by Digital Equipment Corporation (DEC), an American computer company that produced a succession of computers under the name PDP (Programmable Data Processor). The PDP-7 had no disc storage, but the link made it possible to store programs and data on the Titan's discs. One piece of 'research' was a two-person shoot-em-up spaceships game called *Duel* that I did once play, late at night. You had to control your 'spaceship' (a circle with a short line for its gun) by toggling three or four small switches on the computer's control panel; there were no computer mice in those days. The PDP-7 generated radio signals as it operated; if a nearby transistor radio was tuned to the right frequency, you could hear a series of tones. Special programs were written that did not actually do anything useful, but obeyed suitable sequences of instructions so that what came out of the radio were musical tunes.

The computer scientists were not the only people who used the PDP-7. From the earliest days of the first EDSAC computer anybody in the University who had a good case had always been encouraged to make use of the Maths Lab's facilities. Peter Linington remembers using it to scan and analyse electron microscope pictures for his Physics PhD thesis. This involved writing PDP-7 programs which, once compiled, were punched out onto paper tape that was then read back in so that the program could be obeyed. The problem of forward references was solved by punching the tape backwards so that the symbol table, punched last, was read in first. This is typical of some of the tricks that were used in those days.

In the summer of 1969, the Maths Lab moved from the Old Anatomy Building into a new building that had been erected next door. Subsequently, the old building was demolished, to be replaced by a similar new building that housed the Department of Metallurgy. The old green door of the Maths Lab, which opened into Corn Exchange Street, was saved from the skip and kept by the computer engineers. It has subsequently been brought out at retirement parties for anyone who once used it, so that they could be ‘shown the door’. Nowadays it is mounted as a display in the Computer Laboratory.

By the time of the move, the Titan was providing a sufficiently important service that its loss for any length of time could not be contemplated. Fortunately, the Computer Aided Design centre was just being set up in premises in Madingley Road, and they were to have another Atlas II computer. Staff from the Maths Lab installed the Titan supervisor and associated software in the new Atlas, telephone lines were switched over, and for some weeks the Atlas II ran the University’s Computing Service before being handed over to the CAD centre.

The physical move of the Titan was accomplished by using a crane to lift its various parts up and into a hole in the side of the new building. This was rather scary, as there was no insurance, but it was accomplished without mishap.

The PDP-8

The first PDP-8 in Cambridge was delivered to DAMTP as I started my final year. The PDP-8 was the first so-called ‘minicomputer’, about the size of a small refrigerator. It used 12-bit words, with a very restricted instruction set of only eight main instructions. The DAMTP machine was one half of a hybrid analog–digital experimental computer system. It felt very primitive compared with the Titan, but it allowed me to do some hands-on computing again. Using this early model PDP-8 could be very tedious: it had no disc, and no built-in programs. To start it, you had to toggle in a short machine-code program on its key switches. This was enough to read a paper tape containing a loader program, and this could load (again from paper tape) the program you really wanted to run. To develop a new program you had to run an assembler this way, and then run the paper tape that it punched out.

I quickly realized that program development could be speeded up by writing a PDP-8 assembler that ran on the Titan, so I did just that, using the Titan assembly language (IIT). PDP-8 programs could now be held on the Titan disc, edited and assembled interactively online, and then output on the Titan’s paper tape punch. The only disadvantage was the need to go over to the Maths Lab to collect the tape. This assembly system was used for the main application that was developed (by others) for the hybrid system, and it continued to be used after I had left DAMTP.

Right at the end of my time in DAMTP, a second PDP-8 system arrived. This had discs, and ran a timesharing system called TSS-8. It was quite an achievement to write a timesharing operating system for that hardware, though it was rather slow under load. The discs were used for swapping, and it was believed that it could also use its mini-magnetic tapes (DEC tapes) if the discs filled up, though I never saw this happen. (It was further rumoured that if the tapes filled up, it would swap to paper tape, but that story was probably a joke.) This new model PDP-8 had some extra features in its instruction set, so I modified my assembler so that it could be used to build the TSS-8 operating system.

A short stint in the Computing Service

In the time between submitting my PhD thesis and my oral examination, I was employed for a few weeks by the Computing Service to add debugging facilities to the Titan’s FORTRAN system. With no prior experience or training, I had to figure out how the compiler (a large monolithic assembly language program) and the runtime system worked, and design code that would cause it to save enough information so that it could print out the values of programming variables in the event of a program crash. I managed to get a usable version working in the available time.

3. A brief career as a lecturer

After I completed my PhD at Cambridge, I returned to Cape Town and became a lecturer in Applied Mathematics at UCT. While I had been away, a new computer, an IBM 1130, had been installed, though the old ICT 1301 was still running administrative jobs. The 1130 was another hands-on machine, with a card reader for external input, but it also had a console keyboard so you could run interactive programs. Removable discs provided long-term magnetic storage. I wrote an interactive program for inspecting and patching disc files in binary.

An IBM 029 keypunch

As well as trying to teach mathematics, I found myself running FORTRAN courses for students. They did not have hands-on access to the 1130, but had to submit their decks of cards to be run by an operator. Evenings and weekends were really the only time I could get any substantial amount of computing time for myself, and though I had no major projects to work on, I made myself familiar with the computer and its machine code.

During the year I was there, the 1130 was enhanced by the addition of two half-inch magnetic tape decks. I learned that IBM magnetic tapes were not like the Titan tapes I was familiar with. They were not pre-blocked, and did not have a fixed block length. Furthermore, in theory, whenever a block was written to the tape, all the subsequent data was invalidated (because a new write could overwrite what followed). However, there was a command for skipping along the tape without writing, and some software from an American university (I forget which one) made use of this to implement an archiving system with an index that was kept (and updated) at the start of the tape. Learning about this turned out to be useful a year or so later.

UCT were already thinking about what their next computer should be. They wanted a much larger, multiprocessing system that could also support time-shared interactive access. Salesmen from the major computer companies of the day were plying us with information. IBM was hopeful, having a toe in the door with the 1130. I read a lot of manuals and was rather horrified by the very primitive job control language (JCL) that was all IBM offered for its 360 series of machines. Univac seemed to be the better deal to me, especially as it had what looked on paper like a reasonable time-sharing system. In the end, Univac was the successful bidder, but I do not know how much influence my views had on the decision.

My career as a lecturer did not last long. At the end of the year (before the Univac was installed) I returned to Cambridge to re-join the Computing Service as a software developer.

4. A computer officer at Cambridge

When I got back to Cambridge at the start of 1971, I found that the Computing Service, which had previously been an informal group within the Maths Lab, had now been formally established with David Hartley as its director. The Mathematical Laboratory itself had been renamed as the Computer Laboratory. The Computing Service, though independently managed, was still part of the same department, and in fact this continued until the teaching and research half of the Laboratory moved to new premises in West Cambridge in 2001. At that point, the Computing Service became fully independent.

I was appointed to the post of Computer Officer (grade IV), but there was no precise job description. I was just expected to do stuff connected with computers to the satisfaction of the head of the department – a far cry from the hugely detailed job descriptions that are required for every appointment nowadays.

I was put in charge of maintaining the Titan FORTRAN compiler, and in particular, tidying up the debugging system I had implemented a year earlier, which had had no maintenance in the meantime. I shared an office with Tony Stoneley, who had joined the Computing Service a few months previously, and who was maintaining the runtime system. Despite the heroic efforts of the engineers, there were still regular breakdowns that lasted several hours. We used to go for walks, or even punting, when this happened.

The machine was run with the doors off the cabinets, because closing them affected the temperature and caused problems. The circuit boards were all in vertical slots and in full view, and would occasionally get dislodged. There was one operator, whose name I forget, who always wore ‘winkle-picker’ shoes. He was adept at running his toe along the lowest row of circuit boards to make sure they were all properly seated.

The FORTRAN compiler had been written as a single, monolithic program, in IIT, except that names were used instead of numbers for the Titan registers. The code was passed through the ML/I macro processor to convert the names into numbers before assembly. It took several hours of real time to build the FORTRAN compiler on the Titan (mixed in, of course, with other jobs that were running at the same time). To save having to do this for every small change, the code was written to leave some spare space in the binary program, so that additional machine-code instructions could be patched in by hand, using a program called (obviously) *patch*. Needless to say, this was a somewhat fraught process, especially as such patches were only in the binary, and not in the source code. Forgetting to update the source would lose the patch at the next re-compile. (The verb ‘patch’ has changed its meaning since those days, and nowadays it means to modify source code, not binary.)

I discovered that the application used for building the operating system, called TSAS (Time-shared Supervisor Assembly System) and implemented by Barry Landy, could also be used for building applications. It allowed a program to be split into many parts, and only those parts that changed needed to be re-compiled. It also had facilities for storing source code along with files of edits, so that changes could be tracked and if necessary, removed. In other words, it was a kind of early version control system, combined with something like the Unix *make* program.

I spent a few weeks chopping up the FORTRAN compiler into separate modules and getting it to build with TSAS. After that, recompiling small parts of it could be done quickly, even online sometimes, so maintenance became much easier and less error-prone, and there was no longer any need to indulge in binary patching. Then I refurbished the debugging system. After about a year though, Titan software maintenance and development came to an end.

Replacing the Titan

Planning was already well underway for a replacement for the Titan, and soon after I joined the Computing Service it was announced that Cambridge would be getting an IBM 370/165 mainframe, the first of the 370 models. It would be running OS/360 MVT, with the newly announced Time-Sharing Option (TSO). Cambridge had successfully argued with the funding bodies that its users needed compatibility with their overseas colleagues, especially those in the USA, where IBM machines were widespread. Most other UK universities were installing British-made ICL mainframes at this time.

It was rather ironic that, having argued against IBM for UCT, I was going to be stuck with their system nevertheless. We technical staff started reading IBM manuals and were sent on training courses. I even went on a PL/I programming course where (having read the manual) I amazed the instructor by writing an example program that worked first time. We also travelled to IBM’s data centres in London to get experience with IBM mainframes and the methods of generating the operating system. It was somewhat of a culture shock after the Titan.

The machine itself arrived early in 1972 and it was all hands to the pumps to get it into a state where we could run some kind of user service. Meanwhile, the Titan continued to run until it was closed down in the autumn of 1973.

Many years, many computers

The remaining chapters of this book are arranged by topic rather than chronologically. A time-ordered description would have to interleave material about different computers, and in any case, my memory of exactly what order things happened in is a bit hazy.

I was involved with IBM mainframe software until the early 1990s, but during that time I also wrote programs for a number of other computers. In the area of communications I worked on a PDP-11 front-end to the mainframe, and also on an Interdata remote job entry system. Later on I did a small amount of work on the Ultrix, the VAX/VMS system, and even very briefly on MS-DOS. At home I played with a Sinclair ZX-81, then a BBC micro, and later Acorn's 32016 and ARM second processors and their Archimedes and RISC OS machines. One major application that came out of my home computing was *Philip's Music Writer* (PMW), my music typesetting software. Though some of my code runs on Windows, I have never myself used that operating system, though this was more by chance than deliberate choice.

Around 1990, the Computing Service decided to set up a central Unix service, and to that end purchased a cluster of Sun boxes, running SunOS (later Solaris). Since then my focus has been on writing software for various flavours of Unix and Unix-like operating systems. In 1995 I started development of the Exim mail server, and in 1998 the PCRE regular expression library, which between them occupied most of my working hours until I retired in 2007. At that point I stopped working on Exim, but I continued to maintain PCRE.

Over the years, the Computing Service, having initially appointed me as a Computer Officer Grade IV, promoted me through the grades (with one period as a Senior Technical Officer) to finish up at Grade I. For a short time I was 'Manager of Applications', in charge of several other programmers working on applications software, and I was Director of Studies at Emmanuel College for ten years, looking after the academic progress of the small number of computer science students there. For a few years I gave lectures about the IBM mainframe operating system to computer science students.

I also did some computer science supervision, including a couple of interesting student projects. One of these, on music typesetting, was what inspired me to get involved in that myself (more about this later). Another was Nick Smith's email client for the Acorn RISC OS system, which went on to become a commercial product after he graduated and went to work for ANT Ltd.

As it turned out, managing people did not really suit me and I did not like doing it, so I slipped back into being a software developer for the rest of my time in the Computing Service, though in addition to writing code and documentation I also ran a number of training courses, gave occasional seminars, and chaired various meetings.

5. The Phoenix Years

The Computing Service ran an IBM mainframe from 1972 until 1995. The 370/165 that arrived at the start of 1972 was the first to be installed in the UK; when it was replaced in 1982, it was the last to go. I wrote a lot of software for the mainframe system. The utility programs were all written in Assembler, for performance and space reasons, and I wrote in BCPL for some larger applications.

The 370/165 arrived with just one megabyte of core memory; with this we managed for about eighteen months until money was found for an upgrade. A second megabyte of what was then new technology, semiconductor memory, was bought from a third-party manufacturer, after some negotiation to ensure that IBM would still maintain the machine. Each of these megabytes occupied a box the size of a large refrigerator. Commissioning the second megabyte was one of the more fraught times in the early IBM years. The installing engineers could not get it to work. Eventually Chris Cheney and Mike Guy read the circuit diagrams, spotted the error, and worked out how to fix it. During their testing, they had to find a way of seeing an event that occurred half a microsecond before its effect was noticed; the answer was to use 500 feet of coaxial cable. Later, two more megabytes were added, taking the machine up to its memory limit of four megabytes.

2017: *Today I have three gigabytes of RAM in the phone in my pocket, that is, 750 times the amount in the 370/165 memory, at a fraction of the size and cost. It is amazing how much has changed since those days.*

The mainframe had a cathode ray tube ‘visual display unit’ (VDU) as the operators’ console, whereas the Titan had used a ‘golf-ball’ typewriter. One disadvantage of the VDU was that there was no hard copy, not something one thinks about at all nowadays. We had had to make a special request to IBM to get a US rather than a UK keyboard for the console. The UK keyboard had a pound sterling sign instead of a dollar, though it generated the same EBCDIC character code. This in itself was not a big problem. The real issue was the dollar sign that replaced the US cent sign. Many system operations commands started with a dollar, and we knew that a dollar on the keyboard that did not produce a dollar on the screen would be a recipe for confusion.

File management utilities

When we first started using OS/360, we discovered that the utilities provided for doing things such as listing one’s files were very clumsy and primitive, and, before the user service started, I hastily wrote some new ones that were a bit more user-friendly. I also wrote, in PL/I, a very simple batch (or ‘stream’) text editor called E4, to provide similar editing commands to those that we were used to. This stopgap was used for a few months until Gill Cross implemented a ‘proper’ editor in Assembler. Later, when Gill left Cambridge, I had the job of developing a successor. Chapter 7 covers my experiences with text editing and text formatting, which lasted for many years.

The main debugging tool for Assembler programmers was the ‘core dump’, which consisted of a printout of the contents of machine memory. We found the IBM dumping facilities to be far too gross for the ordinary user – pages and pages of hexadecimal, most of which consisted of parts of the system. I wrote a local supervisor call (SVC) that produced a compact dump, controlled by options set by the program itself, and capable of producing output in a variety of formats such as instructions, decimal, characters, etc. This found heavy use for several years until we were fully into time-sharing and found that a binary dump to disc was more useful. By that time I had written a disc inspection and patching program, and an interactive debugger.

One of the problems with OS/360 was that it supported many different file formats. The file system did not use fixed sized disc blocks, as the Titan had done, and as modern systems do. Instead, files were allocated as whole numbers of tracks, which the user program could fill with blocks of any size. Various ‘access methods’ were provided for reading and writing files in several different formats. Unlike modern operating systems, where a file (as seen by a program) is just a stream of bytes, OS/360 files were read and written as *records*.

The two most common formats for text files were fixed length records (typically card images of 80 bytes, but any length was permitted), and variable length records where each record contained its length in two bytes at the start. A file containing one of these types of record could be *blocked*, that is, many records could be stored in each disc block. Furthermore, each record could optionally contain a FORTRAN-style carriage control character at its start, for example ‘0’ to skip a line or ‘1’ to start a new page. A third type of file, known as *unknown length records*, held each record in a single disc block, without an explicit length.

A utility program such as a text editor that needed to be able to handle any kind of file had to deal with these many different file formats. To make it more straightforward to write utilities, I wrote a library (in its final incarnation called NIOP), which handled all the different file types while presenting a standard interface to its caller. This was widely used by myself and other software developers throughout the mainframe years.

Software from other sources could often handle only a subset of the file types, or would misbehave if given a file of an unexpected type. For example, passing a file of variable-length records to the SORT/MERGE utility without an appropriate control file caused the file to be sorted by record length, which was rarely useful.

A consequence of allocating files in tracks was that the smallest possible file was rather large, because one track of the then new 3330 disc drives could hold up to 13030 bytes. IBM had thought about this issue, and had invented *partitioned datasets* or PDSs (*dataset* being their word for file). Like the *microlib* feature we had had on the Titan, a PDS was a single file that contained a number of smaller files, similar to modern-day archive files, except that individual members could be read or written directly from the PDS. One of the utilities I wrote was for listing the members of a PDS in a convenient format.

Timesharing and the birth of Phoenix

A card-based batch IBM mainframe service went live in 1972, a few months after the 370/165 was delivered, delayed for several weeks because of electricity strikes that were happening at the time. Users were allocated disc space on which they could store their data, but for quite some time there was no data protection at all; anybody could read or write any file. Furthermore, there was no enforced automatic control over the amount of space any user used. Amazingly, this did not seem to cause any problems.

Meanwhile, the Computing Service was working towards providing a time-sharing capability. IBM's operating system had only recently added TSO ('time-sharing option'), which, however (in our opinion) fell far short of what the users of Titan were used to. Barry Landy started work to add what we would now call a shell on top of TSO to provide a command environment that was similar to the Titan. As the 370/165 had no paging hardware, something similar to the Titan's 'normal mode' was required to give acceptable performance for the expected number of users. The new shell was eventually given the name *Phoenix*, and in time this name came to be applied to the entire customized Cambridge mainframe system. The timesharing service went live early in 1973. The Phoenix command processor could also be used to control batch (offline) jobs, as an alternative to IBM's Job Control Language (JCL), which was very cumbersome.

IBM's communications controller had been judged too expensive and inflexible, and instead, one of the first PDP-11s (model 11/20) had been purchased, along with special hardware (DX-11) to connect it to an IBM input-output channel. Mike Guy wrote a small operating system for it, and implemented software to allow the mainframe to talk to Teletype terminals. Later, I joined him to develop the PDP-11 software further (see chapter 6).

Peter Linington had persuaded the Computing Service to buy another PDP-11 in the form of a DEC GT40 Graphic Display Terminal, which was based on a PDP-11/05. This was set up in the computer room to allow for graphics experiments using its monochrome (green) display. The DEC engineer who installed it brought a ‘moon landing’ game on a paper tape, which entertained Computing Service staff for many hours. The aim was to land your craft safely as near to the outlet of a well-known hamburger purveyor as you could, and after the ‘pilot’ had bought a hamburger, take off again. After some practice, we all became quite good at it. The GT40 was eventually used as a graphics preview facility, running a terminal emulator that Peter wrote.

Mainframe peripherals

Punched cards were the main input medium to Phoenix, at least until a screen-based text editor became available, though paper tape could also be read by a reader connected to the PDP 11/20. As on the Titan, the main output device was a lineprinter. At the beginning, we were unusual in insisting on upper and lower case letters and plain white paper. At first this consisted of the conventional large pages, but later a shorter sheet was introduced, with the lines printed closer together. This was easier to handle, and it also saved money. There was a Calcomp pen plotter for graphical output, and both paper tape and cards could be punched.

Teletypes displayed only upper case letters. Phoenix did support IBM golfball terminals, but only a very few were ever used. Later, when daisywheel terminals came along, one was installed in the Computing Service bookshop. Users could send ‘high quality’ output there, paying a per-page price for the privilege. This situation lasted until laser printers, capable of printing both words and pictures on A4 paper, became available.

In the early days of VDU terminals, there were many competing manufacturers, each inventing their own set of control codes. In the Unix world, this caused the invention of the *termcap* and *terminfo* databases, so programs could be written in a reasonably device-independent manner. These databases still exist, though most of the contents are now only of historical interest. Making use of the control features required character-by-character interaction; this was not available on the IBM mainframe, which performed terminal input and output line-by-line. As a result, screen-based applications came quite late to Phoenix.

For the 370/165, we avoided using IBM screen terminals (model 3270 and similar) because they required their own coax cabling and special communications protocols. It was clear that supporting them for users was not a viable option. In the 1980s, when we started running MVS on new hardware, 3270 terminals were installed in some staff offices because by that time so much IBM system software assumed their use.

For the users, after trying several types of screen-based terminal, the Computing Service settled on a model made by Newbury – very rugged and utilitarian. Although the Newburys supported some screen control features, they were never seriously used in a fancy way. They were just ‘glass Teletypes’, displaying a scrolling sequence of lines of text. There was no scrollback. Newburys were superseded by BBC micros, which were just as cheap, and much more flexible. I will come back to this later, when discussing text editing in chapter 7.

A dead-end operating system

In 1972, only a few months after the start of the 370/165 service, and rather too soon for our liking, IBM announced the 370 model 168, whose major innovation was the addition of paging hardware to support the use of virtual memory. The operating system was to be upgraded and renamed MVS. An upgrade to the model 165 was available, but it was so expensive that the possibility of getting it was never considered. Not having the appropriate hardware meant that Cambridge was cut off from IBM’s operating system development. With hindsight, however, this turned out to be a blessing in disguise.

We had the source code of the operating system we were using, and we were trying to do novel things with it. Since there were to be no further updates, we had no compunction about hacking it about to suit our needs, and hack we certainly did, until the time came in the early 1980s to replace the hardware and then move on to MVS, which by that time had acquired sufficient features for our requirements.

Programming languages

Phoenix supported many programming languages. The command interpreter was written in Assembler, and many of the utility programs were as well. A BCPL system was implemented fairly early on, and this was heavily used, both by Computing Service programmers and by the users. Also fairly early on, I ported the ML/I macro language, though I do not think the Phoenix version ever saw much use.

From IBM we had PL/I and FORTRAN compilers. For PL/I there were two: a 'debugging' compiler and a 'production' compiler. The idea was that you used the former (which produced poor code, but was fast and had lots of diagnostics) to get your program working, and the latter (which did a lot of optimization) to compile for production. In practice, they were never entirely compatible, and this caused problems for several users. There were also two FORTRAN compilers, known as 'G' and 'H'. The latter used more memory, but was better at optimizing its output.

For a couple of years I was in charge of applications software, and during that time I worked on a FORTRAN graphics library, which had functions for drawing graphs and various other kinds of graphic. It generated output for the pen plotter. I have to admit I was relieved eventually to hand this over to a colleague who knew a lot more about graphics than I did.

Also at that time I was looking after a Simula implementation. Simula was a simulation language that came from the Norwegian Computing Centre in Oslo. It was an early object-oriented programming language. The major user at Cambridge was a research student called Bjarne Stroustrup, who kept finding bugs in the system. Each time we had to put the evidence onto a magnetic tape and post it off to Norway. A few weeks later, back would come the tape with a fixed version of Simula. Even when the bugs were mended, however, Bjarne found that Simula was very difficult to use for the tasks he wanted it to do. After he finished his PhD at Cambridge, he moved to Bell Labs and invented C++.

A version of SNOBOL called Spitbol was supported. This was a non-procedural language for pattern matching and text manipulation. I played around with it a bit, but I never really got the hang of it, and did not implement any serious projects using it.

There was a project run by Steve Bourne in the research side of the Computer Laboratory to implement a version of Algol 68 called *ALGOL68C*. Later this was taken over by the Computing Service, where Martin Cole and Chris Cheney did a lot of work on it. Tony Stoneley joined them at the tail end of this project and wrote one of the test suites. A compiler and runtime system were put into service, but the project was never fully completed. I did not use Algol 68 myself until the Phoenix system was being ported to MVS, when I helped Tony a little with the Eagle project (see later).

The original Phoenix system, running on OS/360 MVT, never had a C compiler. For a number of years the Computing Service looked without success for a C compiler for Phoenix/MVS. Then in 1987 the Norcroft ANSI Standard C compiler became available. This compiler was written by Arthur Norman and Alan

Mycroft, lecturers in the Computer Laboratory, and was originally developed for different hardware (Acorn's ARM processors). The compiler was ported to MVS by one of its authors, but it lacked a fully implemented runtime system. The Computing Service undertook to provide a runtime system to the ANSI Standard.

The Assembler sections of this system were designed and written by Nick Maclaren. I undertook to test his code and to create the C portion of the library, partly by modifying the code of an existing implementation, and partly by writing new C code. This work was done over the summer of 1988. Once the library was up and running, I wrote a comprehensive testing package to test every routine in the library. This showed up a number of bugs in both the new and the old code. Finally, I wrote complete documentation for the MVS C system – an 80-page booklet.

Other people's projects

As well as the command processor written by Barry Landy, a lot of other software work was done by Computing Service staff to create the Phoenix system and all that went with it. A few of the projects are described below, to give a flavour of the kind of things we had to do in the 1970s.

Extending HASP

The MVT operating system had no concept of user accounts, and no means of controlling the resources that users used. Also, the spooling facilities (for reading and queueing jobs and output) were inefficient and unfriendly. Fortunately, an IBM field engineer in Houston had written some software called HASP (*Houston Automatic Spooling and Priority*), which ran 'on top of' MVT, provided much better facilities, and was freely available. HASP took control of the slow peripherals to provide input and output spooling. This was a clumsy approach, because the underlying MVT still dealt with IBM channel programming, and HASP had to catch interrupts and simulate the hardware, but it worked, and many sites used it.

At the time there was great rivalry between the maintainers of HASP and IBM, the latter being very wary of anything that was 'not invented here'. However, some years later IBM changed their minds and absorbed HASP into what became the MVS system, renaming it JES (Job Entry System).

The first modification to HASP was to make it recognize user accounts. Peter Linington wrote an application called 'Job 0', which was a continuously running program that could verify user account information. When reading jobs from the card reader, HASP could now call Job 0 and thus enforce the correct use of userids and project numbers. Userids with a length of four characters were issued, based on users' initials, followed by digits as necessary. Four characters conveniently fitted into a single 32-bit hardware register for processing.

I became PH10. Userids were usually written in upper case in those days. We started the digits at 1 to avoid confusing the digit 0 with the letter O. With hindsight, we should have started with 2, which would have avoided confusion with the letter 'l' in later years when userids were written in lower case for Unix and other non-mainframe systems. Though no userid was ever to be re-used, it was felt that four characters would last long enough. In the event, longer userids were eventually required, but not for many years. At first, the data for Job 0 was maintained by reception staff in a file on the Titan, and transferred daily to the IBM machine. Subsequently it was extracted from the Jackdaw database.

Another early modification that Peter made to HASP was to reduce the amount of lineprinter paper used by a small test job from six pages to one. The saving in the cost of consumables was more than his annual salary.

The second major feature that was implemented for HASP was a job scheduler. On the Titan, users had been allocated a certain amount of processor time for their projects, but the job queue was first-in-first-out. John Larmouth felt that a better scheme was needed for the 370/165, and so he implemented the *Larmouth Scheduler*. This was based on the notion of giving each user a certain number of *shares*, which were an entitlement to a certain proportion of the machine, averaged over time.

The scheme was complex, and required continuous tinkering, but on the whole it did a remarkably good job. The most innovative aspect was in trying to make accurate forward commitments about when a job would be processed, but this made it hard to keep the whole thing stable. Peter Linington maintained it after John left Cambridge.

John was also involved in setting up a FORTRAN 'batch monitor' for the use of students. This was based on WATFIV, a FORTRAN system from the University of Waterloo (an upgrade to their original WATFOR). By using a self-service card reader and lineprinter, students could get more or less immediate turnaround for their small FORTRAN programs.

Magnetic tape storage

IBM's half-inch magnetic tapes were unlike those on the Titan, and, not being pre-blocked, did not support random access. Data blocks of arbitrary size could be written, and writing a block supposedly invalidated what followed on the tape. In theory, therefore, a tape archive system that kept a table of contents at the start of the tape (as had been used on the Titan) could not be implemented. However, having had the experience of using just such an application on the IBM 1130 at UCT (see chapter 3), I insisted that it was possible, and as a result Gian Boggio-Togna was able to implement TLS (*Tape Library System*), which was then used as the main archive tool for users' files.

2017: *The tapes themselves (model 3420, I think) were much easier to handle than the one-inch Titan tapes, which had had to be carefully threaded around various capstans when loading. The IBM tapes were in cartridges and all one had to do was put the cartridge onto the deck and push a button – it then self-loaded.*

A user message system

In the Titan multiple-access system, users could send each other short messages; in effect this was a very early, very restricted email facility. IBM's TSO did come with something similar, but used a whole file for each message. This was far too wasteful of disc space, as the smallest size of file was one whole track (13K bytes). I implemented a system in which all the messages were stored in a single file, with space being re-used when messages were deleted. This was used as the Phoenix inter-user messaging system until 'proper' email was introduced in 1986.

The Jackdaw database

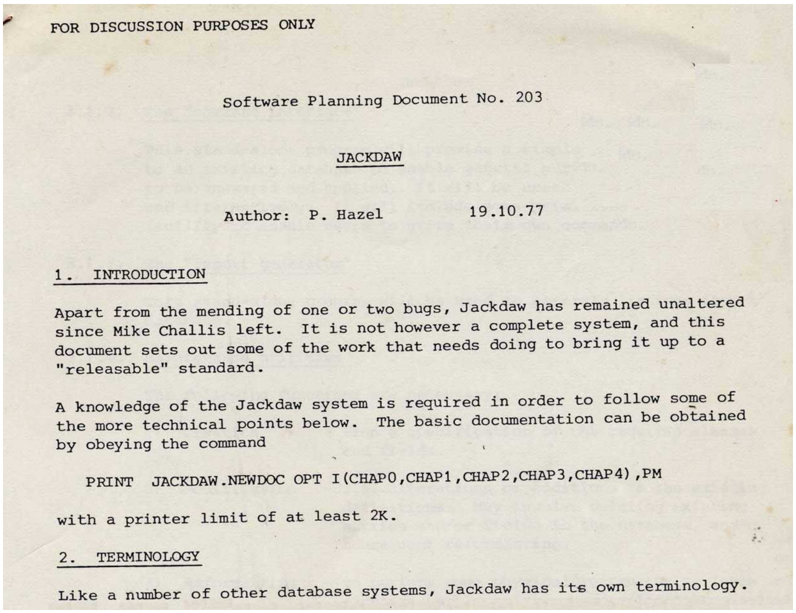
The IBM mainframe was expected to have many more users than the Titan, and a database of some sort was needed for the convenient management of accounts. At this time there were no suitable database products available, and databases as we now know them were in any case in their infancy. Mike Challis set to work to implement a database package for the use of the Computing Service. It was written in BCPL, and was called Jackdaw. (After the name Phoenix was chosen, many of the subsequent software projects were named after birds.)

Nowadays, relational databases are what most people use, but back in the 1970s, they were new and experimental. Two other ways of structuring data, so-called *hierarchical* and *network* databases, were being implemented. Jackdaw was a network database.

After Mike had been working for some time, the management must have become worried that he was not progressing, because they asked me to join him to work on the project. I found that in fact there was very little left to do, and as far as I can recall, the only thing I contributed was a way of loading the BCPL runtime library dynamically, which meant that the latest version would automatically be used whenever Jackdaw was run.

In 1977, the Computing Service hosted a meeting of SEAS (*Share European Association*), the European users' group for large IBM systems. Around 360 people attended. Starting about a year beforehand, I implemented a Jackdaw database to hold details of all the delegates and all the presentations for this event. One of the more challenging programming problems was writing the application for printing the timetable in multiple columns, automatically adjusted to fit the size of the data. We were able to accommodate last-minute changes and print up-to-date information and invoices on demand. This was all done using a lineprinter, the only available hardcopy output device at that time. The meeting was a great success, but it did divert a lot of Computing Service manpower.

Mike Challis had left the Computing Service by the time of the SEAS meeting, and after the meeting was over, I wrote a planning document about the future of Jackdaw. I had completely forgotten about this until Mike recently drew it to my attention (Roger Stratford had sent him a copy).


FOR DISCUSSION PURPOSES ONLY

Software Planning Document No. 203

JACKDAW

Author: P. Hazel 19.10.77

1. INTRODUCTION

Apart from the mending of one or two bugs, Jackdaw has remained unaltered since Mike Challis left. It is not however a complete system, and this document sets out some of the work that needs doing to bring it up to a "releasable" standard.

A knowledge of the Jackdaw system is required in order to follow some of the more technical points below. The basic documentation can be obtained by obeying the command

```
PRINT  JACKDAW.NEWDOC OPT I(CHAP0,CHAP1,CHAP2,CHAP3,CHAP4),PM
```

with a printer limit of at least 2K.

2. TERMINOLOGY

Like a number of other database systems, Jackdaw has its own terminology.

Software Planning Document 203

This document was written in the days before wordprocessing, and I must have written it by hand and had it typed by one of the secretaries. It contains a number of ideas for the further development of Jackdaw. I do not know how many were ever implemented, because I did no more work on Jackdaw. Later development was done by Charles Jardine, who maintained Jackdaw for the rest of the life of Phoenix.

As well as being used by the Computing Service for user management and the SEAS meeting, the Jackdaw software was used by one other large project. The University's central administration constructed a Capacity Record database, containing details of every building and room that were part of its estate. I can remember assisting with this, but the details are very hazy. Once it was up and running, Janet Linington maintained the data from home, using a Teletype and a dedicated line. (There was no networking of any kind back then.)

The Phoenix HELP system

For the SEAS 77 meeting we thought it would be a good idea to make 'information terminals' available to the delegates. This was a novel idea at the time. Several Newbury terminals were installed for delegates' use. These were text-only, monochrome screens, probably running at 1200 baud, though some Newburys could manage 9600. I wrote a simple 'help' program that scanned a file for lines of keywords and then output the paragraphs that followed any matches. This was based on a similar scheme that Barry Landy had used for keeping track of works in progress.

We created some basic information such as the local train times, location of pubs and restaurants, times for the conference and so on. The system was well used. It logged all failed requests, and each evening Roger Stratford examined the log and updated the information if possible. This particularly impressed the delegates.

After SEAS was over, I added some more features to the program, and it went into service as the Phoenix *help* command, with Roger in charge of the data. Over the years a huge amount of information was amassed, including a few jokes. Best remembered is probably the response to '*help god*', which was '*Deities must be invoked directly and not via Phoenix*'. The program was also used by Mike O'Donohoe, who created a help system containing information about numerical analysis and relevant software.

Replacing the 370/165

As the 1970s drew to a close, the Computing Service started thinking about replacing the 370/165. At that time, universities' central computer systems were funded by the government Computer Board, and a case had to be made. IBM naturally started lobbying for the contract. In 1979 they took some of us to the United States on a study tour. We visited several universities and IBM sites on both the East and West coasts. In Poughkeepsie we met with one of the chief developers of the operating system. One outcome of this was that in the summer of 1980 the Computing Service undertook a joint project with IBM, to port the Phoenix command interpreter to an unmodified MVS system.

An IBM man spent several months with us in Cambridge while we did this, using terminals that connected to a machine in one of IBM’s London data centres. Working remotely like this was still quite a new thing. Installing the utilities that I looked after was straightforward, and so I spent most of my time working on documentation so that we had a complete manual of the resulting system. Barry Landy modified the command processor to use only the native facilities of MVS.

The final result was taken to Poughkeepsie (on magnetic tape) by Barry and Tony Stoneley and installed on a computer there. The idea was that it should be run for a number of weeks so that IBM personnel could play with it and perhaps pick up some ideas. Someone from the Computing Service would remain there to babysit. John Powers took over from Tony and Barry, and I took over from him for the final two weeks at the end of November. It was interesting, but rather strange being on the ‘inside’ of IBM, but with very little to do. My stint was over Thanksgiving; this allowed me to escape to visit some friends in Massachusetts for a few days. I do not know how much attention IBM paid to Phoenix, or whether it actually had any influence on their software.

While we had a presence in Poughkeepsie, IBM announced replacements for its 370 series, in particular, the 3081 processor. This was of course what they would now try to sell us. In 1981 a second study tour happened, and in due course the Computer Board allowed Cambridge to purchase a 3081, which was installed in 1982.

IBM’s VM (*Virtual Machine*) operating system allowed us to continue to run the old Phoenix/MVT software that had been developed on the 370/165 while at the same time developing a more complete implementation of Phoenix under MVS. Each operating system ran in its own virtual machine, on the same hardware. As a result, it was no longer necessary to book dedicated time, and therefore take the system away from the users, in order to test or reboot (or rather, in IBM-speak, to ‘IPL’, from *Initial Program Load*). Operating system testing at arbitrary times was something that had never previously been possible, and naturally it made development easier and reduced the number of antisocial hours that staff had to work.

VM had been developed at IBM's Cambridge (Mass) labs, and was used by a number of universities, but had never been a fully-fledged product, rather like HASP. In both cases, IBM eventually realized that the software should be fully supported. HASP was integrated into MVS and renamed JES (*Job Entry System*) and VM became a mainstream product. There had been fears that running a production MVS (or for that matter MVT) system under VM would be too inefficient, but these fears proved groundless. We were able to run the eventual production MVS under VM without any performance problems.

Until the arrival of the 3081, Computing Service staff had used Teletype or Newbury terminals, just like the users. Because Teletypes were so noisy, they were mostly in shared terminal rooms, with only those staff who had an office to themselves being able to have their own private terminal until the Newburys (and later the BBC micros) came on the scene. When the 3081 arrived, coax cabling was installed so that those working on VM and MVS could use IBM 3270 terminals from their offices. Trying to do MVS development on anything else would have been extremely difficult.

Although the Phoenix command interpreter had mostly been ported to MVS during the project with IBM Poughkeepsie, there was still a lot to be done before the user service could be moved over. One major missing item was a job scheduler. Tony Stoneley and I started to write a replacement for the Larmouth Scheduler in Algol 68C. We called this *Eagle*, since it was to watch over jobs and online computer usage. My contribution was very small, and I soon moved on to other things. I wrote some data storage modules in the beginning, but the bulk of the work was done by Tony.

Another major project was a program called CIP ('Communications Interface Processor'), written by John Powers. I can no longer remember the details, but I believe it was used to replace IBM's terminal-driving software, at least for non-IBM terminals.

Two years after the change of hardware, in 1984, the user service was moved from Phoenix/MVT to Phoenix/MVS. There were some subsequent processor upgrades between then and 1995, when Phoenix was shut down, but they were all compatible, and did not require any major software upheavals.

Leaving Phoenix

In the second half of the 1980s people started thinking about the long-term issues of running a mainframe. Personal computers had not yet reached the power they have today, but in many academic institutions, medium-sized machines such as DEC VAX systems were in use. Some ran DEC's VMS operating system, and some ran Unix. It was not obvious at the time which of these two horses to back. For a number of years, the Computing Service ran a VAX/VMS support service, offering assistance to Departments that were using VMS. As part of this service, John Line ran an in-house VMS system that could be used for testing. My only contact with VMS was in connection with E, my portable text editor (see chapter 7).

In 1989, with Unix becoming more popular, the Computing Service decided to add a Central Unix Service (CUS) for the use of staff and graduate students. To this end a cluster of Sun systems, funded by the Computer Board, were installed early in 1990. At that time, I stopped doing anything other than tinkering work on Phoenix, and moved over to developing and supporting software for Unix, starting with a number of programs and scripts to enable reception staff to manage CUS accounts from the User Database, which at that time was still running on Phoenix. We also used the Phoenix print queues in the beginning. The details of my subsequent work appear in other chapters. Phoenix itself continued to run until 1995. The CUS system ran, with various hardware and software upgrades over the years, until 2008.

6. Programming in communications

When the first mainframe was purchased, IBM’s communications controller was insufficiently flexible as well as rather expensive, so instead a PDP-11/20 was bought, along with some special hardware that allowed it to connect to an IBM input-output channel. Both the PDP-11 itself and the channel interface (a DX-11) were new products for which there was no off-the-shelf software. The PDP-11 was also to provide a paper-tape interface to the 370/165.

Mike Guy set to work on the PDP-11, and implemented a basic communications system. Before he could do this he had first to write a small realtime operating system, and a system debugger (called *Totally Integrated Testing System*). Later, after a terminal system for Teletypes had been in service for some time, I joined Mike to work on the PDP-11.

I was given the job of re-writing and extending the paper tape input facilities to support different tape codes and escape characters. Then I did a similar job for the code that controlled asynchronous terminals (mostly Teletypes). I rewrote the code so that different hardware multiplexers could easily be introduced, and implemented support for IBM 2741 ‘golf ball’ terminals. More user-friendly control of terminals such as the provision of escapes for characters not on the device (lower case letters being a prime example), and the clearing of screens for cancelled lines, were also included. I continued to extend and maintain this software until 1976.

Testing the terminal-handling software had to be done in dedicated time. Once it was all apparently working, we put it into service in the middle of the night, whereupon it promptly crashed. Further testing failed to uncover the fault. After a couple more attempts, Mike realized that the increased load in service was causing the software to use more memory than it did in testing, and that the extra memory hardware was faulty – it had no error-detection or correction facility. Replacing the hardware fixed the problem, much to my relief.

At the IBM end, the terminals were driven by a large, unwieldy piece of software called TCAM (*Terminal Communication Access Method*), much of which was not relevant for us. There were also problems with Teletype input because the PDP-11’s DX11 interface to the IBM channel could not completely emulate the IBM controller that TCAM expected to see. Tony Stoneley proposed to replace TCAM with a home-grown, more efficient program that was tailored to our needs and which would work within the DX11’s limitations. This required new software in the PDP-11; as we shared an office, he and I worked out a suitable protocol by simulating the data flow between our desks. Tony’s replacement for TCAM was dubbed *Parrot*, and lasted until the changeover from MVT to MVS.

After I left the communications team, the single PDP-11 expanded into a cluster, and evolved into a network controller, running X.25 networking protocols as well as controlling terminals. Eventually, terminals no longer had their own dedicated lines, but were multiplexed onto X.25 connections. Hardware for doing this, called the *Transport Service Box* (TSB), was designed by Chris Cheney and Mike Guy. The *Joint Network Team* (JNT), a national body that was overseeing networking in universities and research establishments, gave a contract to a company called Camtec to manufacture a compatible version of these boxes and sell them under the name 'JNT PAD'. As a result, they were quite widely used all over the UK.

The next step was network connections to other institutions. I well remember the first time I logged on to a machine that was outside Cambridge. I had been asked to install my mainframe text editor on an IBM system at the Rutherford Laboratory. Instead of going there in person, which was what would have had to happen in the past, I was able to log directly into their system from an experimental terminal in the Cambridge machine room – this was before networking was made generally available. The source had been loaded onto their disc from a magnetic tape that I sent by post. It was quite exciting being logged into a 'foreign' machine that was so far away, though in principle it was no different to what we had been doing for years.

The connection may well have used the Post Office's *Experimental Packet Switching System* (EPSS), which was the first public networking facility to be set up in the UK. (This was in the days before BT, when the Post Office ran the telephone network.) Shortly after EPSS was set up, there was an Open Day to show off networking to businesses, and I went to one of the demonstration sites in London. They were able to connect to Phoenix, but were having trouble with one of the demonstration programs. I knew the author was Arthur Norman, so I offered to contact him for help. Logging in to Phoenix, I sent him a message. All this was visible to the watching audience, as the terminal's display was being projected onto a large screen.

After a while, a reply that solved the problem arrived, followed by the sentence 'By the way, I am in Salt Lake City.' I think that interaction did more to convince the audience of the benefits of networking than any of the set-piece program demonstrations.

At the end of the 1980s, the Computing Service created the Granta Backbone Network (GBN), which consisted of a series of ducts connecting the majority of

University and College premises, into which cable and optical fibre could be placed. Chris Cheney, who was in charge of the project, had to learn quite a lot of civil engineering. Eventually, the GBN provided the means to support a high-speed TCP/IP network, overseen by Philip Cross. I was not involved in any of this, except for the implementation of an X.25↔TCP/IP gateway system when we first started running TCP/IP services (see chapter 9).

The Interdata-11 Remote Job Entry System

The University of Cambridge is spread over a wide area, and back in the early Phoenix days, when many users were submitting jobs on punched cards and getting the results as lineprinter output, the ability to use the system from remote locations was desirable. HASP supported this way of working via its *Remote Job Entry* (RJE) protocol.

The Department of Applied Economics were big users of Phoenix, and were keen to have their own outstation, to save their users from having to travel to the Computer Laboratory to submit jobs and collect output. They made available a room in their basement on the Sidgwick Site. An Interdata 11/70 minicomputer was purchased, and I was given the task of setting it all up.

The machine had a 16-bit processor, and an instruction set based on that of the IBM 360. It did not have a disc, so programs had to be loaded from cards or paper tape. It arrived with a simple operating system called BOSS (an imperfect acronym from *Basic Operating System*). We had also obtained Assembler code for an RJE program from University College London (UCL). As I had done some years before in my PDP-8 days, they had realized that life would be much easier if the program could be edited and assembled on a larger machine that had discs. They had an IBM 360 at UCL, and what they did was to write a set of macros for its Assembler, somehow persuading it to generate punched cards that the Interdata could load.

I was not happy with this arrangement. For one thing, using so many macros made assembling a program of any size rather slow, and (to my mind more importantly) it was not possible to support the full complement of opcodes for the Interdata-70 because of the way some of them differed from 360/370 opcodes. I solved both these problems by writing a proper cross-assembler that ran on the mainframe. It was called MIDAS (*My Interdata Assembler*) and was written in IBM Assembler, so it ran quickly. It could generate both punched cards and paper tape output. The latter was preferred, not only because it was more portable, but also because the Interdata's 'cheap and cheerful' card reader often had problems.

Once I had an efficient means of assembling Interdata code, I started work on the RJE software. Two things became immediately apparent: the operating system was very crude, and there was no debugging facility. There were both hardware and software errors while trying to get the RJE program to work, but the OS could not distinguish between them. In the end I abandoned it, and wrote my own operating system, called Phobos (*Philip's Own Basic Operating System*). It was heavily influenced by the OS that Mike Guy had written for the PDP-11/20, and I also implemented a similar debugging program to the one used on the PDP, calling it *New Interdata Testing System*.

After I had hacked it about (to add support for paper tape, amongst other things), I called the RJE program Deimos (to go with Phobos). When it was all working, there was a grand opening ceremony on the Sidgwick Site, at which Lucy Slater, a computer pioneer who had used computers in Cambridge from their very beginning, did the honours. The outstation operated successfully for a number of years. As well as the RJE system, there was room for a few terminals where users could work online. They were able to route any printouts to the local printer.

7. Text editing and text formatting

My first foray into text editing was the writing of E4, a simple PL/I program for the IBM mainframe that supported some of the basic stream editing commands of Titan's E3 editor. This was a stopgap, and was soon superseded by an editor called *Edit*, written by Gill Cross. When Gill left Cambridge, the job of developing the next version of *Edit* fell to me. It had to be written in Assembler, for speed and size, since it was a permanently loaded re-entrant program, shared by all jobs and timesharing sessions. This editor went into service in the middle of 1973, and I published a paper about it in *Software – Practice and Experience* in 1974 (text editing was still sufficiently novel at that time).

ZED

After the diversion of working on the database for SEAS 77, I returned to the text editor, for which there had been many suggestions for enhancement. In the end a complete re-write was undertaken, and the result was a line editor called ZED. I think it was Maggie Carr who suggested this name, as it was to be 'the last word in editors'. It was also particularly appropriate, because in 1978, the year it appeared, I played the part of a character called Uncle Zed in an amateur production of the musical *Salad Days*. A description of ZED was also published in *Software – Practice and Experience,* in 1980.

ZED was a programmable editor that could be used interactively or driven by a script, but in the beginning it was still a one-pass, line-by-line editor, in the same style as its predecessors. ZED was used for many text processing tasks other than simple editing. We found that users preferred to use the editor they knew rather than write special-purpose programs in a language such as SNOBOL, which was the only text-manipulating language available at the time. This tendency did not change until much later, when languages such as Perl and Python became available.

Being written in IBM Assembler, ZED was not portable to other systems. For this reason, a BCPL version was written (not by me) for use other than on the mainframe. I do not think it ever fully implemented all of ZED's features, but it was used for a number of years by ZED-lovers.

When the Computing Service took part in the Phoenix/MVS evaluation project with IBM in 1980, we used full-screen terminals (IBM 3270s) for the first time. These terminals were character-based, but could place the cursor and display characters at arbitrary points on the screen instead of just scrolling up from the bottom. All our own terminals at this time were real or so-called 'glass'

Teletypes. In order to demonstrate that full-screen working was possible in the Phoenix system, I made some additions to ZED that added basic full-screen support. These were later re-worked and, with the advent of our own full-screen terminals (Cifer 2632s, IBM 3278s, and later BBC micros with support for the SSMP protocol), became part of the standard ZED.

A portable screen-based text editor

The screen editing additions that were made to the ZED editor were not entirely satisfactory, because they were bolted on to an existing line editor. In addition, ZED was written in IBM Assembler, as it dated from the days when main memory was scarce. In 1986 I started the design of an entirely new editor that would be screen-based from the start. Although I wanted it to be portable, the editor’s main use was to be on the mainframe (which by then was an IBM 3084Q), driving terminals equipped with the means to support the *Simple Screen Manipulation Protocol* (SSMP).

The SSMP protocol and the BBC micro

SSMP was a protocol designed at the University of Newcastle for the support of screen-based applications over slow communications lines. The work of maintaining the screen image was split between the server that was running the application, and the client that was controlling the screen. Simple operations such as moving the cursor took place in the client without the need for an interaction, and text input was batched up and sent asynchronously, with the screen being updated locally. This avoided the need for character-by-character interaction, which was exactly what we needed because MVS was designed for IBM 3270 terminals, and they interacted in a similar way, page by page.

The BBC micro was the ideal base for an SSMP terminal. Shortly after the first of them arrived, the Computing Service’s *Microprocessor Support Unit* produced a plug-in EPROM (the *Phoenix chip*) that not only turned a BBC micro into a terminal emulator but also had file transfer facilities. As the cost of a BBC micro was comparable with the screen terminals that had previously been used, it quickly cornered the market.

The Microprocessor Support Unit had been set up in the expectation that many University users would be buying microprocessors and creating their own hardware applications. In the event, this never happened because complete microcomputers soon became available and were preferred. The name was eventually changed to *Microcomputer Support*.

At around that time, I was given a BBC micro terminal to play with. I wrote some small mainframe programs locally on it, using file transfer to copy them to the mainframe for execution. These were the first programs I had ever written that were never printed out. The era of program listings was drawing to a close. However, repeated file transfers were somewhat cumbersome, and it would obviously be better if a screen editor could be implemented for Phoenix.

For this reason, I was interested in adding support for SSMP to the BBC micro terminal emulator, and I managed to persuade Graham Dixon, a teaching fellow of Churchill College and a keen programmer, to implement it. He produced a second EPROM that worked in conjunction with the Phoenix chip. In the final configuration there was an automatic switch between the two when an SSMP application started or finished. BBC micros equipped with this pair of EPROMs were the standard Cambridge terminal for a number of years.

Some years later, I implemented an SSMP client in BCPL so that I could log in to Phoenix from more modern workstations and still use the full-screen editor. In various forms, this ran on Acorn systems and later under Unix. Martin Cole used the SSMP code from it in a PC version. Having an SSMP client on Unix turned out to be useful in the early Internet days when I was trying to use a computer situated in Oregon. The network was so slow at times that character-by-character interaction was very painful. I installed my SSMP-supporting editor in Oregon, and was then able to work with much less pain.

The E editor

When I started to design the new editor, I felt strongly that it should be written in a portable fashion so that it was not confined to the IBM system. Computers were becoming much more widespread, and I thought that users who used more than one computer should not be required to learn a new text editor for each one. The language I chose to write in was BCPL, because at the time this was the only suitable language available on all the machines on which the editor was intended to run.

The result was a text editor called E, which saw use on several widely-differing machines. On Phoenix it could drive both SSMP and IBM 3270 terminals; on VAX/VMS systems it supported SSMP terminals and standard DEC terminals; on VAX/Ultrix, Sun systems, and other Unices it supported SSMP terminals and any other terminal that could be described by the *termcap* mechanism; on Acorn's 32-bit processors (the Cambridge workstation running Panos and the Archimedes running Arthur and then RISC OS), and on IBM PCs running MS-DOS, it used the local screen and keyboard. It could also function as a line-by-line editor, which made it usable from ‘dumb’ terminals as well as in a script-like, non-interactive manner.

On all the full-screen terminals, the keystroke allocation was as similar as possible, so that the user was presented with a standard interface. For example, for users with Acorn Archimedes workstations, the keystrokes used while running E locally were identical to those used when logged on to Phoenix or one of the DEC systems using a terminal emulator that supported SSMP. The only difference the user saw was a slower response time on the remote systems.

The original implementation was done on Phoenix and an Acorn 32016 workstation, to ensure that it really was portable. Tony Stoneley did an initial port to VMS as another check. In addition to its portability, E was a very configurable editor. Almost all the keystrokes could be changed by the user to suit personal taste or habit. Following the success of ZED, the editor was intended from the start to be programmable as well as usable interactively.

E’s screen editing capabilities needed suitable facilities to be available in the terminal, but because it could be run as a line editor, it could be used from any terminal whatsoever. Phoenix users needed to learn only one editor, even if they did not always log on from screen-enabled terminals. The powerful line editing commands of E were also available from screen mode for performing systematic editing operations such as global exchanges.

E was widely used by the mainframe user community and was the standard editor taught to new users in the final years of Phoenix. It was also popular with some users of smaller systems such as VAXes, and was exported to several different institutions around the country.

Porting E to non-BCPL systems

The E text editor was written in BCPL because that was the only suitable language available on the mainframe at the time. However, the C language gradually displaced BCPL in Cambridge, especially after C was standardized. At the same time, a number of manufacturers introduced new models of computer with different instruction sets. Where once BCPL could be expected to be ported onto any new machine in the Computer Laboratory, this became no longer the case.

Translation of the code of E into ANSI C was an obvious way forward, though this was not a small project. Fortunately, a stopgap approach was possible. The 32-bit BCPL compiler for MS-DOS compiled not into basic instructions, but into a compact code known as CINTCODE which was then interpreted at runtime. I wrote, in C, an interpreter for CINTCODE that was capable of running E on any Unix system. By this means E was ported onto new DEC 5000 series workstations that used the Mips R3000 chipset and onto Sun 386i systems, one of which was my first personal Unix workstation. The performance of the interpreted version was, of course, slower than that of a fully-compiled version, but the power of the new workstation processors was sufficient to make it acceptable.

The successor to E

The CINTCODE solution for porting E to non-BCPL systems was not a long-term solution to its continued existence, but by the time I started translating the code into C, it was clear that the days of the IBM mainframe were numbered. The requirement of support for both SSMP and IBM 3270 terminals had constrained some of the facilities in E. I decided to abandon mainframe support in E's successor – rather boringly called NE (*New E*) – and assume that it would run only where character-by-character interaction was available. In practice, this meant Unix and Unix-like operating systems, though one version did run on MS-DOS, and for a while I had a version for Acorn's RISC OS.

NE has its devotees, and is still freely available in source form for anyone who wants it. It is my normal editor, and I am using it to write this memoir, using the same editing keystrokes as in the days of full-screen ZED, nearly 30 years ago. Like E and ZED before it, NE can be used as a line editor as well as a screen editor. The commands, which include various looping constructs, can be used during full-screen editing, and they have also occasionally been useful when a session's terminal characteristics have got into a strange state. However, I do not think that E or NE have been used for regular scripted text manipulation in the way that ZED was. More efficient text manipulation languages such as Perl and Python have taken over that job.

2017: *I am still using NE. Since 2009 it has been upgraded to support Unicode characters and UTF-8 encoding.*

Text formatting

Back in the days of the Titan, John and Judy Matthewman wrote a program called *The Matthewman Paginator*. This formatted documents for output on the lineprinter, which of course had only a single, fixed-width font. It was also possible to send the output to paper tape, which could be used to type the result on a Flexowriter. A friend of mine used this feature to create wax stencils in order to duplicate his PhD thesis. Observing how fiddly this process was, I produced my own thesis on a manual typewriter using carbon paper. Both of us had to insert Greek letters and other mathematical symbols by hand.

In the 1970s, word processing of various kinds was becoming widespread, and something was clearly needed for the Phoenix system. However, the Computing Service was unable to acquire a suitable package at an affordable price and on acceptable terms, so in 1979 I wrote a text formatter called GCAL. The code was written in BCPL and the name was an elaborate wordplay. Major features of Phoenix were always named after birds. An early Unix text formatter was called *runoff*, and the roadrunner is a bird that runs off. That name seemed rather too long for a command, so I abbreviated the Latin name, *Geococcyx californianus*, instead.

At that time Phoenix users still had access only to line-by-line terminals, so GCAL was not an interactive WYSIWYG word processor of the sort that appeared on personal computers a few years later. In concept it was similar to TEX, which was being developed at around the same time. However, GCAL was much less ambitious; I did not attempt to support mathematical typesetting, nor did I consider using any fonts other than those that were already available in the various printing devices.

GCAL read a source text interspersed with formatting directives (also called ‘markup’) and, like TEX, produced an intermediate code that was later interpreted by another program in order to drive the final output device. Unlike TEX, GCAL could also produce plain text output for lineprinters or for saving in a file for online inspection. There were basic directives to control the layout of the output, and these could be combined into macros to provide a higher-level interface. For example, a macro called *chapter* would start a new page, set a chapter title in an appropriate font, and increase the chapter number. We learned by experience that if a document was marked up using only the higher-level *generic* markup, it was much easier to format it for different output devices.

For some years GCAL was very popular and was used by many people, both in Cambridge and elsewhere, to typeset all kinds of documents, from letters to books. As printing technology changed, GCAL was updated to handle new kinds of output device such as dot-matrix and daisywheel printers. The Computing Service installed a daisywheel printer in its Bookshop and charged users by the page for its use. Although this device had only one font, the spacing between letters and between lines could be adjusted, giving the impression of a different typeface. For example, footnotes could be printed with the letters and lines closer together than the main text.

The arrival of laser printers heralded the end of single-font output. The computer scientists had a very early one that used a wet process involving bottles of toner, but I never had any dealings with it. The first one in the Computing

Service was about the size of a small refrigerator, and had its own parochial control language. I adapted GCAL to generate output for it, and it was used within the Computing Service, but I do not think it was ever offered to users. Then, in 1985, the first PostScript printer (an early Apple LaserWriter) arrived. This was the stimulus for some major enhancements to GCAL, both to enable it to take advantage of the newer printing technology, and to improve the facilities generally.

GCAL was used on computers other than the mainframe. In consultation with Acorn Computers I ported GCAL to their 32-bit processors, as a result of which it became widely used internally in Acorn. It was also ported (not by myself) onto several other types of computer in the Computer Laboratory for use by the computer scientists. One or two sites outside Cambridge also used GCAL.

Other uses of PostScript

The first PostScript printer was installed in a computer room on the ground floor of the Phoenix building, and connected to the network via its serial port. My office was on the first floor, so in the early days I spent a lot of time going up and down stairs each time I sent some test output. It did not help that error messages from the printer, which were sent out on the serial line, were discarded, so bug fixing was not that easy.

Once we had some experience of using PostScript printers, I felt that we should be able to put them to wider use than just printing text and simple graphics. I spent some time writing a description of the University's coat of arms in the PostScript language so that it could be automatically incorporated into documents. This found widespread use around the University, and I suspect it may still be in use.

I played around with PostScript's font-defining facilities, and produced a font with a few graphics characters, such as an open square box, that were not in the standard fonts. I do not think this was ever widely used, but the experience was useful when I got interested in typesetting music. More about that later.

The successor to GCAL

In 1989, when GCAL was nearly ten years old and had been much modified since it was first written, I re-wrote it in ANSI C, both to make it easier to modify in future, and to make it more portable. The new version, called SGCAL (*Son of GCAL*), also had additional features such as automatic hyphenation, kerning, and the better handling of paragraphs and pages.

By this time, PostScript was established as a standard page description language and there was no longer any need to support dot-matrix or daisywheel printers. Furthermore, PostScript has facilities for handling graphics as well as text, so I wondered about extending SGCAL in that direction. Having added some primitives for lines and curves, I thought about how best to provide for higher level constructs. I looked at Unix's *pic* command to get some ideas, and then wrote my own program and called it *Aspic*.

The original version of Aspic read a script containing a description of a line graphic in the form of commands like *box* and *circle*, and output text in the form of SGCAL input. I extended SGCAL to allow it to pass some of its input through an external program before processing it itself. This made it possible to mix text and graphics in a single source file. A few others made use of Aspic, but at the time when it appeared many people were no longer using the mainframe for wordprocessing, because suitable tools had become available for personal workstations.

SGCAL and Aspic were in service on the mainframe and were also installed on the Central Unix Service and made generally available in source form for use elsewhere. I used them in all my subsequent formal documentation, including many editions of the Exim manual and the Exim 4 book.

Using XML and DocBook

The story of how I came to write Exim is told in chapter 9. One of the goals I set for the Exim documentation was that it should be available in as many formats as possible. Using SGCAL meant that a plain text version, which could be searched with a text editor, was easy to produce from the same source that made the PostScript output, and the PostScript itself could easily be made into a PDF. Requests for *Texinfo* and HTML versions were met by writing Perl scripts to convert the SGCAL input appropriately, though they required continuous maintenance and were never entirely satisfactory.

A few years before I retired, when I started to think about handing Exim maintenance and development over to others, I had to figure out what to do with the documentation. Continuing with SGCAL, which was by then quite old and unlikely to be maintained, along with several lashed-up scripts, was not really sensible. The best prospect seemed to be to change to DocBook XML as the primary source, because there were free programs that could process DocBook into various output formats.

However, XML is not a format that is convenient to write or edit. For this reason, I chose to convert the source of the Exim documentation into a format from which DocBook could be generated rather than into DocBook itself. The idea was that DocBook would be the standard that underpinned everything. The way it was generated (the back end) could be varied, as could the way it was processed into the final formats (the front ends).

I discovered a program called *Asciidoc*, which generates DocBook from stylized plain text files, and for a couple of releases this was used. However, Asciidoc was not really designed for something the size of the Exim manual. Not only was it quite slow, being written in Python, but it had difficulty coping with all the typographic variations needed in something the size of a book.

In the end, I wrote my own preprocessor in C, called *xfpt* (*XML From Plain Text*). This is a program that generates DocBook XML from relatively simple markup. Going back to markup got rid of the ambiguities that had caused trouble with Asciidoc's plain text approach. I implemented a macro facility so that only a few primitives were needed in the program itself. For the next few releases of Exim, *xfpt* generated the DocBook, and software from elsewhere generated the output.

Front end woes

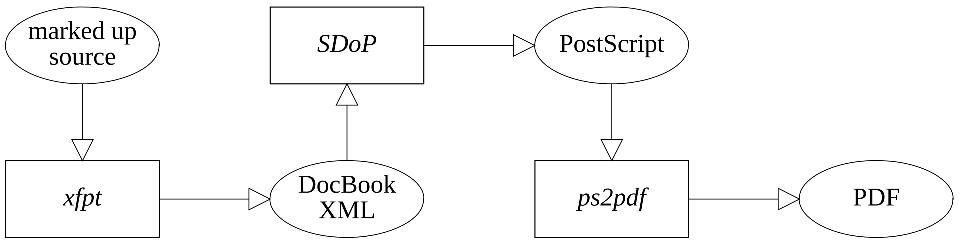
Unfortunately, that was not the end of the story. There were free programs for processing DocBook into all the required output formats, but I was not entirely happy with the results. For some formats, the problems were fairly straightforward and could be solved by writing pre- and post-processors (for example, to change 'typographic' quotes into ASCII quotes for plain text output). However, for PostScript and PDF output the issues were more serious.

A two-stage process was used for these types of output. First, the source was turned into *formatting objects*, and then a program called *fop* (*Formatting Object Processor*) processed these to produce the final output. At that time *fop* was still in development, and running it on the Exim documentation produced a whole slew of warning and error messages, though the output did not seem to be affected.

One annoying problem was that *fop* was written in Java and ran very slowly – on my workstation it took 2.5 minutes to process the Exim manual. Furthermore, the quality of the typesetting was far below what I was used to with SGCAL. It seemed that twenty years of computer typesetting experience was being ignored. The hyphenation was overly aggressive, *orphan* and *widow* lines abounded, headings could appear at the bottom of pages, and there was no way of using anything other than the roman font in index entries. This last point was for me the final straw: many technical terms in the Exim manual were always printed in either italic or bold face, and showing them otherwise in the index was just plain wrong.

So I gave in, and wrote another program, again in C. This one is called *SDoP* (*Simple DocBook Processor*). It reads DocBook XML, and writes PostScript. It is ‘simple’ in that it supports only a subset of DocBook’s facilities. In the beginning, this was just those features needed for the Exim documentation, but later I discovered that a subset of DocBook called *Simplified DocBook* had been defined, so I upgraded SDoP to handle most of that.

‘Simple’ also applies to the way SDoP operates. It does not check its input against a DTD, because there are XML validation programs that can do that. Also, it does not use stylesheets. Instead, there are configuration settings that can be embedded in the source for adjusting things such as the fonts used for particular entities, or the way the table of contents is constructed. SDoP can process the Exim manual in 2.5 seconds (60 times faster than *fop*), and the output is much nicer.



Exim documentation processing

The SGCAL version of the Exim documentation had contained a few Aspic line graphics. These were removed during the changeover to DocBook, and never reinstated. However, I did convert Aspic so that it could be used with SDoP. I removed it from the SGCAL distribution and made it stand-alone, adding options for two new output formats: encapsulated PostScript, and Scalable Vector Graphics (SVG). The former is supported by SDoP, and was used to create the figure above.

Different times, different strategies

It was interesting coming back to writing typesetting code twenty-five years after I first did it. Back in 1979 on Phoenix/MVT, main memory was very scarce. The entire machine had only four megabytes and no virtual memory. GCAL was written as a one-pass processor. The source was read line by line, and the output was written before moving on to the next line.

SGCAL was not very different (even though it was for a virtual memory environment) because it was really just a re-implementation of GCAL. It did process paragraph by paragraph rather than line by line, but it still used a single pass. Support for forward references, which I needed for large documents, had to be done by running SGCAL twice (there was a special option for this). Title pages, tables of contents, and an index were created by post-processing auxiliary output that SGCAL produced as it went along.

By the time I started work on SDoP, main memories of hundreds of megabytes were common, so I was able to code it in an entirely different way. The entire source is read into memory and then scanned a number of times as the code processes it. Finally, an output scan writes the entire formatted book as a single PostScript file. Despite this radically different way of working, some of the typesetting algorithms (for example, those that lay out paragraphs and determine hyphenation) are the same ones that I developed for SGCAL.

Another thing that had changed significantly since the 1980s was character encoding. Back then, in an English-speaking environment, ASCII or EBCDIC code was mostly sufficient. Then various versions of the ISO 8859 code were brought out, each containing a different selection of additional characters (such as accented letters) for use in different European countries. However, the underlying problem of using only eight bits to define a maximum of 256 characters in any one code meant that it was hard to mix encodings (for example, to mix French and Hungarian in the same document). Furthermore, there was a demand for encodings of more exotic, non-Roman, alphabets.

The long-term solution was to move to a character code of more than eight bits, which allows for thousands of characters in a single character set. By the time I started work on SDoP, Unicode was well established and being used by a lot of software, so I decided that SDoP would be a Unicode application. Its input uses the UTF-8 encoding, which is compatible with ASCII for the first 128 characters. Other characters can be specified either by the appropriate UTF-8 sequences, or using escape codes such as `&#x200B;` to specify a character by the number of its Unicode code point.

SDoP automatically interprets Unicode code points when generating PostScript so that the user need be aware only of Unicode encodings. The standard PostScript text fonts contain more than 256 characters, though only 256 are accessible from an individual ‘binding’. SDoP gets round this by generating PostScript that binds each font twice, with different character encodings, so as to give access to all its characters. PostScript’s special fonts are used for symbols that are not in the text fonts.

8. Acorn computers, music printing, and athletics

Sometime in the early 1980s I wrote to my parents ‘Today I bought a computer’. It seemed really weird. Up till then, a personal computer was like a personal jumbo jet – theoretically possible, but unbelievably expensive.

The computer I bought was a Sinclair ZX81, which, when connected to a portable cassette recorder and an old television, allowed me and my family to play a few primitive games. I poked around on the machine and learned a bit about its machine code, but I never had the inspiration or the patience to implement any real programs in the constricted 8-bit environment.

The BBC micro that we bought for Christmas a few years later was more advanced. This at least had floppy disc drives and a full-sized keyboard, and the games were more challenging. Despite previous success with the original moon landing program, I never did learn how to land the aeroplane properly in *Aviator*, which was a surprisingly sophisticated flight simulator. With an Epson FX-80 dot-matrix printer, the ‘Beeb’ made quite a good wordprocessor, and my letters to my parents started to look a lot better than they had done in the days of the manual typewriter.

I wrote a few toy programs in BBC BASIC, one of which, *Whack-a-Mole*, was published (as source text) in one of the computer magazines of the day. The game was based on a delightful mechanical arcade game that we had come across on Pier 39 in San Francisco on one of the American study tours. For several years, readers would laboriously re-type programs published as text in magazines. There was no public Internet providing free downloads in those days.

Many early home computers did not even have floppy disc drives, and ready-made games were initially published on cassette tape. When the first version of *Elite* came out, it broke new ground by requiring a disc drive, which it made use of while the program was running, another innovation.

Music has always been one of my hobbies. The Beeb had quite a sophisticated sound system for its time, and I played around with programs for making it play music from scores encoded as text files. This never went very far, but it influenced what I did later in music typesetting.

Acorn’s first 32-bit system

The BBC micro was designed from the start with an interface that allowed a second processor to be connected. This was called the *Tube*, and the idea was that the Beeb would act at as a front-end input–output system for a number of different processors. The first of these was a second 6502 processor, though I never had occasion to use such a system.

After personal computers started to become popular, many makers upgraded from 8-bit to 16-bit processors. Acorn skipped this generation, and went straight to 32-bit systems. Their first offering used a National Semiconductor 16032 processor (also known as 32016). It was offered both as a freestanding second processor for the Beeb, and packaged up into a single box called variously the *Acorn Business Computer* (ABC) or *Acorn Cambridge Workstation*. The operating system was written in Modula-2 and was called Panos, after a restaurant that the design team patronised. Panos was a single-user operating system that provided a command line interpreter, a screen-based text editor, and compilers for FORTRAN 77, C, Pascal, and LISP.

There was also a BCPL system, which was useful because Acorn wanted a text formatter for Panos. A deal was negotiated that involved me porting GCAL, as a result of which I acquired a 32016 second processor. At last I was able to run and develop 'proper' programs at home. This was the start of a consultancy relationship with Acorn that lasted for a number of years.

The ARM processor

Acorn's 32016 systems were not a success, and few were sold. They realized that the future lay in windowing operating systems, but the available processors were too slow for their purposes, to a large extent because they supported very complicated instructions. Having learned about the growing interest in *reduced instruction set* (RISC) processors, Acorn went ahead and designed its own. This was the first ARM (*Acorn RISC Machine*); later, ARM (now renamed *Advanced RISC Machines*) was spun off as a company in its own right.

2017: *While Acorn languished and eventually disappeared, ARM became a highly successful company whose chip designs are widely used.*

The first ARM chips were built into second processor boxes that could use a Beeb for their input and output. I do not think that any of these were ever sold, but they were fairly widely distributed to developers. A crude operating system, similar to the Beeb's, was made available, along with a text editor and some compilers. BCPL was again among these, and I was able to port GCAL without difficulty.

The first product that Acorn released containing an ARM chip was the Archimedes computer. No longer an add-on to a Beeb, it had its own I/O hardware. The operating system was called Arthur, derived from the Beeb's OS. There had been an intention to ship the Archimedes with an operating system called ARX, which was a pre-emptive, multitasking, multithreading, multi-user (that is, ‘all-singing, all-dancing’) system. However, this design was over-ambitious, and ARX never saw the light of day. Arthur was hastily implemented so that a product could be released. Rumour had it that the name Arthur stood for ‘ARX by Thursday’.

Arthur Norman and Alan Mycroft, lecturers in the Computer Laboratory, developed a C compiler for the ARM (*Norcroft C*), initially as a teaching tool for computer science. This supported the recently standardised version of the C language. Acorn liked the result, and decided to adopt and market it. I was asked as a consultant to review the draft documentation, and also to write tests for all the library functions. As a result, I learned a lot about standard C.

I had the manuals and learned the details of the ARM assembler, but I never wrote anything other than small auxiliary functions or tests. Although I did not realize it till later, my time as an assembler programmer was effectively over.

The Arthur operating system had a simple windowing add-on interface, written in BASIC. Later, this was rewritten and integrated into the operating system, which was renamed RISC OS. The hardware was developed, with a number of different Archimedes, and later Risc PC, models being released, all of which were software-compatible. Over the years I used a number of these machines, both at work and at home, and wrote or ported a variety of software for RISC OS. I used BCPL initially, but changed over to C when it became clear that BCPL support was waning.

RISC OS was a nice system in which to program. The interfaces to the OS and windowing system were clean and well documented. Fairly early on, Acorn published a style guide for windowing programs, and almost all applications followed it. This made for consistency in the way programs interacted with users, thus ensuring that RISC OS was an also easy system to use.

For some years I did occasional further consultancy work for Acorn, mainly checking the software and documentation for various releases of the C system, though I also wrote a program called *MakeModes*, which generated the data necessary for the different screen modes that could be used on different makes and types of monitor. This was intended as an internal testing tool, but somehow it escaped, and a number of Acorn owners made use of it.

Music typesetting

One of the student projects I supervised in the early 1980s was an investigation of music typesetting. At that time, the only available graphics hardcopy output device was a pen plotter. The student wrote a program that could format a single stave of music which was then drawn, laboriously, on the plotter. However, not long after this, the first PostScript laser printer arrived, and I realized that here was a much better device for printing music. At the time, my children were playing recorders at school, and we used to play together, so I aimed for something that could print out simple recorder arrangements. First, however, I had to develop a font of musical characters.

The first PostScript product (the Apple LaserWriter) had a number of standard built-in outline fonts. These were so-called ‘Type 1’ fonts, encoded in binary and encrypted so as to keep the details of how they worked secret. I think this was a mistake on Adobe’s part, because it immediately caused other companies to develop their own types of font in competition (for example, TrueType fonts) so the world now has to cope with several different font formats. After a few years, Adobe relented and published the specifications of Type 1 fonts, but in the early days, a PostScript user could create only a ‘Type 3’ font, in which each character is defined as PostScript code. I set about doing this for musical characters (notes, rests, clefs, and other useful glyphs, including stave fragments). The LaserWriter had a resolution of 300 dots per inch (dpi) and I was pleased to discover that a Type 3 font looked pretty good when printed on it. Then I started thinking about the typesetting problem.

Music typesetting was a personal project, of course, not part of my work for the Computing Service, and I worked on it at home in my own time. The development machine was an Acorn 32016 second processor running Panos, and I wrote in BCPL, partly because I was fluent in it, and partly because I wanted to make the program available on the University’s mainframe, which at that time did not have a C compiler. (Panos did support C, but with a pre-standard compiler.) It is always hard choosing a name for a program. I modestly called my music typesetter *Philip’s Music Scribe*, later changed to *Philip’s Music Writer*. For consistency, I use the later name (abbreviated as PMW) in what follows.

Encoding music for printing

I had attended a seminar about the Oxford Music Processor, another program that used a pen plotter for output. Its input design was based on the positions of the keys on a computer keyboard, the idea being that you could ‘play’ the notes. I did not like this idea at all, because I knew that there were many different keyboard layouts around. I thought it would be better to use input characters that related to what they represented, so I returned to the encoding that I had previously started working on for playing music on the BBC micro.

I wanted something concise, which would also be easy for a musician to learn, so I settled on the single letters A–G for notes, using upper case for minims (half notes) and lower case for crotchets (quarter notes). Plus and minus suffixes doubled or halved the note length for the other note values, and there were suffixes for raising and lowering the pitch by octaves (with a settable default). The other notation was as memorable as I could make it, for example, a vertical bar for a bar line and a sharp sign (#) for a sharp. There being nothing in the computer's character set resembling a flat or a natural, I chose the adjacent two characters on the Acorn keyboard (dollar and percent).

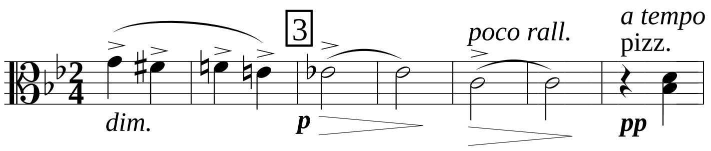
Some people I talked to about this expressed surprise that I was not considering taking input from a musical keyboard, until I pointed out the two main drawbacks. Firstly, it is impossible, even for the best keyboard players, to play absolutely precisely in time. If realtime capture is attempted, note lengths have to be rounded, which is not easy, especially with anything other than simple musical lines. Other people have done work on this, but I felt it was a distraction I could do without. Using a musical keyboard to indicate pitch only, with some other way of indicating the note lengths, is another possibility, but that seemed overly complicated.

Another reason for preferring encoded text input was that for all but the simplest of music, the notes are only a small part of what is printed on the page. There are often quite a few performance marks and words as well, which could not be input from a musical keyboard. My conclusion was that one might as well input the whole encoding as text from a computer keyboard. This had the advantage that any computer could be used to prepare an input file for PMW, and also that text-based facilities such as macros could be provided.

2017: *Here is a very short example of some PMW input, shown with each bar on a separate line, though that is not a requirement (input is in free format). The items starting with an ampersand are macro calls; the macro definitions (not shown) specify common text strings and set their sizes and fonts in a similar way to the 'dim.' item.*

```
Key Gm Time 2/4
Justify top left Unfinished
[stave 1 alto 1]
[slur/a/co2] "\it\dim."/s2 [\>\] g#f |
%f%e [endslur] |
["3"/d6/l2] &p/l8/u2 > $E_ [\\] |
E >/h |
&pocorall > C\>\_ |
C >/h |
&pizz &atempo &pp r (b`d) |
[endstave]
```

This is the output that is produced from the input above.



The first version of PMW

The BCPL version of PMW read an input file and wrote PostScript. It also had a mode whereby it displayed the music on a screen, using the Beeb’s bit-mapped, fixed-width screen characters. There were no curves; slurs had to be approximated with straight lines. This crude preview did not have enough resolution for checking that everything was in the right place, but you could at least be sure that the right notes had been entered before sending the PostScript output to be printed.

PMW ran initially on Panos and under Phoenix/MVS, where the preview mode could be made to work on a BBC micro terminal. A number of mainframe users made use of it, and one of the students, Mark Argent, became interested in the details of how it worked. He felt that some of the musical shapes in the font could be improved. To this end he taught himself PostScript, which was quite a feat as he had never done any computer programming before. Mark contributed a number of improvements and some additional characters, and later used PMW to typeset music for commercial publishers.

I ported PMW to the ARM systems when they appeared, and in particular to the first Archimedes. Word had got round about this, and I received some enquires from people who arranged music as a living as to whether I would sell and support the software. In June 1988 I installed PMW at the home of the first customer, Clifford Bartlett, who did a lot of early music arrangements. He had bought an Archimedes and an Apple LaserWriter, which was then quite an expensive investment. I had been using the networked LaserWriter at work (and paying per page for ‘private use’), so I had no experience of driving one directly from the Archimedes over a serial link. Fortunately, I did have some experience of using a serial link for a terminal, so I was able to create an appropriate application, and debug it at Clifford’s house.

I did not get my own laser printer at home until November 1990, when I bought one of the first Apple Personal LaserWriters (still with a resolution of 300 dpi) costing £1789.98. I used this printer until 2003, at which time it was replaced with an ethernet-connected, 1200-dpi printer/scanner/copier for around one-third of the price.

Two more customers bought copies of PMW in August and December, 1988. Of course they soon discovered various bugs and missing facilities, and so I found myself maintaining and rapidly developing the software, with a growing user base.

I had renewed contacts with musical friends in Cape Town when I started regular visits to my increasingly frail parents. When ‘in town’ I was able to sing with St George’s Singers, a choir run by Barry Smith, the cathedral’s organist and director of music. In November 1988, my father died and I flew out for the funeral. During this visit I found the Singers were rehearsing a new work by a South African composer, Peter Klatzow. This was his Mass for choir, horn, marimba and strings, to be performed the following Easter. The photocopies of the hand-written vocal scores were not entirely clear, so when I got back to Cambridge, I set about typesetting the Mass myself.

This was my own first use of PMW for a substantial piece of music, and, like the other users, I came across bugs and the need to extend the program. When I had done the first movement, I sent it to Peter, and fortunately he liked it, and approved of what I had done without asking permission. The remaining movements were completed in time for the first performance on Easter Day 1989, when I was again in Cape Town and able to take part. Later, the vocal score was published from my page images.

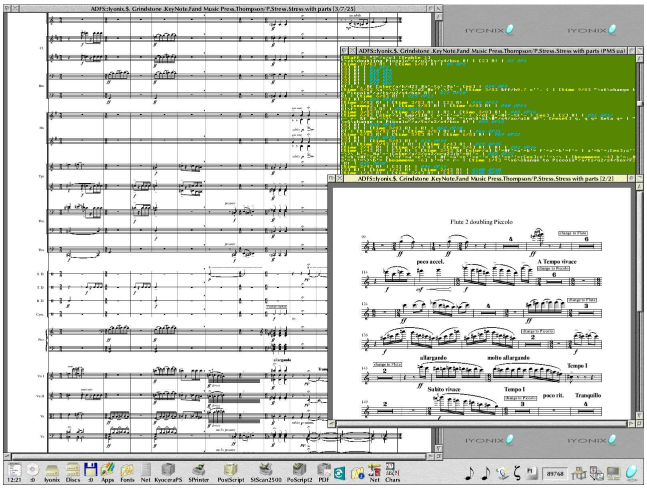
2017: *A few years later I typeset the musical examples for Barry Smith’s biography of Peter Warlock.*

Another use I found for PMW was typesetting early recorder music for Theo Wyatt, who ran a small publishing business selling music that was cheaply copied using stencils. Theo’s business model was interesting. I suppose you could call it a sort of cooperative. He had a number of people creating page images, and the profits he made, after he had taken his cut, were divided according to the number of pages each person had contributed. I did rather well out of this, because I could generate pages of this relatively simple music much faster than anyone could by hand.

PostScript printers were expensive for a long time, and it was only those using PMW commercially who could afford the investment. Most Archimedes users had dot-matrix printers. Fortunately, I was able to modify PMW so that it produced output that could be sent to a dot-matrix printer, as an alternative to PostScript. The result was rather coarse, but still usable for performance copies, which was what most of these users wanted it for. This extension gained me a number of users, some through a company called ElectroMusic Research, who sold software for Acorns. They branded it as *Professional Music Scribe*. In total, I sold 20 copies of the first version of PMW, and EMR sold around 50.

Acorn fonts and a re-write for RISC OS

As part of developing the Archimedes software into a full-blown windowing system, Acorn implemented their own outline font system. This made it possible to display typographic characters on the screen more or less exactly as they would appear when printed, within the limits of the screen resolution. By this time too, a fully-functional standard C compiler was available, so I decided to re-implement PMW in C as an Archimedes windowing application that could display accurate preview pages on the screen. This meant that there would be no further development of the Phoenix/MVS or Panos versions.


Music on an Acorn desktop

The first thing I had to do was to convert the PostScript font into an Acorn outline font, not only for the screen display, but also so that output could be passed through the RISC OS printer drivers, which were capable of driving a large variety of printer hardware. Then I converted the typesetting code by hand, adding new code around it in order to create a proper windowing application. I retained the ability to generate PostScript output directly, partly for compatibility, and partly because comparing the PostScript output with previous runs was my main means of testing the program. However, PMW could be compiled without this facility, and most of the copies I sold did not need it.

The new version was released towards the end of 1992. Richard Hallas, who had been using PMW in his one-man typesetting business for some time, was one of the first users. He was not entirely happy with some of the characters in the Acorn font, so he proceeded to tidy them up for me. Later Richard provided some additional characters and also an entirely separate font of extra musical characters that were useful in normal text when writing about musical concepts.

I continued to extend and maintain PMW till the end of the decade, though the rate of change slowed as the program matured. My user base increased by another 30 or so, the last new user being a student in 1998. EMR also sold the windowing version of PMW, but sadly, mainly owing to the illness of its proprietor, the company closed down in 1993 or 1994. I did not try to find a replacement.

One major extension to PMW was the addition of a facility for playing the music. This was not for any kind of performance, but as a 'proof-hearing' tool. The ear is much better than the eye for checking the notes in a piece of music; elementary mistakes such as missed accidentals can easily be overlooked on sight, but stand out strongly when the music is played. The RISC OS sound system had only eight channels, so only a few musical lines could be played at once. However, I also implemented output to a MIDI interface that could be connected to a MIDI instrument – in my case a keyboard – which allowed for many more voices and different instrumental sounds.

Rivals and demonstrations

I developed PMW to the point where it was capable of typesetting almost any conventionally notated music. I was not the only person in the world tackling this problem; indeed, I was not the only person in Cambridge, where, to my knowledge, at least three other music typesetting programs were written. One of these was *Sibelius*, originally developed on an Acorn by the brothers Finn, and subsequently turned into a very successful commercial product.

I never saw Sibelius as a 'rival' for PMW. It takes a WYSIWYG (point-and-click) approach, which is much better suited to the casual user. The PMW input language is concise, but has to be learned, and devoting the time to learning all the details is really only worth it if you are going to be doing a lot of typesetting. However, I cannot deny that I was pleased when one or two people bought PMW in preference to Sibelius, having looked at both.

Somebody once said that people are either *doers* or *describers*. The former prefer WYSIWSG interfaces, which you can make do things as you watch, whereas the latter (a smaller fraction of the population, I think) prefer to create files that describe their requirements, that is, they like markup languages. The doers prefer applications like Microsoft Word and Sibelius; the describers go for

TEX and PMW. I think it is fair to say that, in general, people who compose music or who want to make arrangements just for performance are happy with a WYSIWYG approach, but those who are interested in creating high quality page images for publication (in effect ‘music engravers’) prefer PMW because it gives very precise control over the placing of every mark on the page.

In 1988 I was contacted by the Center for Computer Assisted Research in the Humanities, which is based at Stanford University in California. At that time they published an annual directory of music-related computer research, and as part of this they listed all music printing systems they knew about. They also sent out test pieces to the program authors, inviting them to send back their output for inclusion in the directory. Output from PMW was included for the first time in 1989, and I was pleased to note that it was typographically better than several other contributors. I contributed sample output to the directory for the next three or four years, by which time there were a number of high quality programs available (for various operating systems), all of which could set the increasingly complicated test pieces very well. At that stage, the printing of samples for comparison was judged to be no longer useful. However, PMW did get one further mention by the CCARH, in a book called *Beyond MIDI, The Handbook of Musical Codes*, published in 1997. I contributed a short chapter describing the basics of the PMW input encoding.

Music exams

In 1992 I was contacted by a lecturer in the Faculty of Music who asked if I could typeset music examples for the Tripos examinations. The music had previously been sent away from Cambridge to a commercial typesetter, which was not only inconvenient, but also expensive, and of course there was also a confidentiality issue. For several years thereafter I used to set these music examples, until eventually the examiners had access to facilities that allowed them to do it themselves.

This project turned out to be trickier and more interesting than it seemed at first sight because some of the exam questions required the music to be set incorrectly, or with parts missing, so that the examinee could correct it. Just omitting things (for example, accidentals) was easy, but printing notes and bars of the wrong length, for example, needed a bit more ingenuity. Luckily, PMW has facilities that make this relatively straightforward.

Porting PMW to Linux and beyond

As the twentieth century drew to a close, development activity on PMW more or less ceased. I myself was using it for occasional jobs, and there were still people using it commercially, but there were no new users, and no further development, though I still had to fix an occasional bug. The final release happened in December 2001.

In the autumn of 2001 I acquired a laptop, the first machine with PC architecture that I had owned. I bought it to use for slide displays, because by then I was running training courses and giving talks about Exim (see chapter 9) in my day job. Although others continued to develop the hardware and software after Acorn ceased to exist as a company, it was clear that, for me at least, the days of running a RISC OS machine were numbered.

As I was by then a developer of free software for Unix-like systems, and had never used Windows, I ran Linux on the laptop, and some time later I set out to port PMW. It was not difficult to create a command-line version that produced PostScript output. Almost all I had to do was to cut out all the screen-handling and other code that was specific to RISC OS. However, the widths of characters had been obtained from the Acorn's font system, so I had to install some new code to use AFM files instead. Luckily, I was able to copy most of this from SGCAL.

Previewing PostScript on the screen was not the problem it had been a decade earlier; *GhostScript* did the job nicely. In November 2003 I released this version of PMW as free software under the GNU General Public License, and put the source on a web site. A few people found it, and once again I started to receive feedback. For about a year, until I bought a new home workstation (finally retiring the RISC OS box), all the support for this revived PMW was done on the laptop.

Two major upgrades were soon requested. Whereas the Type 3 PostScript font looked satisfactory when printed, it was not so good when displayed on the screen, because of the much lower resolution. Fortunately, by this time Adobe had released the specification of Type 1 PostScript fonts, and there were some utilities for creating them from text files. As well as the data about the shapes of characters, the Type 1 input contains additional information that the renderer uses to improve the way they look on a screen. I converted the music font by hand, adding appropriate 'hints', and the result was indeed much better.

The second upgrade was the restoration of music playing facilities using MIDI. Under RISC OS, this required a hardware MIDI interface and a MIDI instrument. I discovered that there was now a defined format for MIDI files, and also that there was a free application, called *Timidity*, which could play them using the normal Linux sound system. It was a simple matter to modify the old code to generate a MIDI file.

At that point, the code languished for a few years, but after I retired I did a thorough re-write, finally expunging the remnants of the RISC OS code, and removing a number of 32-bit dependencies that had prevented PMW from running on 64-bit systems. While I was doing this, I received an email from an Egyptian user, who asked if I could modify the program so that it set the music from right to left instead of left to right. Apparently, in the nineteenth century, missionaries who went out to Egypt wrote out hymn music this way, to make it easier for readers of Arabic. Thanks to the generality of the coordinate system used by PostScript, it turned out not to be too difficult to do what the user wanted; I do not know if any other music typesetting software is capable of doing this.

Because PMW is now free, and available from a web site, I no longer have any idea how many people are using it. Several have emailed me, so I imagine there must be others who do not feel the need to get in contact. At least one user compiled the source code for use on Windows. Since it is all standard C, it should compile on any computer with a suitable compiler and runtime system.

PMW with hindsight

Many of my software projects have extended and expanded far beyond what I envisaged when I started, and PMW is no exception. What started out as a little project to print simple recorder music ended up as a fully-fledged typesetter, used for some major printing projects. In the process, I learned a lot about printed music, much of it from Gardner Read's book *Music Notation*. John Line had lent me this book when he discovered that I was working on music typesetting, and it soon became so invaluable that I had to buy my own copy.

After Clifford Bartlett had been using PMW commercially for a year or so, I asked him if it had come up to his expectations, as compared with writing out arrangements by hand. His response was interesting. He had initially hoped that it would save time, but this had not proved to be the case for instrumental parts. However, having encoded the parts, a score could be produced without further work, and that did save time. Furthermore, the score and parts were guaranteed to be consistent with each other. He also appreciated the ability to make changes easily, and the ability to transpose the music.

Other users of PMW have singled out the fact that it gives precise control over how marks such as slurs are drawn, and where most items on the page are placed, though to do this the user has to learn about the many available options.

Scoring athletics matches

In the 1990s my youngest son was a member of the Cambridge and Coleridge athletics club, and my wife and I became volunteer parent helpers. She was a coach and field judge, and I was asked to help with the scoring for home matches. This always involved a lot of frantic arithmetic when the last event had finished, to determine which team had won the match.

One year C&C were hosts to the final of the East Anglian League, in which there were both adult and junior events in several age groups. The final match was contested by more than the usual number of teams. I decided that we scorers needed some computer assistance, so I wrote a multi-windowed program that would take care of all the arithmetic, and also provide nicely-printed score sheets. On the day of the match I carted the Risc PC workstation, the CRT monitor, and the personal LaserWriter to the athletics track and, with another helper, put it to good use. A few minutes after the last event we were able to hand laser-printed score sheets to the various team managers, who were suitably impressed.

It had not gone without a hitch, however. I had written the program so that the team declarations (the lists of which athletes were competing in which events) had to be input before any scoring could be done. Then, for each event, typing the team letter pulled up the athlete’s name automatically. The problem was that the team managers waited till the last minute before the match started before giving us the declaration sheets. This meant that it was some time before we could actually process results, because we were putting in all the names first. Once we did start, everything went very fast, but the teams were used to hearing announcements and seeing result slips on the noticeboard soon after the events finished, and not in a great burst after an initial delay, so there was a bit of angst when it did not happen in the expected way.

I used my program for some other matches afterwards. For smaller events, the initial delay was not so noticeable. However, the hassle of dragging all the kit to the venue was certainly tiresome. Today one could use a laptop and a small USB printer, which would be easy to transport, and perhaps a standard spreadsheet could be set up to do the job.

9. Email, the DNS, and regular expressions

Before the start of the Central Unix Service in 1990, I was just a user of email. At the time when the academic institutions in the USA were experimenting with the early Internet, the intention in the UK was eventually to use what were then still developing International Standards. The Joint Academic Network (JANET) was set up using X.25 networking and the so-called *Coloured Book* protocols (each published with a different colour of cover). These were interim protocols, to be used while waiting for International Standards to take over.

The email protocol was largely based on the Internet's RFC 733, later updated for better compatibility with RFC 822, with the transport piggy-backed on the file transfer protocol. Very soon there was a requirement for a gateway so that email could be exchanged with the Internet, to enable academics in the UK and the USA to communicate with each other. The gateway was set up at the University of London Computer Centre (ULCC). The fact that the message structures in the two networks were similar no doubt made it easier to program the gateway.

There was, however, one major difference. The UK had chosen to write host names and email domains the opposite way to the Internet. My email address, for example, was *ph10@uk.ac.cambridge.phoenix*. Needless to say, this led to a number of problems, of which more later. There was also an insistence that names should be 'meaningful' and not unnecessarily abbreviated, hence the use of the full names *cambridge* and *phoenix* above. I used to tease my networking colleagues by asking why, if abbreviations were so bad, my address could not be *ph10@united-kingdom.academic.cambridge.phoenix*, but nobody ever took this bait.

A problem arose, though, when the network came to be implemented. It was discovered that these long names were too wide for convenient display on the network monitoring screens. This was overcome by allowing each name to have a 'short form' as a synonym, for example, *uk.ac.cam.phx*. However, the two forms could not be mixed: *uk.ac.cam.phoenix*, for example, was not allowed.

There was a strict injunction that short form names should not be promulgated for general use. Of course, the injunction was ignored and the users used them all the time. Who is going to type 'cambridge.phoenix' when 'cam.phx' will suffice? When a short name was encountered, software that processed it was supposed to rewrite it into the equivalent long form. Some software did, some did not.

The short forms were limited to 18 characters, and this was a problem for subdomains. For example, the Department of Applied Mathematics and Theoretical Physics where I studied for my PhD was and is universally known as DAMTP, but it was forced to use the short domain *uk.ac.cam.amtp* so that there were at least three characters left (after the next dot) for its host names. The double naming system was undoubtedly a cause of misunderstanding and confusion for users.

Email outside Cambridge

I started taking a serious interest in non-local email in the late 1980s, when it became possible to use it to correspond with a friend in Cape Town who was keeping an eye on my ageing parents. In the early days, a message sent from Phoenix went first to the Internet gateway at ULCC, from where it was forwarded to a computer in Portland, Oregon. Twice a day there was a dial-up connection from Rhodes University in Grahamstown, South Africa for the exchange of email. The message was then passed over the South African academic network to the University of Cape Town, where my friend worked.

This was a dodgy process. The ULCC gateway was rapidly becoming swamped with an increasing flood of email, and delays of many hours and sometimes days were not unknown. There were also issues with the connection to South Africa. We took to numbering each message so the recipient would know if one had been lost. On one occasion, for a reason I can no longer remember, I contacted the postmaster of the mail system in Oregon, and this was how I came to meet Randy Bush, who was to change my life in several ways.

Email on the Central Unix Service

When the CUS was starting up in 1990, I was asked to install *Elm*, a screen-oriented *Mail User Agent* (MUA) as an alternative to the primitive line-oriented *mail* command that was the default. Having done this, I found myself taking over the rest of the email system as well. The CUS operating system was SunOS (later migrating to Solaris), and the *Mail Transfer Agent* (MTA) was *Sendmail*, using the *Grey Book* protocol over X.25.

In order to sell its hardware to British academic sites, Sun had commissioned another university to get *Sendmail* working with the Coloured Book protocols. How it operated was horrendous. The JANET central management ran a *Name Registration Service* (NRS). This maintained a file that matched service names to X.25 addresses, just like the ‘hosts file’ of the early Internet. The NRS data was downloaded to hosts periodically (typically once a week). It enabled hosts to connect to one another by name for remote logins and file transfers (which included email). So far, so good. The horror came with email addressing and routeing – a collection of scripts processed the NRS data, extracted the names of all the valid email domains, and generated a raw *Sendmail* configuration file that was about 1,500 lines long. This was known, confusingly, as *UK Sendmail*.

Part of the reason for the great length was the desire to handle partial domains. The valid service names were constrained so that no component could be a name that was further up the tree. For example, because there was a university called *newcastle*, no host could have that name. The idea was that you could send email to *user@newcastle* and that would be unambiguous.

Sendmail operated by pattern matching domains and transforming them again and again until it arrived at something it knew how to deal with. An ‘ordinary’ configuration file might be some tens of lines long, so *UK Sendmail* was a monster. Nevertheless, it worked satisfactorily within the confines of JANET. The problems arose with mail destined for elsewhere – that is, the Internet. There were two issues: domain name ordering, and, a bit later, the use of the Internet’s mail-transfer protocol, SMTP.

The problem of domain reversal

Users who were corresponding with people outside the UK would often write the addresses in Internet order (toplevel domain last), because that is what they had been told, or seen in an overseas publication (not a web page – this was before the web, and in any case we were not yet connected to the Internet). To try to deal with reversed domains, the *UK Sendmail* configuration had a list of toplevel domains (*com*, *edu*, *org*, *uk*, etc.) and one of its pattern matches tried to find which end of the domain had the toplevel component. The domain was then reversed if necessary.

When I first discovered this I felt sure it would lead to grief. Sure enough, one day somebody sent email to *uk.tele.nokia.fi*, a domain that has toplevel components at both ends (*fi* is Finland). I cannot remember whether *Sendmail* went into a loop with this – I suspect not, because it must have had protection against pattern matching loops – but it certainly could not handle this kind of thing. We also had some local two-letter domains (deliberately short because of the length constraint mentioned above) that caused problems: *cl* for the Computer Laboratory, and *ch* for Chemistry. Unfortunately, these were the country codes for Chile and Switzerland.

JANET converts to Internet Protocol (IP)

About the time that the CUS was being set up, there was growing pressure from British academics for direct connection to the Internet. The delays caused by the increasingly congested email gateway were frustrating, and also they wanted access to file transfer and other services as well as email. In January 1991 the JANET IP Service (JIPS) was set up as a pilot project to host IP traffic by piggy-backing on the existing X.25 network. IP traffic rapidly took over from X.25, leading to a changeover to a fully IP network over the course of the next few years.

Cambridge was part of the initial JIPS pilot, led by the computer scientists, who were most eager to join the Internet. Fortunately, there was a rudimentary ethernet network in Cambridge that could be connected to JIPS. In the Computing Service, those of us working on the CUS became involved, because Phoenix did not at that time have an ethernet connection. (Later on it did, though it never supported email over IP.) One of the things I did was to set up a gateway computer to allow users on one network to access services on the other, in other words, an X.25↔TCP/IP gateway. The service was called *ipgate* and it ran on a computer called *janus* during the transition period when many of the University's institutions still had only X.25 connections, but more and more services were accessible only via TCP/IP.

At the start of the IP service I was asked to set up a small group to coordinate IP activities among various interested parties in the University. This group met regularly under my chairmanship for several years while the use of IP was getting established in the University. We abolished ourselves when the group had outlived its usefulness.

When JIPS was being set up, the powers that be were worried lest it fragment the community into those institutions that had an IP connection, and those that were still using X.25. In particular, they wanted to be sure that there would be no 'them-and-us' divide for email, and for this reason, the use of the SMTP protocol (the Internet's email protocol) was initially banned. Well, *almost* banned; the loophole was that SMTP was 'permitted for experimental purposes'. As email manager on the CUS, I immediately started 'experimenting' by modifying the *Sendmail* configuration so that all mail for non-UK addresses was sent by SMTP. This bypassed the London gateway, and gave a much-needed improvement in performance.

Abandoning Sendmail

The configuration of *Sendmail* was becoming increasingly difficult to manage, so I started to look for a replacement that could provide a better way of dealing with domain ordering and a more flexible way of choosing between IP and X.25 when delivering email. I happened to mention this (in email) to Randy Bush, and he suggested that I take a look at *Smail 3*, an open source MTA that he was running on his hosts. I did so, and decided that, with a few additions to the code, I could make it do what was needed.

It was clear to me that the days of JANET's NRS were numbered, and that in future, the Internet's *Domain Name Service* (DNS) would be where information about host and email domain names would be found. Luckily, JANET's email domains were already being maintained in the DNS in order to facilitate the exchange of email between the two networks. For each JANET email domain such as *uk.ac.cam.phx* there were DNS *Mail Exchange* (MX) records with a reversed name (*phx.cam.ac.uk*), pointing to the London email gateway hosts. This was how Internet users were able to send email to JANET users.

I modified *Smail* to add two configurable features, which of course I turned on in the CUS configuration. When deciding where to send an email, it first looked up the recipient's domain, exactly as supplied by the sender, in the DNS. If nothing was found, the domain was reversed for a second lookup. If the second lookup found nothing, the domain was invalid. Otherwise, the DNS data was used to decide how to deliver the message. If MX records pointing to the London gateway were found, the message was sent over the X.25 network; otherwise it was sent using SMTP over the Internet.

Smail was still being actively maintained and developed. Luckily, my *Smail* patches were accepted by the MTA's author and incorporated into the base code, making it straightforward to upgrade to new releases. In principle, this meant that other UK academic institutions could run it as well, though I am not sure if any ever did.

The modified *Smail* was installed on CUS before there were very many users. We encouraged them, and all subsequent new users, to change to using Internet-style domain names in email addresses. However, the first CUS machines had the appropriate hardware and software for X.25 networking, and at the start, outgoing X.25 deliveries used this. I therefore had to write code to reverse all the domain names in messages that were delivered by that route.

Later, I was able to get rid of this complication by sending such messages over SMTP to the University's central mail switch, which had been implemented by some of my colleagues. This ran an MTA called PP, which had been designed to translate between a number of email protocols, including SMTP, Greybook, and X.400. Although it was a clumsy beast, causing much tearing of the hair at times, we had to keep running it until the X.25 network was no longer in use at Cambridge. This did not happen until the IBM mainframe was turned off in 1995.

A central mail store called Hermes was implemented in 1992 or 1993. This system held a large number of users' mailboxes, which could be accessed either by the remote-access IMAP and POP protocols from users' own workstations, or by logging in to the system and running the *Pine* user agent, which by that time had been installed on the CUS and was preferred to *Elm*. The Hermes system is still running as the University's main email service. Most of the internals have been replaced at some stage, but the users' interface has remained stable.

As Hermes is for email use only, users who log in to the system are not allowed access to a general-purpose command shell. Instead they are presented with a menu of actions such as running Pine, changing their password, and so on. I wrote the first version of the Perl script that implemented this facility.

2017: *At some stage a webmail interface was added to Hermes, at first using local software written by David Carter, but now based on a free software application called Roundcube.*

At first, Hermes ran PP, but this was overkill, and after *Smail* had been successfully running on the CUS for some time, the Hermes managers switched to using it as well. I wrote them a short 'care and feeding' document.

Several years passed. I made more modifications to *Smail* from time to time. The most important of these was a check on the sender addresses in incoming messages. I had noticed that, with the increase in email usage, we were also seeing an increase in the number of messages with bad senders (which at that time was usually caused by misconfiguration rather than malice). If such a message could not be delivered, the bad sender meant that an error report could also not be delivered, and the postmaster (me) had to intervene to sort things out. By rejecting messages whose sender address could not be verified, I pushed the onus of dealing with this problem out to the sending MTA. This kind of checking is now standard MTA practice (along with many other checks for spam and viruses).

Managing the DNS

As well as dealing with email, I was also in charge of the Cambridge DNS zones for some years, acting as the local 'hostmaster'. The JIPS pilot was initially set up by staff associated with the Teaching and Research side of the Computer Laboratory (that is, the computer scientists). Piete Brooks managed the *cam.ac.uk* and *cambridge.ac.uk* zones for a while, until handing them over to me when the Computing Service took over the fledgling IP service. We ran the nameserver software on the CUS for some years (having no other suitable Unix hosts), but eventually dedicated nameservers were acquired.

I insisted from the start that we should make *cam.ac.uk* our main Internet domain, and I pushed for the abolition of the *cambridge.ac.uk* synonyms as soon as possible. We could not just drop them, because various hosts around the University were configured to use and recognize both forms. It took a couple of years, but eventually all the long-form names were deleted.

From the start, however, the ‘reverse zone’, which translates IP addresses into names, translated into the short names. Piete had written an *awk* script to generate the reverse zone automatically from the ‘forward zone’ so that the two could be kept in step. This had its limitations, and did not do as much syntax checking as I wanted, so I wrote a more sophisticated version in Perl, and called it *makezones*. Perl was a new programming language at that time, and this was my learning project, so the result was not the most beautiful code in the world.

I made the *makezones* script public by putting it on our FTP site (then hosted on the CUS), and it was picked up by a number of sites round the world. For a time it was even included in the distribution files for *BIND*, a popular free nameserver implementation. This was the first complete piece of open source software that I released. People continued to use it for many years; to my surprise I received a technical query about it in the middle of 2008.

At Cambridge, *makezones* was in use long after I ceased to be hostmaster. Even while I was still doing the job, the time taken to do the daily updates for the whole of the *cam.ac.uk* zone was getting onerous; some time after I handed the job on, Tony Stoneley implemented an Oracle database application to store the data, and made it possible for administrators in the various University departments to manage their own subsets, instead of having to send requests to the Computing Service. At that point, the majority of *makezones* was no longer needed, but it continued to be used in a cut-down form for the small amount of data that did not come from the database until early in 2008.

The start of Exim

By 1995, JANET’s transition to an IP network was essentially complete, and Phoenix was soon to be turned off, severing Cambridge’s last link with the old way of doing things. We could now concentrate on providing services that were ‘Internet standard’.

Like all its contemporary MTAs, there was very little checking and verification in *Smail*. In the early days of networking, the developers were happy just to get the applications working at all, and the users were academics and researchers who were on the whole benevolent. All MTAs were ‘open relays’ – that is, they would accept any message and deliver it, whatever the recipient addresses were. I used to test the software and the network by sending a message addressed to myself to various external MTAs, and check that it came back successfully. Today, trying to do this would be seen as wicked.

As the use of Internet email increased, and commercial sites started connecting, it became clear that a lot more checking would be needed in future. I had made a start with sender address checking in *Smail*, but it was a very small start. I considered making further modifications, but in the end I decided to begin again from scratch. There were several reasons for this.

Smail was by then eight years old, and written in pre-standard C. The lack of function prototypes had directly caused at least one serious bug. Also, *Smail* had been originally designed as a UUCP mailer. UUCP was an early set of programs and protocols for transferring files, netnews, and email between Unix systems, often over dialup connections (the acronym stands for *Unix-to-Unix Copy Program*). UUCP email addresses were colloquially known as ‘bang paths’ because they were a list of hosts through which the mail was to be routed, followed by the user name, with the different components being delimited by exclamation marks (pronounced by some as ‘bang’), for example, *!bigsite!foovax!barbox!me*. Because the Computing Service came relatively late to Unix, by which time ‘proper’ networking was available, we never had to tangle with UUCP.

By the time I started using *Smail* it had SMTP interfaces, and supported Internet email addresses, but the code still carried a lot of UUCP baggage, including processing addresses internally in UUCP format. Another area I felt needed improvement was the way retries were handled when a message could not be delivered at the first attempt. *Smail* allowed a fixed retry time to be set for different email domains, but that was all. Every message was retried according to these rules, and there was no facility for increasing the intervals between retries as time passed. I wanted to arrange retrying by host rather than by message – I saw no point in trying to send to a host when a previous attempt at connection for a different message had just failed – and I wanted to be able to use variable retry time intervals. I also wanted better pattern matching facilities for use in configuring the MTA.

Overall, I wanted to retain *Smail*'s general approach, but lose the historical UUCP baggage, and add facilities, while generalising wherever possible. I also made the decision to assume a 'modern' operating system environment, a Standard C compiler and runtime system, and a permanent connection to a reasonably fast TCP/IP network, where most email would be delivered almost immediately. (I could see from the logs that over 95% of CUS email was delivered at the first attempt.)

In March 1995 I started writing a new MTA, following these ideas. I was not sure if this would ever be a finished product, so I called it *Experimental Internet Mailer* (Exim). Later, my attention was drawn to a number of other enterprises that shared the name, including a car repair service in the United States and an estate agent in the south of France. There was also a bank that used the name as an abbreviation for 'export-import', which is actually quite appropriate for an MTA. Better still, someone discovered that the adjective 'eximious' means 'excellent, distinguished, eminent'.

At first, Exim was a private project that I worked on when I could. Management did not learn of it till much later. The first code was written on the CUS, but not long after I started I inherited a secondhand Solaris workstation that was surplus to the requirements of the central mail support group, and that is where the main development happened. Later, my workstation was upgraded to a bigger and better Solaris box, but the final upgrade in 2003 was to a Linux workstation, which I ran for the four years before I retired.

By November 1995, Exim was just about able to send and receive emails. I happened to mention to Piete Brooks, who was managing the computer scientists' email system, that I was working on a new MTA. Piete's hosts were running PP, but now that the X.25 network had gone, he wanted to change to something else, so he asked if he could have a copy of Exim to experiment with. I said that I could not yet provide one, because I had not written any documentation. 'I don't want the documentation, I want the code.' was Piete's response. Nevertheless, I made him wait for a few days while I wrote a preliminary edition of the Exim manual.

One Friday afternoon in late November I packaged it all up, sent it to Piete to play with, and went away for the weekend. I came in on the Monday morning and there was an email from Piete: 'I've put it into service.' Wow! We had a very exciting, hairy week. I am forever grateful to the computer scientists for putting up with a very bumpy email service without complaint, just before Christmas. I do not think we lost any messages, but Exim went through a number of versions in rapid succession, and there were certainly delays.

Things soon settled down, however. I continued to develop the code, and around Easter time we were able to replace *Smail* with Exim on the CUS. Later still, the Hermes mail store, and eventually the central mail servers, switched over to Exim. Meanwhile, Piete, who was active on the UK postmasters mailing lists, started telling others about it, and I started to receive more requests for copies, so I began to put each new release on a public FTP site. In June 1996 I gave a talk on Exim at a ‘Campus Mail Day’ held in Aberdeen, which caused a number of other academic institutions to start using it.

The early releases of Exim were numbered 0.00, 0.01, and so on, and one day, when we had got to 0.57, I had a phone call from The Welding Institute (now branded as TWI), a research-based company near Cambridge. The caller was their postmaster, who wanted to run Exim on their mail servers, but his boss would not allow any software with a version number less than 1.00 to be used. Fortunately, a new release was being prepared, so I released it a few days later as 1.58, and everybody was happy.

It turned out that I was not the only one to see a need for a new, freely available MTA. Both *Postfix* and *qmail* were released at much the same time as Exim. This was a repeat of what had happened when I was working on music typesetting. Exim was initially released under a locally-written licence, but this was changed to the GNU General Public Licence (GPL) after I received a request for this from Richard Stallman.

Growth in the use of Exim

I was quite surprised how quickly Exim spread without the use of any serious advertising. A mailing list was soon needed. I was able to create one on the University’s list system, but this required manual maintenance, because that system was designed for lists of designated people, not for arbitrary, self-managed subscriptions.

It was not very long before a web site was suggested, and here I was fortunate in that Nigel Metheringham, then working for Planet Online Ltd, offered to run this together with a ‘proper’ mailing list, and he persuaded his employers to contribute a host on which to do it. Planet, and its successor, Energis Squared, continued to provide this support until 2004, when the Computing Service set up a host called *sesame* for open source development. This system (upgraded to new hardware and renamed more than once) has grown over time, and now holds the source code repository, the mailing lists, the bug reporting system, and the wiki. Nigel is still active in helping to run it.

2017: *Some time after I stopped working on Exim, the new developers moved the source to GitHub. The other facilities remain on a Computing Service system that is now called* hummus, *generically accessed as* exim.org.

I had very little experience of Unix other than SunOS/Solaris when I started work on Exim, but I assumed that it would be run on other operating systems, so I created a build framework that allowed for this. Over time, as it was tried on various systems, I was sent patches for many different environments. As I explored the differences, I began to understand just how many-headed ‘Unix’ actually is. One particular issue that amazed me was that there seemed to be no common way of obtaining a value for the system load. I had assumed there would be a standard system call for ‘read system load’, but it turns out that there are three or four different ways of doing it.

Several other pitfalls were discovered and papered over. Systems vary in their default action when an interrupt that occurs in the middle of a system call is caught by the standard *signal()* function and the handling function returns normally. Some resume the interrupted system call; others terminate it with an error code. Fortunately, there is a way to override the default and specify which action is required; in the end Exim was changed to be explicit and never to rely on the default. I also learned a lot about file locking, especially in connection with the Network File System (NFS), which allows files on one host to be accessed from others. The details are far too arcane to be included here.

In 1997 or 1998 (I have failed to track down the exact date or release) Exim was ported to Linux, which by then had become an operating system that could be used in production environments. At that point I was able to say that Exim would run on ‘most Unix and Unix-like’ operating systems. Later still, it was ported to the Cygwin environment, which does its best to mimic Unix on Windows systems. I do not think it is fully functional in this environment, but to my amazement there were (and probably still are) a number of users.

Other contributors

As the use of Exim spread, I began to receive code contributions from many people. Some were quite minor, but there were also some substantial pieces of code, for example, the first implementation of IPv6 support by Philip Blundell, the integrated interface to virus and spam scanning by Tom Kistner, support for the *Sieve* filtering system by Michael Haardt, and support for a number of different databases by several others. Before the source was stored in a CVS repository, I used to review all the contributions before applying them to the code. Later, a number of trusted contributors were given commit access to the repository.

Design re-thinks

During the twelve years that I worked on Exim there were a number of major re-designs of different parts of the program. For most of the time, new releases were backwards compatible with their predecessors, but releases 3.00 and 4.00 broke new ground by introducing new, incompatible features. These were always concerned with the way in which the MTA operated and was configured; the actual sending and receiving of messages hardly changed at all. My initial design turned out to be too rigid in several ways, and the changes all aimed towards increasing flexibility. Here are some examples:

- Exim originally supported only a few kinds of stored data, such as plain text and DBM files, and the code was not separated from the rest of the program. This was changed to create an internal interface for generalized data lookup, and subsequently interfaces to a number of different kinds of database were added.
- The original email routeing copied the way *Smail* did it, by splitting addresses into ‘local’ and ‘remote’ right at the start, and handling each kind in different ways. As email developed, the boundary between ‘local’ and ‘remote’ became blurred, leading to untidy and messy configurations. For Exim 3, the distinction was abolished, and all addresses were handled in the same manner.
- Exim 4 introduced *Access Control Lists* (ACLs) as a generalized means of controlling access to email services, replacing a large number of separate control options, whose interactions with each other were never very clear.

Exim and ISPs

Early in 1999 I was contacted by an ISP in St Albans, who asked if I could run an in-house Exim training course for them. I was surprised to learn that a commercial organization was running Exim under Linux. They explained that it was simple economics: even after taking into account the cost of employing maintenance staff, using standard PC-style boxes with a free operating system and free application software was much cheaper than proprietary hardware such as Suns, or commercial software such as Windows.

It was not long before other ISPs also picked up Exim. One of them paid me to add some special code so that Exim could interwork with their parochial environment, and so that I could do this on their hosts from home, they installed a 64K leased line to my house in 1999. This was the first time that I had been online at home as I had never felt the need to set up a dialup facility. Instead, I used to carry a floppy disc with me so that I could transfer files between home and work. A dedicated 64K line was far superior to the dialup connections that everybody used at that time for home connections, but this lasted for only a few years until broadband came along. By 2004, when the leased line was replaced by a broadband connection, 64K was old hat.

2017: *Needless to say, home broadband connections have got much faster still since 2009.*

Exim training

Soon after I had developed training material and delivered a course for the ISP in St Albans, an academic Exim user asked if a public course could be mounted, and in September 1999 we ran the first Exim course at Robinson College, in Cambridge. This was a one-day, lunchtime-to-lunchtime course, and to our surprise it attracted 120 attendees, completely filling the lecture theatre and making a healthy profit for the Computing Service. Both academic and commercial organizations sent people to the course, in roughly equal numbers.

The following March we ran a follow-up ‘advanced’ course of the same length, and from 2001 onwards, an Exim course in July became a regular event. Over the years the length increased, at first to two days, with invited speakers to break the monotony of listening to me all the time. Then for the final three years before I retired, we expanded to three days and added practical exercises in one of the Computing Service’s classrooms.

Maggie Carr and her team handled the administration of all these courses brilliantly, and Robinson College looked after us very well. Those that attended often complained that the food was too good. When we extended to two days, we gave the attendees a taste of the ‘Cambridge experience’ by taking them punting, and in later years laid on a guided walk back from the practical classroom to the College. For those that came from commercial organizations, this was a different kind of training course to those they normally attended.

The numbers varied over the years. There was a surge for the first course that covered Exim 4, a release that had many new features. Many of those who had attended previously came again that year. Although Maggie worried sometimes that not enough people were going to sign up to cover our minimum booking at Robinson, her fears always proved false.

One brave person, David McLaughlin from ETH in Switzerland, attended every single course. I asked him how he could stand listening to me saying the same things year after year, but he assured me that it was never the same twice, and that he always went home with new ideas – not only from the lectures, but also from talking to other Exim users. It was inevitable that David would eventually be one of the external speakers. We gave him a present for stamina at the final dinner in 2007.

Exim in Africa

In April 2000, Randy Bush contacted me to ask if I had any teaching material for email systems. He explained that a new organization, the *African Network Operators Group* (AfNOG) was being set up, and that a five-day technical workshop was happening as part of the first meeting, which was to be held in Cape Town in May (that is, the following month). I consulted my diary, and was able to reply that not only was he welcome to my material on Exim, but that I would be happy to go to Cape Town and deliver it, though I would have to leave just before the end of the workshop. It would not cost them anything for accommodation, because I could stay with friends. I mentioned this trip on the Exim mailing list, and within a couple days I received sponsorship for the airfare.

Thus began my relationship with AfNOG. When I arrived in Cape Town only a few weeks later, I found an international team of instructors who had been running workshops around the world with the help of the Network Startup Resource Center (NSRC), based at the University of Oregon. The NSRC originated out of a volunteer effort by Randy to set up networking in Southern Africa in 1988, and it became a formal organization in 1992. In other parts of the world, network operators (academic, governmental, and commercial) had formed organizations such as the North American Network Operators Group (NANOG), and the formation of AfNOG was seen as a way to promote networking in Africa.

That first meeting had two tracks in the workshop, network infrastructure and network applications. There were about 20 students in each, drawn from all over Africa. Some had had great difficulty in travelling to Cape Town, whereas all I had to do was to get on a non-stop overnight flight from London. From other African countries the routeing was sometimes very tortuous, and one student had even hitched a ride on a freight transport plane.

They were all very eager to learn; in all the breaks they would buttonhole the instructors to ask questions. At times I felt like a sponge being sucked dry. Although the workshop was ‘hands on’, with each student seated at a PC running the FreeBSD operating system, I presented a subset of my Exim material in lecture form only, without any exercises, because that was all I had. The other instructors all had laptops which they used for showing slides and for getting online when not actually presenting, but I was still using overhead transparencies, and had to log on to the network management host in order to get access to my email.

I missed the second AfNOG meeting in Accra the following year, but in the years after that I went to Lomé (Togo), Kampala (Uganda), Dakar (Senegal), and Maputo (Mozambique). I bought a laptop before Lomé so that I could do without an overhead projector, but I did not manage to get its wireless interface working till the following year. I also had time to devise some practical exercises that involved installing Exim and running tests as its configuration was modified in various ways. Over the years these were refined and expanded.

Classroom at AfNOG 2008

After a gap of three years, I went to my final AfNOG workshop in Rabat (Morocco) in 2008, seven months after I had retired from my job and from maintaining Exim. It was quite a contrast to the original Cape Town meeting, having expanded to four tracks (one in French), teaching 100 students, many of whom were women, from 23 African countries. A number of the original non-African instructors were there, but now there were plenty of African instructors too, some of them having been students at previous workshops. I found the 2008 students to be much more experienced than the original ones had been, and there was lively interaction in our classroom, something that had not happened in earlier years.

I was always something of an anomaly at AfNOG because I am a developer and very much a specialist. The other instructors were far more experienced than I in installing and running networking applications. Nevertheless, the students always seemed to appreciate my efforts. In Morocco, several of them called me ‘Uncle Phil’, which I took as a great compliment.

After some of the workshops there was time for bit of tourism. In Dakar a group of us, mostly with white faces, but also including a Ugandan, visited Gorée Island from where slaves were transported centuries ago. There were stalls selling souvenirs. Seeing a black face, the sellers called out in the local African language, and could not understand why the Ugandan paid absolutely no attention! I had learned a single phrase in Wolof, roughly meaning ‘Push off, I don’t want to buy’, and this always elicited smiles whenever I used it.

Uganda was particularly memorable for the tourist activities. Not only did some of us visit the source of the Nile where it leaves Lake Victoria, but we also were able to attend the *Royal Ascot Goat Races* in Kampala – one of the social events of the year, and tremendous fun. How do you get goats to race? They have ways...

Royal Ascot Goat Race, 2003

AfNOG is unusual in that it operates across the Anglophone–Francophone divide in Africa (and has also held one meeting in Portuguese-speaking Mozambique). I think it has been very successful in pushing forward networking in Africa, and also in forging friendships between networking personnel across the continent. To a large extent, Africa has been starved of information, and modern communications technology is one way of trying to redress this.

Other continents

2004 and 2005 were years in which I did a ridiculous amount of travelling, in Europe, North America, Asia, and Australasia, in order to deliver one-day or two-day Exim training courses, and to speak at conferences. I went to the *South Asia Network Operator’s Group* (SANOG) meetings in Kathmandu (Nepal) and Thimpu (Bhutan), where workshops like AfNOG’s were being run, again with the assistance of the NSRC. Both meetings were in the monsoon season, so the high mountains were not visible, but after the 2004 meeting a number of us took the ‘Everest Flight’ to view the high peaks from above the clouds, and I did also manage a three-day walk with a guide between villages on the edge of the Kathmandu valley. In 2005 I had a couple of days’ touring in Bhutan.

In 2004 I was invited to teach at a Linux conference in Australia, and the following year at a meeting of the New Zealand Network Operators Group (NZNOG). I was able to work the first of these into a three-week trip with my wife, which included visits to relatives and friends, and after the New Zealand meeting I spent a week touring the North Island and stretching my legs on some of the wonderful day walks.

Debian’s use of Exim

When Debian switched to Exim as the MTA for their Linux distribution, it had both good and bad effects from my point of view. On the positive side, Exim acquired a large number of new users, but on the negative side, many of those users were new to system administration and even new to Unix-like systems. When I started to develop Exim, I expected it to be used by experienced system administrators who would be installing it from source on moderate-sized Unix systems. The documentation was written with such a readership in mind.

After the first Debian stable version containing Exim was released in March 1999, there was a big increase in the number of ‘newbie’ questions on the mailing list. Fortunately, there were a number of helpful subscribers who were willing to jump in and answer these questions, so it was not a catastrophe, but it changed the nature of things a bit, and ultimately led to an Exim developers’ mailing list splitting off from the Exim users’ list.

Debian’s long release cycle meant that it continued to ship with Exim 3 for several years after Exim 4 was released. During this time, each new user of Debian became a new user of Exim 3. This was not good, because all the experienced Exim users were on Exim 4 and were rapidly forgetting Exim 3. When maintenance of Exim 3 ceased, Debian had to take responsibility for back-porting security patches from Exim 4 to Exim 3. It was a relief when Debian finally did move to Exim 4, and there was no longer a need to keep referring Debian users back to Debian for answers to their questions.

The Exim books

The documentation distributed with Exim was, and is, very much a reference manual. When people with less experience started trying to administer Exim, the lack of introductory documentation was often cited. This was one of the reasons why I wrote the first book, *Exim: the Mail Transfer Agent*, as a kind of tutorial. I found it immensely hard work, much harder than I had anticipated. I was fortunate in having Andy Oram as my editor at O'Reilly. He helped me find a way to arrange the information in a sensible order, and made many valuable comments about the content and the writing style. I wrote the book using restricted SGCAL markup that I then translated into DocBook markup to send to O'Reilly. However, it took a long time to get the book finished and it was not published until the summer of 2001.

This was very poor timing, because Exim 4 was then in development, and was released early in 2002. The differences between Exim 3 and Exim 4 were such that the book was then obsolete in significant parts. I started to revise the text, but unfortunately O'Reilly could not justify the cost of taking on a new edition. However, they were happy to return the copyright to me so that I could seek a new publisher. When I asked on the mailing list about this, I was pleased and surprised to receive an offer from Niall Mansfield of UIT (a company in Cambridge), who was literally 'just down the road'.

UIT published *The Exim SMTP Mail Server: Official Guide for Release 4* in 2003. This time I typeset the book myself, using SGCAL, and sent camera-ready copy to the publisher in PDF format. In some ways it was a relief to write and punctuate in 'English English' after having had to try to be American for O'Reilly. In 2007 I updated this book yet again, and UIT published a second edition.

One of the features of the AfNOG (and similar) workshops in developing countries is that the students are given technical books donated by the publishers. UIT, being a very small company, was unable to do this, but I was pleased that Niall offered Exim books at cost, and this offer was taken up for several AfNOG workshops.

Exim in retrospect

The Exim project (to give a name to something that was never any kind of formal organization) has grown much bigger and lasted much longer than I ever expected. I was writing an application for the University of Cambridge. I hoped that some other UK academic institutions, and perhaps a few outside the UK, might pick it up and find it useful. I never expected commercial sites to get involved, nor for it to become the default MTA in any operating systems. In short, I did not foresee that it would grow into the fully-fledged open source development project that it has.

I have been asked what is special about Exim, compared to other MTAs. This is a question that I usually duck, pointing out that I know almost nothing about other MTAs. Nevertheless from what I do know, I think that one of Exim's strengths is its flexibility. This just 'grew' from the initial design as people asked for new facilities. The result has been likened to a Swiss Army Knife; it may not be the most efficient at certain specific tasks, but it can be adapted to many different requirements. I am pleased that Exim can be found in all kinds of environments, from one-user personal computers to very large, multi-host mail servers, and it seems to perform well in all of them. Providing complete, detailed documentation with each release is something I felt was important, because I know how frustrating it is to try to use software for which there is only fragmentary information.

I was 51 when I started to write Exim, nearer the end than the start of my career. It certainly made the final decade an exhilarating period for me. I learnt a lot, met interesting people, made new friends, and visited some exotic places. However, by the time I retired, I had been working on Exim for over twelve years, and giving it up felt right. More than a decade on one project is long enough.

2017: *That last sentence has come back to bite me: I am still working on PCRE (see the next section), nineteen years after I started to write it.*

Regular expressions and PCRE

The phrase 'regular expression' originates in pure mathematics, where it has a very precise meaning. In computer software it refers to patterns that are used for matching strings of text. The early implementations were indeed based on mathematical regular expressions, but as pattern matching developed, features were added that deviated from the original theoretical basis. For modern types of pattern, the name 'regular expression' is no longer accurate in the mathematical sense, but it is so widely used that, in effect, its meaning has changed.

Pattern matching in text editors

My introduction to regular expressions happened some time in the 1970s when Martin Richards showed me a clever algorithm for matching simple types of pattern. These regular expressions had a different syntax to those that are common these days, and were quite restricted in what they could do. The interesting part about the algorithm was that the amount of space required to hold the compiled form of an expression was known in advance.

I implemented this algorithm in IBM Assembler in order to add regular expression support to the ZED text editor, and subsequently I re-implemented it in BCPL and then in C for the E and NE editors that succeeded ZED. Meanwhile, simple regular expressions had become a feature of Unix systems. Around 1986 Henry Spencer, at the University of Toronto, released a library that was widely used by software of the time.

Unix-style patterns

In the 1990s, the arrival of Perl and other similar languages seems to have stimulated the development of regular expressions, and the emergence of the World-Wide Web provided lots of potential applications. In 1997, Jeffrey Friedl's book *Mastering Regular Expressions* was an unexpected best-seller. There have since been two more editions, in 2002 and 2005, each heavily updated, which indicates the rate at which development was happening in that period.

I learned some basic Unix-style regular expressions when I started using Unix seriously, but it was not until I began to work on Exim that they really came back into my life. *Smail* had only very primitive string matching capabilities (for example, `*.a.b` to match all hosts whose names ended in `.a.b`). I wanted more flexibility for Exim, and regular expressions were the obvious answer. The only freestanding implementation I could find was Henry Spencer's library, so the early versions of Exim used that.

I had used Perl quite a bit by that time, and liked its extended pattern matching features. The restrictions of the Spencer library began to irk, so I looked for an alternative, but found none. I even considered stealing code from Perl itself, but when I looked at it, it was too tightly bound into the Perl coding environment to be easily cut out. In the end, I solved the problem the way programmers generally solve their problems: by writing something myself.

Development of PCRE

In the summer of 1998 I wrote the first version of the *Perl-Compatible Regular Expression* (PCRE) library. I knew it was going to be tricky, so I did most of the early work at home, where I could not be disturbed. I took care to write C code that was as efficient as I could possibly make it, and was pleased that the result beat the Spencer library on patterns that they both supported. When it was done, I bundled it into the next release of Exim, but I also released it as a standalone library in its own right, because I knew it had other uses. I soon changed the NE text editor so that it could be compiled with PCRE, and sometime later abandoned its old regular expression features altogether.

2017: *As a test for PCRE, I had written a 'grep' program that used PCRE. This was released with the library, the result of which was that people started using it and so I had to develop it to support many of the features of other 'grep's.*

PCRE was originally released with a home-brewed licence that allowed anybody to do anything with it, as long as they put an acknowledgement of its source somewhere in their code or documentation. This turned out not to be a good idea. I received a very irate (and also rather sad) email from a man in the USA. His Windows system had been infected by a virus which had destroyed some of his valuable data concerning his son's illness. On digging around to find out about the virus he had come across my name in an acknowledgement, and assumed that I (and indirectly the University) were responsible.

I was surprised to learn that virus writers paid any attention to licenses. I tried to explain to him that all I had done was create a general purpose tool, and though I regretted its use for evil purposes, I could not do anything to stop that happening. I did, though, change the licence wording to the standard 'BSD' licence, which does not require acknowledgements. (I then received a complaint from another North American about 'mis-spelling' the word 'licence'.)

The original PCRE library was compatible with Perl 4, and built only as a static library, as I had no experience of shared libraries. I did not expect there to be many releases, so I numbered it with just one decimal fraction, release 1.0. How wrong I was! Although there was no advertising, people found PCRE and to my surprise it found its way into quite a few well-known software packages, including the *Apache* web server. I was particularly gratified when I heard that it was being used by the *Postfix* MTA. I realized then that I had almost accidentally filled a serious need. After a while, operating system distributions started including PCRE as part of their offerings, so nowadays a programmer can expect to find it available almost as standard. For this reason, Exim stopped bundling its own version in 2008.

More users find more bugs, but as PCRE was used more and more, I also started to receive requests for enhancements of various kinds. The code itself was enhanced to support Unicode character sets and the extra pattern matching features of Perl 5. The build system was originally just a *Makefile* that I wrote by hand. For version 3.0 (February 2000) I started using *autoconf* to generate a standard *configure* file, and *libtool* to allow both static and dynamic libraries to be built.

The original plan had been to keep PCRE's facilities as close to those of Perl as possible. This did not last. After some years, I started adding features that went beyond what Perl provided. Amongst these was support for recursion inside a regular expression, based on a patch supplied by Robin Houston. Named subpatterns, a Python feature, were also added, as well as several other facilities taken from other regular expression implementations. The wheel has now come full circle, with the latest versions of Perl now offering some of these facilities. I suspect that Jeffrey Friedl's book, which compares a number of implementations, has been instrumental in causing implementors to look at what others are doing.

An additional way of matching

Friedl describes two basic ways of matching a pattern, which he refers to as NFA (from *Nondeterministic Finite Automaton*) and DFA (from *Deterministic Finite Automaton*). Purists will argue that, like the term 'regular expression' itself, these names are inaccurate, but they are a useful shorthand. They were in use before Friedl wrote his book; he gives the alternative names 'regex-driven' and 'text-driven' as perhaps more accurately descriptive.

PCRE and Perl use NFA algorithms, which could also be called 'depth first' because that is the way they search the tree of matching possibilities. Failure at any point in the tree causes backtracking, both in the tree and in the subject string. Success is recorded when the first possible match is found. By contrast, a DFA algorithm searches 'breadth first', that is, it considers all possibilities at once as it moves along the subject string. There is no backtracking, and the characters of the subject are used only once. Matching continues until the longest match is found.

In 2005, it occurred to me that I could use the existing form of a compiled pattern in PCRE to provide a DFA matching function in addition to the standard NFA function. This was added for release 6.0. It is not Perl-compatible, of course, but I thought some people might find it useful because it can do some things that the NFA algorithm cannot. Because it scans the subject just once, it can read it in parts, and it finds all possible matches (starting at a given point) rather than just the first. The downside is that it is not possible to extract partial strings from whatever is matched. I have had some bug reports, so the new function is being used, but I suspect it has limited appeal.

PCRE under Windows

At some point, people started sending modifications to make it possible to build PCRE on Windows systems. I have never used Windows, let alone programmed for it, so I knew nothing about the way its libraries worked. After some years of adding whatever people sent me, I was eventually persuaded, in 2006, to abandon my hand-crafted *Makefile* in favour of *automake*. An enthusiastic PCRE user did most of this conversion work. It certainly helped standardise the build process for Windows, but only in some cases. Another enthusiast provided support for *cmake*, which is another application-building system that some people find more suitable for Windows. PCRE now supports both building methods.

PCRE with C++

I was also sent several different implementations of C++ wrappers for the PCRE functions, which I made available in a ‘contributions’ directory on the FTP site. These were never maintained, and so as PCRE developed, they often stopped working. In 2005 I accepted an offer from Craig Silverstein at Google to supply *and maintain* a C++ wrapper. This was integrated into the PCRE distribution and seems to be happily used. The folks at Google are quick to respond to bug reports, and to provide maintenance patches, so this has worked well.

PCRE in hindsight

A decade of continuous development has passed since I started work on PCRE, and there are still regular new releases, both for fixing bugs and for adding features. Along the way, I have learned rather a lot about pattern matching. As with Exim, I never expected it to be this active for so long. Although conceived as an add-on for Exim, it is perhaps now an even more important package because of its widespread use in many different applications and operating systems.

2017: Non-stop development

The development of PCRE has continued apace since I first wrote this document. In 2011 Zoltán Herczeg’s just-in-time (JIT) compiler support was integrated into the main code base. Zoltán has continued to maintain and develop this code, and in 2012 he added support for 16-bit character strings to PCRE. Later that same year, Christian Persch added support for 32-bit character strings.

By 2014 it was clear that the API (programming interface) was not capable of much further extension, and so I created a new incompatible version of the code with a re-designed API, called PCRE2, which was released early in 2015. The C++ support from Google had withered away when Craig Silverstein left, so PCRE2 offers only a C library. I took the opportunity of tidying up the code, particularly with regard to data types, but the main logic was unaltered.

More recently, in 2016, I refactored the pattern compiling code to reduce duplication and improve the way forward references are handled. In 2017 I *refactored the Perl-compatible matching code so that it no longer uses recursive subroutine calls (and therefore the system stack) to remember backtracking positions. Instead, heap memory is used when necessary. This removes the long-standing issue of stack overflows for certain types of pattern matching.*

Recent development of PCRE2 has been aided by the advent of automatic testing by fuzzing, which several sites have undertaken.

10. And finally

I retired from my job in the Computing Service at the end of September, 2007. The decision had been made at the start of the year, and I spent some time ‘putting my affairs in order’, in particular, converting the Exim documentation system to something a little less parochial, and re-implementing the test suite to be more portable.

I knew there were several people who had a reasonable knowledge of how the code of Exim works, and there was also a large user population, so I thought there was enough momentum to keep the project going. I was certain, however, that until I actually did go, nobody else would start to take things over. I have remained on the development mailing list, and was pleased to see that people occasionally worked on the code. As I write this, a second ‘post-Philip’ release is being tested.

Although I am not spending nearly as much time computing as I did, I have not given up completely. For the time being, I continue to maintain PCRE, and in 2008/9 I gave PMW a long-awaited spring clean and added a few new facilities. I had to add some extra features to SDop so that I could use it for the PMW manual, which had previously been prepared using SGCAL.

At the moment, I am not working on any new projects, but who knows what the future may bring?

2017: *The work on PCRE and then PCRE2 has kept me busy enough, with occasional maintenance of other software, so I have not started any new large projects. Exim continues to be maintained and developed, which is gratifying.*

11. Publications

(1) *Numerical Studies of Stratified Shear Flows*, Ph.D. Thesis, University of Cambridge, 1970.

(2) *The effect of viscosity and heat conduction on internal gravity waves at a critical level*, Journal of Fluid Mechanics, **30** (1967).

(3) *Numerical studies of the stability of inviscid stratified shear flows*, Journal of Fluid Mechanics, **57** (1972).

(4) *Some comments on Fortran systems*, (co-author with A.J.M. Stoneley and J.Larmouth), Software – Practice and Experience, **3** (1973).

(5) *A general purpose text editor for OS/360*, Software – Practice and Experience, **4** (1974).

(6) *Development of the ZED text editor*, Software – Practice and Experience, **10** (1980).

(7) *Resource allocation and job scheduling*, Technical Report No. 13, University of Cambridge Computer Laboratory (1980).

(8) *PARROT – a replacement for TCAM*, (co-author with A.J.M. Stoneley), Technical Report No. 5, University of Cambridge Computer Laboratory (1976). Also published in the proceedings of the 1976 SEAS Anniversary Meeting.

(9) *SEAS 77 Delegate Information System*, proceedings of the 1977 SEAS Anniversary Meeting.

(10) *Help Numerical: The Cambridge Interactive Documentation System for Numerical Methods*, (co-author with M.R. O'Donohoe), in *Production and Assessment of Numerical Software*, Ed. M.A. Hennell and L.M. Delves, Academic Press (1980).

(11) *The Development of Text Editing at Cambridge*, IUCC Bulletin, **1** (1979).

(12) *HASP 'IBM 1130' multileaving remote job entry protocol with extensions as used on the University of Cambridge IBM 370/165*, (co-author with M.R.A. Oakley), Technical Report No. 12, University of Cambridge Computer Laboratory (1979).

(13) *Specification of the E Text Editor*, University of Cambridge Computer Laboratory (6th edition, 1990).

(14) *The Phoenix/MVS version of Norcroft C*, University of Cambridge Computer Laboratory (5th edition, 1992).

(15) *Specification of the NE Text Editor*, University of Cambridge Computer Laboratory (1st edition, 1994).

(16) *Philip's Music Scribe: a Music Typesetting Program*, privately published (8th edition, 1996).

(17) *Philip's Music Scribe*, chapter 18 in *Beyond MIDI: The Handbook of Musical Codes*, MIT Press (1997, ISBN 0-262-19394-9).

(18) *Philip's Music Writer (PMW): A Music Typesetting Program*, distributed electronically with the current versions of the program (2009 onwards).

(19) *Specification of the Exim Mail Transfer Agent*, University of Cambridge Computer Laboratory (many editions, electronically distributed with the program).

(20) *Exim The Mail Transfer Agent*, O'Reilly & Associates Inc., ISBN 0-596-00098-7 (1st edition, June 2001).

(21) *The Exim SMTP Mail Server*, UIT Cambridge, ISBN 0-9544529-0-9 (1st edition, June 2003, 2nd edition, 2007).

Index

Colophon

This document was produced almost entirely using software written by the author, running on a Linux system. The source file was created as plain text using the *NE* text editor. The text contained mark-up so that it could be processed by *xfpt* into DocBook XML. This in turn was processed by *SDoP* to create a PostScript file of page images. Finally, the *ps2pdf* command that is part of *GhostScript* (not written by me) was used to convert the PostScript file into a PDF.